Reinforcement Learning: An Overview[1]

Kevin P. Murphy

December 9, 2024

[1]Parts of this monograph are borrowed from chapters 34 and 35 of my textbook [Mur23]. However, I have added a lot of new material, so this text supercedes those chapters. Thanks to Lihong Li, who wrote Section 5.4 and parts of Section 1.4, and Pablo Samuel Castro, who proof-read a draft of this manuscript.

强化学习：概述 [1]

Kevin P. Murphy

2024 年 12 月 9 日

[1]本书的部分内容借鉴自我的教科书第34章和第35章[Mur23]。然而，我添加了大量新材料，因此本文取代了那些章节。感谢李红撰写了第 5.4 节和第 1.4 节的部分内容，以及 Pablo Samuel Castro，他审阅了本文稿的草稿。

Chapter 1

Introduction

1.1 Sequential decision making

Reinforcement learning or **RL** is a class of methods for solving various kinds of sequential decision making tasks. In such tasks, we want to design an **agent** that interacts with an external **environment**. The agent maintains an internal state s_t, which it passes to its **policy** π to choose an action $a_t = \pi(s_t)$. The environment responds by sending back an observation o_{t+1}, which the agent uses to update its internal state using the state-update function $s_{t+1} = U(s_t, a_t, o_{t+1})$. See Figure 1.1 for an illustration.

1.1.1 Problem definition

The goal of the agent is to choose a policy π so as to maximize the sum of expected rewards:

$$V_\pi(s_0) = \mathbb{E}_{p(a_0,s_1,a_1,\ldots,a_T,s_T|s_0,\pi)}\left[\sum_{t=0}^{T} R(s_t, a_t)|s_0\right] \tag{1.1}$$

where s_0 is the agent's initial state, $R(s_t, a_t)$ is the **reward function** that the agent uses to measure the value of performing an action in a given state, $V_\pi(s_0)$ is the **value function** for policy π evaluated at s_0, and the expectation is wrt

$$p(a_0, s_1, a_1, \ldots, a_T, s_T|s_0, \pi) = \pi(a_0|s_0)p_{\text{env}}(o_1|a_0)\delta(s_1 = U(s_0, a_0, o_1)) \tag{1.2}$$
$$\times \pi(a_1|s_1)p_{\text{env}}(o_2|a_1, o_1)\delta(s_2 = U(s_1, a_1, o_2)) \tag{1.3}$$
$$\times \pi(a_2|s_2)p_{\text{env}}(o_3|a_{1:2}, o_{1:2})\delta(s_3 = U(s_2, a_2, o_3))\ldots \tag{1.4}$$

where p_{env} is the environment's distribution over observations (which is usually unknown). We define the optimal policy as

$$\pi^* = \arg\max_\pi \mathbb{E}_{p_0(s_0)}\left[V_\pi(s_0)\right] \tag{1.5}$$

Note that picking a policy to maximize the sum of expected rewards is an instance of the **maximum expected utility** principle. There are various ways to design or learn an optimal policy, depending on the assumptions we make about the environment, and the form of the agent. We will discuss some of these options below.

1.1.2 Universal model

A generic representation for sequential decision making problems (which is an extended version of the "universal modeling framework" proposed in [Pow22]) is shown in Figure 1.2. Here we have assumed the

第一章

简介

1.1 顺序决策

强化学习 或 **RL** 是一种用于解决各种类型顺序决策任务的方法。在这种任务中，我们希望设计一个 **智能体**，使其与外部 **环境** 交互。智能体保持一个内部状态 s_t，并将其传递给其 **策略** π 以选择一个动作 $a_t = \pi(s_t)$。环境通过发送回一个观察 o_{t+1} 来响应，智能体使用状态更新函数 $s_{t+1} = U(s_t, a_t, o_{t+1})$ 来更新其内部状态。见图 1.1。

1.1.1 问题定义

代理的目标是选择一个策略 π 以最大化预期奖励的总和：

$$V_\pi(s_0) = \mathbb{E}_{p(a_0,s_1,a_1,\ldots,a_T,s_T|s_0,\pi)}\left[\sum_{t=0}^{T} R(s_t, a_t)|s_0\right] \tag{1.1}$$

s_0 是代理的初始状态，$R(s_t, a_t)$ 是代理用来衡量在给定状态下执行动作的价值的**奖励函数**，$V_\pi(s_0)$ 是评估在 s_0 处的策略**的**价值函数，期望值是相对于

$$\begin{aligned} p(a_0, s_1, a_1, \ldots, a_T, s_T|s_0, \pi) &= \pi(a_0|s_0)p_{\text{env}}(o_1|a_0)\delta(s_1 = U(s_0, a_0, o_1)) && (1.2)\\ &\times \pi(a_1|s_1)p_{\text{env}}(o_2|a_1, o_1)\delta(s_2 = U(s_1, a_1, o_2)) && (1.3)\\ &\times \pi(a_2|s_2)p_{\text{env}}(o_3|a_{1:2}, o_{1:2})\delta(s_3 = U(s_2, a_2, o_3))\ldots && (1.4) \end{aligned}$$

p_{env} 是环境对观察的分布（通常未知）。我们定义最优策略为

$$\pi^* = \arg\max_\pi \mathbb{E}_{p_0(s_0)}\left[V_\pi(s_0)\right] \tag{1.5}$$

请注意，选择一个最大化预期奖励总和的策略是**最大预期效用**原则的一个实例。根据我们对环境的假设以及代理的形式，有各种设计或学习最优策略的方法。以下我们将讨论其中的一些选项。

1.1.2 通用模型

一个用于序列决策问题的通用表示（这是在 [Pow22] 中提出的“通用建模框架”的扩展版本）如图 1.2 所示。在此，我们假设了

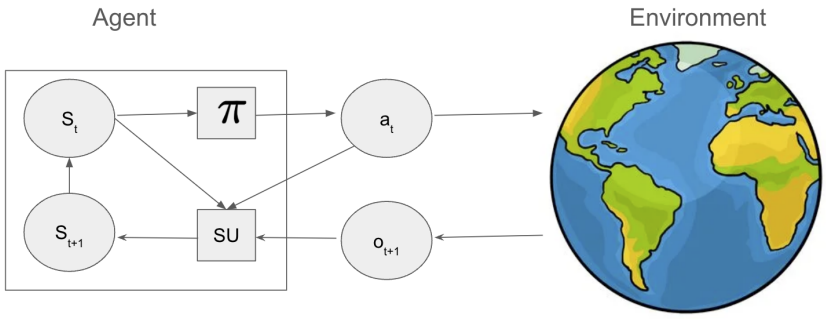


Figure 1.1: A small agent interacting with a big external world.

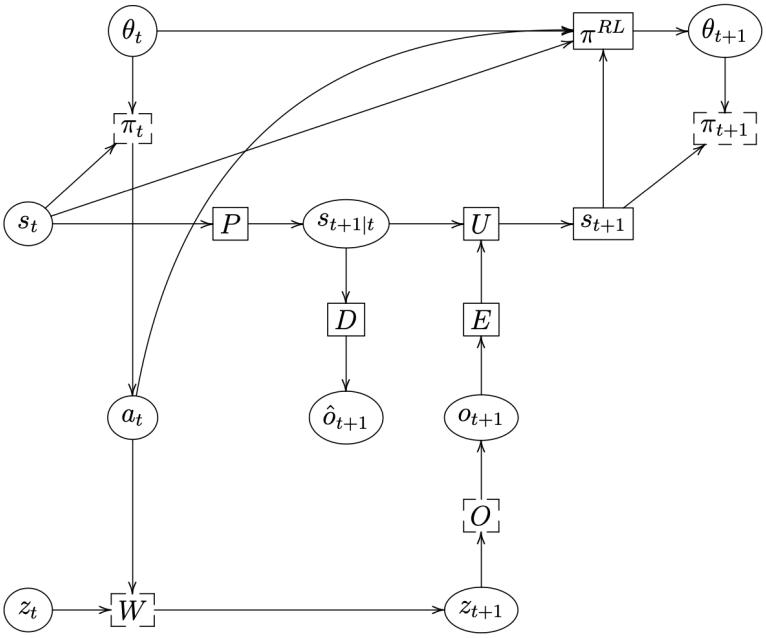


Figure 1.2: Diagram illustrating the interaction of the agent and environment. The agent has internal state s_t, and chooses action a_t based on its policy π_t. It then predicts its next internal states, $s_{t+1|t}$, via the predict function P, and optionally predicts the resulting observation, $\hat{o}_{t+1}$, via the observation decoder D. The environment has (hidden) internal state z_t, which gets updated by the world model W to give the new state $z_{t+1} = W(z_t, a_t)$ in response to the agent's action. The environment also emits an observation o_{t+1} via the observation model O. This gets encoded to e_{t+1} by the agent's observation encoder E, which the agent uses to update its internal state using $s_{t+1} = U(s_t, a_t, e_{t+1})$. The policy is parameterized by θ_t, and these parameters may be updated (at a slower time scale) by the RL policy π^{RL}. Square nodes are functions, circles are variables (either random or deterministic). Dashed square nodes are stochastic functions that take an extra source of randomness (not shown).

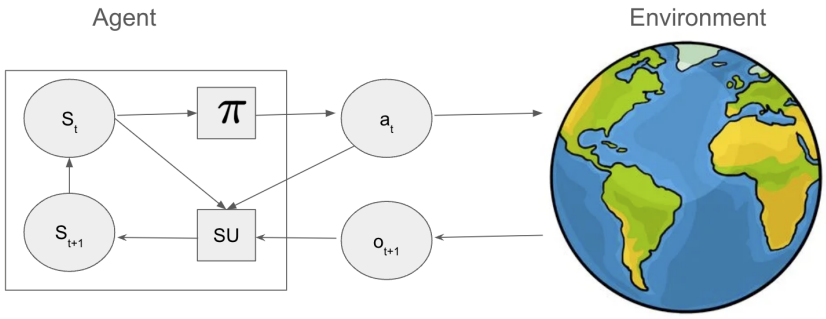


图 1.1：一个小型代理与一个大外部世界交互。

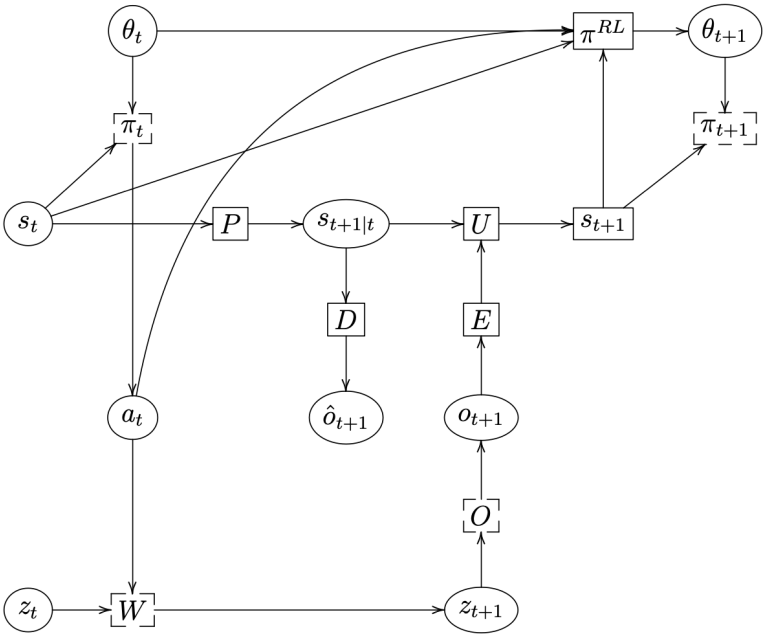


图 1.2：说明智能体与环境交互的图。智能体具有内部状态 s_t，根据其策略 π_t 选择动作 a_t 。然后通过预测函数 P 预测其下一个内部状态 $s_{t+1|t}$，并通过观察解码器 D 可选地预测结果观察 $\hat{o}_{t+1}$。环境具有（隐藏的）内部状态 z_t，该状态通过世界模型 W 更新，以给出新的状态 $z_{t+1} = W\left(z_t, a_t\right)$，作为对智能体动作的响应。环境还通过观察模型 O 发出观察 o_{t+1} 。该观察被智能体的观察编码器 E 编码为 e_{t+1}，智能体使用 $s_{t+1} = U\left(s_t, a_t, e_{t+1}\right)$ 来更新其内部状态。策略由 θ_t 参数化，这些参数可能通过 RL 策略 π^{RL} （在较慢的时间尺度上）更新。方形节点是函数，圆形节点是变量（随机或确定性的）。虚线方形节点是随机函数，它需要一个额外的随机源（未显示）。

environment can be modeled by a controlled **Markov process**[1] with hidden state z_t, which gets updated at each step in response to the agent's action a_t. To allow for non-deterministic dynamics, we write this as $z_{t+1} = W(z_t, a_t, \epsilon_t^z)$, where W is the environment's state transition function (which is usually not known to the agent) and ϵ_t^z is random system noise.[2], The agent does not see the world state z_t, but instead sees a potentially noisy and/or partial observation $o_{t+1} = O(z_{t+1}, \epsilon_{t+1}^o)$ at each step, where ϵ_{t+1}^o is random observation noise. For example, when navigating a maze, the agent may only see what is in front of it, rather than seeing everything in the world all at once; furthermore, even the current view may be corrupted by sensor noise. Any given image, such as one containing a door, could correspond to many different locations in the world (this is called **perceptual aliasing**), each of which may require a different action. Thus the agent needs use these observations to incrementally update its own internal **belief state** about the world, using the state update function $s_{t+1} = SU(s_t, a_t, o_{t+1})$; this represents the agent's beliefs about the underlying world state z_t, as well as the unknown world model W itself (or some proxy thereof). In the simplest setting, the internal s_t can just store all the past observations, $\boldsymbol{h}_t = (\boldsymbol{o}_{1:t}, \boldsymbol{a}_{1:t-1})$, but such non-parametric models can take a lot of time and space to work with, so we will usually consider parametric approximations. The agent can then pass its state to its policy to pick actions, using $a_{t+1} = \pi_t(s_{t+1})$.

We can further elaborate the behavior of the agent by breaking the state-update function into two parts. First the agent predicts its own next state, $s_{t+1|t} = P(s_t, a_t)$, using a **prediction function** P, and then it updates this prediction given the observation using **update function** U, to give $s_{t+1} = U(s_{t+1|t}, o_{t+1})$. Thus the SU function is defined as the composition of the predict and update functions: $s_{t+1} = SU(s_t, a_t, o_{t+1}) = U(P(s_t, a_t), o_{t+1})$. If the observations are high dimensional (e.g., images), the agent may choose to encode its observations into a low-dimensional embedding e_{t+1} using an encoder, $e_{t+1} = E(o_{t+1})$; this can encourage the agent to focus on the relevant parts of the sensory signal. (The state update then becomes $s_{t+1} = U(s_{t+1|t}, e_{t+1})$.) Optionally the agent can also learn to invert this encoder by training a decoder to predict the next observation using $\hat{o}_{t+1} = D(s_{t+1|t})$; this can be a useful training signal, as we will discuss in Chapter 4. Finally, the agent needs to learn the action policy π_t. We parameterize this by $\boldsymbol{\theta}_t$, so $\pi_t(s_t) = \pi(s_t; \boldsymbol{\theta}_t)$. These parameters themselves may need to be learned; we use the notation π^{RL} to denote the RL policy which specifies how to update the policy parameters at each step. See Figure 1.2 for an illustration.

We see that, in general, there are three interacting stochastic processes we need to deal with: the environment's states z_t (which are usually affected by the agents actions); the agent's internal states s_t (which reflect its beliefs about the environment based on the observed data); and the the agent's policy parameters $\boldsymbol{\theta}_t$ (which are updated based on the information stored in the belief state). The reason there are so many RL algorithms is that this framework is very general. In the rest of this manuscript we will study special cases, where we make different assumptions about the environment's state z_t and dynamics, the agent's state s_t and dynamics, the form of the action policy $\pi(s_t|\boldsymbol{\theta}_t)$, and the form of the policy learning method $\boldsymbol{\theta}_{t+1} = \pi^{RL}(\boldsymbol{\theta}_t, s_t, a_t, o_{t+1})$.

1.1.3 Episodic vs continuing tasks

If the agent can potentially interact with the environment forever, we call it a **continuing task**. Alternatively, the agent is in an **episodic task**, if its interaction terminates once the system enters a **terminal state** or **absorbing state**, which is a state which transitions to itself with 0 reward. After entering a terminal state, we may start a new **epsiode** from a new initial world state $z_0 \sim p_0$. (The agent will typically also reinitialize its own internal state s_0.) The episode length is in general random. For example, the amount of time a robot takes to reach its goal may be quite variable, depending on the decisions it makes, and the randomness in the environment. Finally, if the trajectory length T in an episodic task is fixed and known, it is called a **finite horizon problem**.

We define the **return** for a state at time t to be the sum of expected rewards obtained going forwards,

[1] The Markovian assumption is without loss of generality, since we can always condition on the entire past sequence of states by suitably expanding the Markovian state space.

[2] Representing a stochastic function as a deterministic function with some noisy inputs is known as a functional causal model, or structural equation model. This is standard practice in the control theory and causality communities.

环境可以通过一个受控的**马尔可夫过程**[1]进行建模，该过程具有隐藏状态 z_t，在每一步根据代理的动作 a_t 进行更新。为了允许非确定性动态，我们将其写成 $z_{t+1} = W(z_t, a_t, \epsilon_t^z)$，其中 W 是环境的转移函数（通常代理不知道）和 ϵ_t^z 是随机系统噪声。[2]，代理看不到世界状态 z_t，而是看到每个步骤的潜在噪声和 / 或部分观察 $o_{t+1} = O(z_{t+1}, \epsilon_{t+1}^o)$，其中 ϵ_{t+1}^o 是随机观察噪声。例如，在导航迷宫时，代理可能只能看到它面前的东西，而不是一次性看到世界上的所有东西；此外，当前的视图也可能被传感器噪声所破坏。任何给定的图像，如包含门的图像，可能对应于世界上的许多不同位置（这被称为**感知别名**），每个位置可能需要不同的动作。因此，代理需要使用这些观察来逐步更新其关于世界的内部**信念状态**，使用状态更新函数 $s_{t+1} = SU(s_t, a_t, o_{t+1})$；这代表了代理对潜在世界状态的信念 z_t，以及未知的世界模型 W 本身（或其代理）。在最简单的设置中，内部 s_t 可以简单地存储所有过去的观察 $\boldsymbol{h}_t = (\boldsymbol{o}_{1:t}, \boldsymbol{a}_{1:t-1})$，但这样的非参数模型在处理时可能需要大量的时间和空间，因此我们通常会考虑参数近似。然后代理可以将其状态传递给其策略以选择动作，使用 $a_{t+1} = \pi_t(s_{t+1})$。

我们可以通过将状态更新函数分为两部分来进一步阐述代理的行为。首先，代理预测其下一个状态，$s_{t+1|t} = P(s_t, a_t)$，使用一个**预测函数** P，然后它使用**更新函数** U 根据观察更新这个预测，以给出 $s_{t+1} = U(s_{t+1|t}, o_{t+1})$。因此，$SU$ 函数定义为预测和更新函数的组合：$s_{t+1} = SU(s_t, a_t, o_{t+1}) = U(P(s_t, a_t)\, o_{t+1})$。如果观察是高维的（例如，图像），代理可以选择使用编码器将观察编码为低维嵌入 e_{t+1}，$e_{t+1} = E(o_{t+1})$；这可以鼓励代理关注感官信号的相关部分。（状态更新随后变为 $s_{t+1} = U(s_{t+1|t}, e_{t+1})$。）可选地，代理还可以通过训练解码器来预测下一个观察来学习反转这个编码器 $\hat{o}_{t+1} = D(s_{t+1|t})$；这可以是一个有用的训练信号，正如我们将在第 4 章中讨论的那样。最后，代理需要学习动作策略 π_t。我们通过 $\boldsymbol{\theta}_t$ 来参数化它，所以 $\pi_t(s_t) = \pi(s_t; \boldsymbol{\theta}_t)$。这些参数本身可能需要学习；我们使用 π^{RL} 表示 RL 策略，该策略指定了如何在每个步骤更新策略参数。参见图 1.2 以了解说明。

我们注意到，通常有三个相互作用的随机过程需要我们处理：环境的状 z_t（通常受代理的行动影响）；代理的内部状 s_t（基于观察到的数据反映其对环境的信念）；以及代理的策略参数 $\boldsymbol{\theta}_t$（基于信念状态中存储的信息进行更新）。之所以有这么多强化学习算法，是因为这个框架非常通用。在本手稿的其余部分，我们将研究特殊情况，其中我们对环境的状 z_t 和动力学、代理的状 s_t 和动力学、行动策略的形式 π（$s_t|\boldsymbol{\theta}_t$）、以及策略学习方法的形式 $\boldsymbol{\theta}_{t+1} = \pi^{RL}$（$\boldsymbol{\theta}_t$、$s_t$、$a_t$、$o_{t+1}$）做出不同的假设。

1.1.3 一次性任务与持续任务

如果智能体可以永久地与环境交互，我们称其为**持续任务**。或者，如果智能体的交互在系统进入**终端状态或吸收状态后终止，我们称其为**一次性任务，这两种状态都是自身过渡且奖励为 0 的状态。进入终端状态后，我们可以从新的初始世界状态 $z_0 \sim p_0$ 开始一个新的**一次性任务**。（智能体通常也会重新初始化其内部状态 s_0。）一次性任务的长度通常是随机的。例如，机器人到达目标所需的时间可能相当多变，这取决于它的决策以及环境中的随机性。最后，如果一次性任务中的轨迹长度 T 是固定且已知的，则称为**有限时间问题**。

我们定义在时间 t 的状态的**返回**为向前获得期望奖励的总和。

[1]马尔可夫假设在一般性上是不失真的，因为我们总能通过适当地扩展马尔可夫状态空间来对整个状态序列进行条件化。[2] 将随机函数表示为具有一些噪声输入的确定性函数，这被称为功能因果模型或结构方程模型。这在控制理论和因果性社区中是一种标准做法。

where each reward is multiplied by a **discount factor** $\gamma \in [0, 1]$:

$$G_t \triangleq r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \cdots + \gamma^{T-t-1} r_{T-1} \tag{1.6}$$

$$= \sum_{k=0}^{T-t-1} \gamma^k r_{t+k} = \sum_{j=t}^{T-1} \gamma^{j-t} r_j \tag{1.7}$$

where $r_t = R(s_t, a_t)$ is the reward, and G_t is the **reward-to-go**. For episodic tasks that terminate at time T, we define $G_t = 0$ for $t \geq T$. Clearly, the return satisfies the following recursive relationship:

$$G_t = r_t + \gamma(r_{t+1} + \gamma r_{t+2} + \cdots) = r_t + \gamma G_{t+1} \tag{1.8}$$

Furthermore, we define the value function to be the expected reward-to-go:

$$V_\pi(s_t) = \mathbb{E}\left[G_t | \pi\right] \tag{1.9}$$

The discount factor γ plays two roles. First, it ensures the return is finite even if $T = \infty$ (i.e., infinite horizon), provided we use $\gamma < 1$ and the rewards r_t are bounded. Second, it puts more weight on short-term rewards, which generally has the effect of encouraging the agent to achieve its goals more quickly. (For example, if $\gamma = 0.99$, then an agent that reaches a terminal reward of 1.0 in 15 steps will receive an expected discounted reward of $0.99^{15} = 0.86$, whereas if it takes 17 steps it will only get $0.99^{17} = 0.84$.) However, if γ is too small, the agent will become too greedy. In the extreme case where $\gamma = 0$, the agent is completely **myopic**, and only tries to maximize its immediate reward. In general, the discount factor reflects the assumption that there is a probability of $1 - \gamma$ that the interaction will end at the next step. For finite horizon problems, where T is known, we can set $\gamma = 1$, since we know the life time of the agent a priori.[3]

1.1.4 Regret

So far we have been discussing maximizing the reward. However, the upper bound on this is usually unknown, so it can be hard to know how well a given agent is doing. An alternative approach is to work in terms of the **regret**, which is defined as the difference between the expected reward under the agent's policy and the oracle policy π_*, which knows the true MDP. Specifically, let π_t be the agent's policy at time t. Then the **per-step regret** at t is defined as

$$l_t \triangleq \mathbb{E}_{s_{1:t}}\left[R(s_t, \pi_*(s_t)) - \mathbb{E}_{\pi(a_t|s_t)}\left[R(s_t, a_t)\right]\right] \tag{1.10}$$

Here the expectation is with respect to randomness in choosing actions using the policy π, as well as earlier states, actions and rewards, as well as other potential sources of randomness.

If we only care about the final performance of the agent, as in most optimization problems, it is enough to look at the **simple regret** at the last step, namely l_T. Optimizing simple regret results in a problem known as **pure exploration** [BMS11], where the agent needs to interact with the environment to learn the underlying MDP; at the end, it can then solve for the resulting policy using planning methods (see Section 2.2). However, in RL, it is more common to focus on the **cumulative regret**, also called the **total regret** or just the **regret**, which is defined as

$$L_T \triangleq \mathbb{E}\left[\sum_{t=1}^{T} l_t\right] \tag{1.11}$$

Thus the agent will accumulate reward (and regret) while it learns a model and policy. This is called **earning while learning**, and requires performing exploratory actions, to learn the model (and hence optimize long-term reward), while also performing actions that maximize the reward at each step. This requires solving the exploration-exploitation tradeoff, as we discussed in Section 1.4.

[3] We may also use $\gamma = 1$ for continuing tasks, targeting the (undiscounted) average reward criterion [Put94].

每个奖励都乘以一个**折扣因子** $\gamma \in [0,1]$：

$$G_t \triangleq r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \cdots + \gamma^{T-t-1} r_{T-1} \tag{1.6}$$

$$= \sum_{k=0}^{T-t-1} \gamma^k r_{t+k} = \sum_{j=t}^{T-1} \gamma^{j-t} r_j \tag{1.7}$$

$r_t = R(s_t, a_t)$ 是奖励，G_t 是**奖励到目标**。对于在时间 T 终止的周期性任务，我们定义 $G_t = 0$ 用于 $t \geq T$。显然，回报满足以下递归关系：

$$G_t = r_t + \gamma(r_{t+1} + \gamma r_{t+2} + \cdots) = r_t + \gamma G_{t+1} \tag{1.8}$$

此外，我们定义值函数为预期奖励到目标：

$$V_\pi(s_t) = \mathbb{E}\left[G_t | \pi\right] \tag{1.9}$$

折扣因子 γ 扮演两个角色。首先，它确保回报是有限的，即使 $T = \infty$ （即，无限视野），只要我们使用 $\gamma < 1$，并且奖励 r_t 是有界的。其次，它更重视短期奖励，这通常具有鼓励代理更快实现其目标的效果。（例如，如果 $\gamma = 0.99$，那么一个在 15 步内达到终端奖励 1.0 的代理将获得期望的折现奖励 $0.99^{15} = 0.86$，而如果它需要 17 步，它将只能得到 $0.99^{17} = 0.84$。）然而，如果 γ 太小，代理会变得过于贪婪。在极端情况下，当 $\gamma = 0$，代理会变得完全**短视**，只试图最大化其即时奖励。一般来说，折扣因子反映了存在一个概率 $1 - \gamma$，即交互将在下一步结束的假设。对于有限视野问题，其中 T 是已知的，我们可以设置 $\gamma = 1$，因为我们事先知道代理的寿命。[3]

1.1.4 后悔

到目前为止，我们一直在讨论最大化奖励。然而，这个上限通常未知，因此很难知道一个特定代理的表现如何。一种替代方法是按照**后悔**来工作，它被定义为在代理策略下的期望奖励与知道真实 MDP 的先验策略 π_* 之间的差异。具体来说，让 π_t 为时间 t 的代理策略。那么在**每一步的后悔**在 t 被定义为

$$l_t \triangleq \mathbb{E}_{s_{1:t}}\left[R(s_t, \pi_*(s_t)) - \mathbb{E}_{\pi(a_t|s_t)}\left[R(s_t, a_t)\right]\right] \tag{1.10}$$

这里期望的是关于使用策略 π 选择动作的随机性，以及早期状态、动作和奖励，以及其他可能的随机来源。

如果我们只关心代理的最终性能，就像在大多数优化问题中一样，我们只需要关注最后一步的简单遗憾，即 l_T。优化简单遗憾导致了一个被称为**纯探索** [BMS11]，的问题，其中代理需要与环境交互以学习潜在的 MDP；最后，它可以使用规划方法（见第 2.2 节）来解决产生的策略。然而，在强化学习中，更常见的是关注**累积遗憾**，也称为**总遗憾**或简称为**遗憾**，它被定义为

$$L_T \triangleq \mathbb{E}\left[\sum_{t=1}^{T} l_t\right] \tag{1.11}$$

因此，在代理学习模型和政策的过程中，它会积累奖励（和遗憾）。这被称为**边学习边学习**，需要执行探索性动作来学习模型（从而优化长期奖励），同时也要执行每步最大化奖励的动作。这需要解决探索 - 利用权衡，正如我们在第 1.4 节中讨论的那样。

[3] We may also use $\gamma = 1$ 继续任务，针对（未折扣）平均奖励 criterion [Put94].

1.1.5 Further reading

In later chapters, we will describe methods for learning the best policy to maximize $V_\pi(s_0) = \mathbb{E}[G_0|s_0, \pi])$. More details on RL can be found in textbooks such as [Sze10; SB18; Aga+22a; Pla22; ID19; RJ22; Li23; MMT24], and reviews such as [Aru+17; FL+18; Li18; Wen18a]. For details on how RL relates to **control theory**, see e.g., [Son98; Rec19; Ber19; Mey22], and for connections to operations research, see [Pow22].

1.2 Canonical examples

In this section, we describe different forms of model for the environment and the agent that have been studied in the literature.

1.2.1 Partially observed MDPs

The model shown in Figure 1.2 is called a **partially observable Markov decision process** or **POMDP** (pronounced "pom-dee-pee") [KLC98]. Typically the environment's dynamics model is represented by a stochastic transition function, rather than a deterministic function with noise as an input. We can derive this transition function as follows:

$$p(z_{t+1}|z_t, a_t) = \mathbb{E}_{\epsilon_t^z}\left[\mathbb{I}\left(z_{t+1} = W(z_t, a_t, \epsilon_t^z)\right)\right] \tag{1.12}$$

Similarly the stochastic observation function is given by

$$p(o_{t+1}|z_{t+1}) = \mathbb{E}_{\epsilon_{t+1}^o}\left[\mathbb{I}\left(o_{t+1} = O(z_{t+1}, \epsilon_{t+1}^o)\right)\right] \tag{1.13}$$

Note that we can combine these two distributions to derive the joint world model $p_{WO}(z_{t+1}, o_{t+1}|z_t, a_t)$. Also, we can use these distributions to derive the environment's non-Markovian observation distribution, $p_{\text{env}}(o_{t+1}|o_{1:t}, a_{1:t})$, used in Equation (1.4), as follows:

$$p_{\text{env}}(o_{t+1}|o_{1:t}, a_{1:t}) = \sum_{z_{t+1}} p(o_{t+1}|z_{t+1}) p(z_{t+1}|a_{1:t}) \tag{1.14}$$

$$p(z_{t+1}|a_{1:t}) = \sum_{z_1} \cdots \sum_{z_t} p(z_1|a_1) p(z_2|z_1, a_1) \ldots p(z_{t+1}|z_t, a_t) \tag{1.15}$$

If the world model (both $p(o|z)$ and $p(z'|z, a)$) is known, then we can — in principle — solve for the optimal policy. The method requires that the agent's internal state correspond to the **belief state** $s_t = \boldsymbol{b}_t = p(z_t|\boldsymbol{h}_t)$, where $\boldsymbol{h}_t = (o_{1:t}, a_{1:t-1})$ is the observation history. The belief state can be updated recursively using Bayes rule. See Section 1.2.5 for details. The belief state forms a sufficient statistic for the optimal policy. Unfortunately, computing the belief state and the resulting optimal policy is wildly intractable [PT87; KLC98]. We discuss some approximate methods in Section 1.3.4.

1.2.2 Markov decision process (MDPs)

A **Markov decision process** [Put94] is a special case of a POMDP in which the environment states are observed, so $z_t = o_t = s_t$.[4] We usually define an MDP in terms of the state transition matrix induced by the world model:

$$p_S(s_{t+1}|s_t, a_t) = \mathbb{E}_{\epsilon_t^s}\left[\mathbb{I}\left(s_{t+1} = W(s_t, a_t, \epsilon_t^s)\right)\right] \tag{1.16}$$

[4] The field of control theory uses slightly different terminology and notation. In particular, the environment is called the **plant**, and the agent is called the **controller**. States are denoted by $\boldsymbol{x}_t \in \mathcal{X} \subseteq \mathbb{R}^D$, actions are denoted by $\boldsymbol{u}_t \in \mathcal{U} \subseteq \mathbb{R}^K$, and rewards are replaced by costs $c_t \in \mathbb{R}$.

1.1.5 进一步阅读

在后续章节中，我们将描述学习最佳策略以最大化 $V_\pi(s_0) = \mathbb{E}\left[G_0|s_0, \pi\right]$) 的方法。关于强化学习的更多细节可以在以下教材中找到，如 [Sze10； SB18； Aga+22a； Pla22； ID19； RJ22； Li23； MMT24]，以及以下评论，如 [Aru+17； FL+18； Li18； Wen18a]。关于强化学习与控制理论的关系的详细信息，请参阅例如 [Son98； Rec19； Ber19； Mey22]，，以及与运筹学的联系，请参阅 [Pow22]。

1.2 正规示例

在这一节中，我们描述了文献中研究过的环境和代理的不同形式。

1.2.1 部分可观测的马尔可夫决策过程

图 1.2 中所示模型被称为**部分可观测马尔可夫决策过程**或 **POMDP** （发音为 “pom-dee-pee”） [KLC98]。通常，环境的动力学模型由一个随机转移函数表示，而不是一个具有噪声输入的确定性函数。我们可以推导出这个转移函数如下：

$$p(z_{t+1}|z_t, a_t) = \mathbb{E}_{\epsilon_t^z}\left[\mathbb{I}\left(z_{t+1} = W(z_t, a_t, \epsilon_t^z)\right)\right] \tag{1.12}$$

类似地，随机观测函数由以下给出

$$p(o_{t+1}|z_{t+1}) = \mathbb{E}_{\epsilon_{t+1}^o}\left[\mathbb{I}\left(o_{t+1} = O(z_{t+1}, \epsilon_{t+1}^o)\right)\right] \tag{1.13}$$

请注意，我们可以将这两个分布结合起来，推导出联合世界模型 $p_{WO}(z_{t+1}, o_{t+1}|z_t, a_t)$。此外，我们还可以使用这些分布来推导环境的非马尔可夫观察分布，$p_{\text{env}}(o_{t+1}|o_{1:t},a_{1:t})$，在方程（ 1.4 ）中使用，如下：

$$p_{\text{env}}(o_{t+1}|o_{1:t}, a_{1:t}) = \sum_{z_{t+1}} p(o_{t+1}|z_{t+1})p(z_{t+1}|a_{1:t}) \tag{1.14}$$

$$p(z_{t+1}|a_{1:t}) = \sum_{z_1} \cdots \sum_{z_t} p(z_1|a_1)p(z_2|z_1, a_1) \ldots p(z_{t+1}|z_t, a_t) \tag{1.15}$$

如果已知世界模型（包括 $p(o|z)$ 和 $p(z'|z,a)$ ），那么原则上我们可以求解最优策略。该方法要求代理的内部状态与**信念状态** $s_t = \boldsymbol{b}_t = p(z_t|\boldsymbol{h}_t)$ 相对应，其中 $\boldsymbol{h}_t = (o_{1:t},a_{1:t-1})$ 是观察历史。信念状态可以通过贝叶斯规则递归更新。有关详细信息，请参阅第 1.2.5 节。信念状态构成了最优策略的充分统计量。不幸的是，计算信念状态和由此产生的最优策略是极其难以处理的 [PT87； KLC98]。我们将在第 1.3.4 节中讨论一些近似方法。

1.2.2 马尔可夫决策过程（ MDPs ）

A **马尔可夫决策过程** [Put94] 是 POMDP 的一种特殊情况，其中环境状态是可以观察到的，因此 $z_t = o_t = s_t$.[4] 我们通常根据世界模型诱导的状态转移矩阵来定义 MDP：

$$p_S(s_{t+1}|s_t, a_t) = \mathbb{E}_{\epsilon_t^s}\left[\mathbb{I}\left(s_{t+1} = W(s_t, a_t, \epsilon_t^s)\right)\right] \tag{1.16}$$

控制理论领域使用略微不同的术语和符号。特别是，环境被称为**植物**，而智能体被称为**控制器**。状态用 $\boldsymbol{x}_t \in \mathcal{X} \subseteq \mathbb{R}^D$表示，动作用 $\boldsymbol{u}_t \in \mathcal{U} \subseteq \mathbb{R}^K$表示，奖励用成本 $c_t \in \mathbb{R}$ 代替。

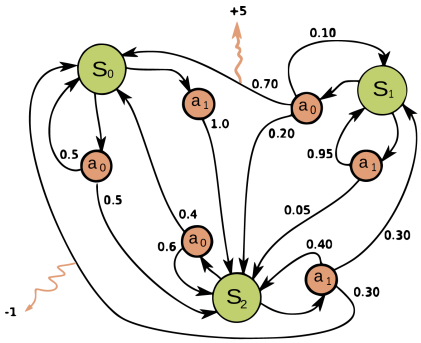


Figure 1.3: Illustration of an MDP as a finite state machine (FSM). The MDP has three discrete states (green cirlces), two discrete actions (orange circles), and two non-zero rewards (orange arrows). The numbers on the black edges represent state transition probabilities, e.g., $p(s' = s_0|a = a_0, s' = s_0) = 0.7$; most state transitions are impossible (probability 0), so the graph is sparse. The numbers on the yellow wiggly edges represent expected rewards, e.g., $R(s = s_1, a = a_0, s' = s_0) = +5$; state transitions with zero reward are not annotated. From https://en.wikipedia.org/wiki/Markov_decision_process. Used with kind permission of Wikipedia author waldoalvarez.

In lieu of an observation model, we assume the environment (as opposed to the agent) sends out a reward signal, sampled from $p_R(r_t|s_t, a_t, s_{t+1})$. The expected reward is then given by

$$R(s_t, a_t, s_{t+1}) = \sum_r r \;\; p_R(r|s_t, a_t, s_{t+1}) \tag{1.17}$$

$$R(s_t, a_t) = \sum_{s_{t+1}} p_S(s_{t+1}|s_t, a_t) R(s_t, a_t, s_{t+1}) \tag{1.18}$$

Given a stochastic policy $\pi(a_t|s_t)$, the agent can interact with the environment over many steps. Each step is called a **transition**, and consists of the tuple (s_t, a_t, r_t, s_{t+1}), where $a_t \sim \pi(\cdot|s_t)$, $s_{t+1} \sim p_S(s_t, a_t)$, and $r_t \sim p_R(s_t, a_t, s_{t+1})$. Hence, under policy π, the probability of generating a **trajectory** length T, $\boldsymbol{\tau} = (s_0, a_0, r_0, s_1, a_1, r_1, s_2, \ldots, s_T)$, can be written explicitly as

$$p(\boldsymbol{\tau}) = p_0(s_0) \prod_{t=0}^{T-1} \pi(a_t|s_t) p_S(s_{t+1}|s_t, a_t) p_R(r_t|s_t, a_t, s_{t+1}) \tag{1.19}$$

In general, the state and action sets of an MDP can be discrete or continuous. When both sets are finite, we can represent these functions as lookup tables; this is known as a **tabular representation**. In this case, we can represent the MDP as a **finite state machine**, which is a graph where nodes correspond to states, and edges correspond to actions and the resulting rewards and next states. Figure 1.3 gives a simple example of an MDP with 3 states and 2 actions.

If we know the world model p_S and p_R, and if the state and action space is tabular, then we can solve for the optimal policy using dynamic programming techniques, as we discuss in Section 2.2. However, typically the world model is unknown, and the states and actions may need complex nonlinear models to represent their transitions. In such cases, we will have to use RL methods to learn a good policy.

1.2.3 Contextual MDPs

A **Contextual MDP** [HDCM15] is an MDP where the dynamics and rewards of the environment depend on a hidden static parameter referred to as the context. (This is different to a contextual bandit, discussed in Section 1.2.4, where the context is observed at each step.) A simple example of a contextual MDP is a video game, where each level of the game is **procedurally generated**, that is, it is randomly generated each time the agent starts a new episode. Thus the agent must solve a sequence of related MDPs, which are

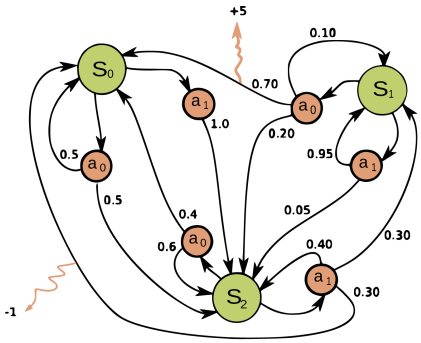


图 1.3：将 MDP 作为有限状态机（FSM）的示意图。MDP 有三个离散状态（绿色圆圈），两个离散动作（橙色圆圈）和两个非零奖励（橙色箭头）。黑色边上的数字表示状态转移概率，例如 $p(s' = s_0|a = a_0, s' = s_0) = 0.7$；大多数状态转移是不可能的（概率为 0），因此图是稀疏的。黄色波浪线上的数字表示期望奖励，例如 $R(s = s_1, a = a_0, s' = s_0) = +5$；没有标注零奖励的状态转移。来自 https://zh.wikipedia.org/wiki/ 马尔可夫决策过程 。经维基百科作者 waldoalvarez 许可使用。

代替观察模型，我们假设环境（而不是智能体）发送一个奖励信号，该信号从 $p_R(r_t|s_t, a_t, s_{t+1})$ 中采样。预期的奖励由以下给出。

$$R(s_t, a_t, s_{t+1}) = \sum_r r \;\; p_R(r|s_t, a_t, s_{t+1}) \tag{1.17}$$

$$R(s_t, a_t) = \sum_{s_{t+1}} p_S(s_{t+1}|s_t, a_t) R(s_t, a_t, s_{t+1}) \tag{1.18}$$

给定一个随机策略 $\pi(a_t|s_t)$，智能体可以在多个步骤与环境交互。每个步骤称为一个 **转换**，由元组 (s_t, a_t, r_t, s_{t+1}) 组成，其中 $a_t \sim \pi(\cdot|s_t)$，$s_{t+1} \sim p_S(s_t, a_t)$，和 $r_t \sim p_R(s_t, a_t, s_{t+1})$。因此，在策略 π 下，生成一个 **轨迹** 长度 T，$\boldsymbol{\tau} = (s_0, a_0, r_0, s_1, a_1, r_1, s_2, \ldots, s_T)$ 的概率可以明确表示为

$$p(\boldsymbol{\tau}) = p_0(s_0) \prod_{t=0}^{T-1} \pi(a_t|s_t) p_S(s_{t+1}|s_t, a_t) p_R(r_t|s_t, a_t, s_{t+1}) \tag{1.19}$$

通常，MDP 的状态集和动作集可以是离散的或连续的。当这两个集合都是有限的时，我们可以将这些函数表示为查找表；这被称为表格表示。在这种情况下，我们可以将 MDP 表示为一个有限状态机，它是一个图，其中节点对应于状态，边对应于动作以及相应的奖励和下一个状态。图 1.3 给出了一个具有 3 个状态和 2 个动作的 MDP 的简单示例。

如果我们知道世界模型 p_S 和 p_R，并且状态和动作空间是表格化的，那么我们可以使用动态规划技术求解最优策略，正如我们在第 2.2 节所讨论的。然而，通常世界模型是未知的，状态和动作可能需要复杂的非线性模型来表示它们的转换。在这种情况下，我们必须使用强化学习方法来学习一个好的策略。

1.2.3 上下文马尔可夫决策过程

A **上下文马尔可夫决策过程** [HDCM15] 是一种马尔可夫决策过程，其中环境的动态和奖励取决于一个称为上下文的隐藏静态参数。（这与第 1.2.4 节中讨论的上下文赌博机不同，在那里上下文在每个步骤都被观察到。）上下文马尔可夫决策过程的一个简单例子是视频游戏，其中游戏的每个级别都是**程序生成的**，也就是说，每次智能体开始新的一集时都会随机生成。因此，智能体必须解决一系列相关的马尔可夫决策过程，这些过程是

drawn from a common distribution. This requires the agent to **generalize** across multiple MDPs, rather than overfitting to a specific environment [Cob+19; Kir+21; Tom+22]. (This form of generalization is different from generalization within an MDP, which requires generalizing across states, rather than across environments; both are important.)

A contextual MDP is a special kind of POMDP where the hidden variable corresponds to the unknown parameters of the model. In [Gho+21], they call this an **epistemic POMDP**, which is closely related to the concept of belief state MDP which we discuss in Section 1.2.5.

1.2.4 Contextual bandits

A **contextual bandit** is a special case of a POMDP where the world state transition function is independent of the action of the agent and the previous state, i.e., $p(z_t|z_{t-1}, a_t) = p(z_t)$. In this case, we call the world states "contexts"; these are observable by the agent, i.e., $o_t = z_t$. Since the world state distribution is independent of the agents actions, the agent has no effect on the external environment. However, its actions do affect the rewards that it receives. Thus the agent's internal belief state — about the underlying reward function $R(o, a)$ — does change over time, as the agent learns a model of the world (see Section 1.2.5).

A special case of a contextual bandit is a regular bandit, in which there is no context, or equivalently, s_t is some fixed constant that never changes. When there are a finite number of possible actions, $\mathcal{A} = \{a_1, \ldots, a_K\}$, this is called a **multi-armed bandit**.[5] In this case the reward model has the form $R(a) = f(\boldsymbol{w}_a)$, where $\boldsymbol{w}_a$ are the parameters for arm a.

Contextual bandits have many applications. For example, consider an **online advertising system**. In this case, the state s_t represents features of the web page that the user is currently looking at, and the action a_t represents the identity of the ad which the system chooses to show. Since the relevance of the ad depends on the page, the reward function has the form $R(s_t, a_t)$, and hence the problem is contextual. The goal is to maximize the expected reward, which is equivalent to the expected number of times people click on ads; this is known as the **click through rate** or **CTR**. (See e.g., [Gra+10; Li+10; McM+13; Aga+14; Du+21; YZ22] for more information about this application.) Another application of contextual bandits arises in **clinical trials** [VBW15]. In this case, the state s_t are features of the current patient we are treating, and the action a_t is the treatment the doctor chooses to give them (e.g., a new drug or a **placebo**).

For more details on bandits, see e.g., [LS19; Sli19].

1.2.5 Belief state MDPs

In this section, we describe a kind of MDP where the state represents a probability distribution, known as a **belief state** or **information state**, which is updated by the agent ("in its head") as it receives information from the environment.[6] More precisely, consider a contextual bandit problem, where the agent approximates the unknown reward by a function $R(o, a) = f(o, a; \boldsymbol{w})$. Let us denote the posterior over the unknown parameters by $\boldsymbol{b}_t = p(\boldsymbol{w}|\boldsymbol{h}_t)$, where $\boldsymbol{h}_t = \{o_{1:t}, a_{1:t}, r_{1:t}\}$ is the history of past observations, actions and rewards. This belief state can be updated deterministically using Bayes' rule; we denote this operation by $\boldsymbol{b}_{t+1} = \text{BayesRule}(\boldsymbol{b}_t, o_{t+1}, a_{t+1}, r_{t+1})$. (This corresponds to the state update SU defined earlier.) Using this, we can define the following **belief state MDP**, with deterministic dynamics given by

$$p(\boldsymbol{b}_{t+1}|\boldsymbol{b}_t, o_{t+1}, a_{t+1}, r_{t+1}) = \mathbb{I}\left(\boldsymbol{b}_{t+1} = \text{BayesRule}(\boldsymbol{b}_t, o_{t+1}, a_{t+1}, r_{t+1})\right) \tag{1.20}$$

and reward function given by

$$p(r_t|o_t, a_t, \boldsymbol{b}_t) = \int p_R(r_t|o_t, a_t; \boldsymbol{w}) p(\boldsymbol{w}|\boldsymbol{b}_t) d\boldsymbol{w} \tag{1.21}$$

[5] The terminology arises by analogy to a slot machine (sometimes called a "bandit") in a casino. If there are K slot machines, each with different rewards (payout rates), then the agent (player) must explore the different machines until they have discovered which one is best, and can then stick to exploiting it.

[6] Technically speaking, this is a POMDP, where we assume the states are observed, and the parameters are the unknown hidden random variables. This is in contrast to Section 1.2.1, where the states were not observed, and the parameters were assumed to be known.

从公共分布中抽取。这要求代理**泛化**跨越多个 MDP，而不是过度拟合特定环境 [Cob+19； Kir+21； Tom+22]。（这种泛化形式与 MDP 内的泛化不同，MDP 内的泛化要求跨越状态泛化，而不是跨越环境泛化；两者都很重要。）

上下文 MDP 是一种特殊的 POMDP，其中隐藏变量对应于模型中的未知参数。在 [Gho+21] 中，他们称这种为**认知 POMDP**，这与我们在第 1.2.5 节中讨论的信念状态 MDP 概念密切相关。

1.2.4 上下文 bandits

A **上下文型强盗**是 POMDP 的一种特殊情况，其中世界状态转换函数与代理的动作和先前状态无关，即 $p(z_t|z_{t-1},a_t) = p(z_t)$。在这种情况下，我们称世界状态为“上下文”；这些是代理可以观察到的，即 $o_t = z_t$。由于世界状态分布与代理的动作无关，代理对外部环境没有影响。然而，其动作会影响它收到的奖励。因此，代理的内部信念状态 —— 关于基本奖励函数 $R(o,a)$ —— 会随时间变化，因为代理学习了一个世界模型（见第 1.2.5 节）。

一个上下文老虎机的一个特殊情况是常规老虎机，其中没有上下文，或者说，s_t 是某个固定常数，永远不会改变。当存在有限数量的可能动作 $\mathcal{A} = \{a_1, \ldots, a_K\}$ 时，这被称为**多臂老虎机**。在这种情况下，奖励模型的形式为 $R(a) = f(\boldsymbol{w}_a)$，其中 $\boldsymbol{w}_a$ 是多臂 a 的参数。

上下文赌博机有许多应用。例如，考虑一个**在线广告系统**。在这种情况下，状态 s_t 代表用户当前正在查看的网页的特征，而动作 a_t 代表系统选择展示的广告的标识。由于广告的相关性取决于页面，奖励函数的形式为 $R(s_t,a_t)$，因此问题是上下文的。目标是最大化期望奖励，这相当于人们点击广告的期望次数；这被称为**点击率**或 **CTR**。（例如，参见 [Gra+10； Li+10； McM+13； Aga+14； Du+21； YZ22] 了解更多关于此应用的信息。）上下文赌博机的另一个应用出现在**临床试验** [VBW15] 中。在这种情况下，状态 s_t 是我们正在治疗的当前患者的特征，而动作 a_t 是医生选择给予他们的治疗（例如，一种新药或**安慰剂**）。

有关强盗的更多详细信息，请参阅例如 [LS19； Sli19]。

1.2.5 信念状态 MDPs

在这一节中，我们描述了一种 MDP，其中状态表示一个概率分布，称为信念状态或信息状态，它由代理（“在其头脑中”）在从环境中接收信息时更新。更确切地说，考虑一个上下文赌博机问题，其中代理通过函数 $R(o,a) = f(o,a;\boldsymbol{w})$ 来近似未知的奖励。让我们用 $\boldsymbol{b}_t = p(\boldsymbol{w}|\boldsymbol{h}_t)$ 表示未知参数的后验分布，其中 $\boldsymbol{h}_t = \{o_{1:t}, a_{1:t}, r_{1:t}\}$ 是过去观察、动作和奖励的历史。这个信念状态可以通过贝叶斯规则确定性地更新；我们用 $\boldsymbol{b}_{t+1} = \text{BayesRule}(\boldsymbol{b}_t,o_{t+1},a_{t+1},r_{t+1})$ 表示这个操作。（这对应于之前定义的状态更新 SU。）使用这个，我们可以定义以下信念状态 MDP，其确定性动力学由以下给出。

$$p(\boldsymbol{b}_{t+1}|\boldsymbol{b}_t, o_{t+1}, a_{t+1}, r_{t+1}) = \mathbb{I}\left(\boldsymbol{b}_{t+1} = \text{BayesRule}(\boldsymbol{b}_t, o_{t+1}, a_{t+1}, r_{t+1})\right) \tag{1.20}$$

并且由奖励函数给出

$$p(r_t|o_t, a_t, \boldsymbol{b}_t) = \int p_R(r_t|o_t, a_t; \boldsymbol{w}) p(\boldsymbol{w}|\boldsymbol{b}_t) d\boldsymbol{w} \tag{1.21}$$

[5]这种术语是通过类比赌场的老虎机（有时称为“海盗”）产生的。如果有K 台老虎机，每台有不同的奖励（支付率），那么代理人（玩家）必须探索不同的机器，直到他们发现哪一台最好，然后就可以坚持利用它了。[6] 从技术上来说，这是一个 POMDP，我们假设状态是可观察的，参数是未知的隐藏随机变量。这与第 1.2.1 节形成对比，在那里状态没有被观察到，参数被假定为已知。

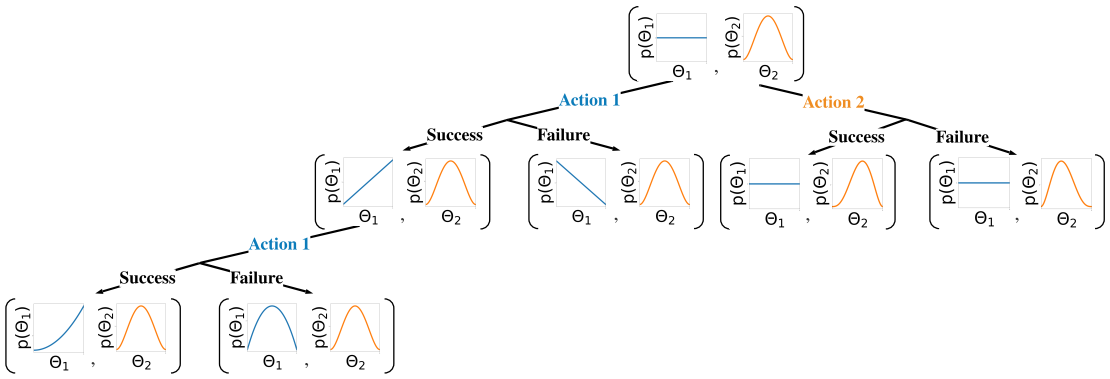


Figure 1.4: Illustration of sequential belief updating for a two-armed beta-Bernoulli bandit. The prior for the reward for action 1 is the (blue) uniform distribution $\text{Beta}(1,1)$*; the prior for the reward for action 2 is the (orange) unimodal distribution* $\text{Beta}(2,2)$*. We update the parameters of the belief state based on the chosen action, and based on whether the observed reward is success (1) or failure (0).*

If we can solve this (PO)MDP, we have the optimal solution to the exploration-exploitation problem.

As a simple example, consider a context-free **Bernoulli bandit**, where $p_R(r|a) = \text{Ber}(r|\mu_a)$, and $\mu_a = p_R(r=1|a) = R(a)$ is the expected reward for taking action a. The only unknown parameters are $\boldsymbol{w} = \mu_{1:A}$. Suppose we use a factored beta prior

$$p_0(\boldsymbol{w}) = \prod_a \text{Beta}(\mu_a|\alpha_0^a, \beta_0^a) \tag{1.22}$$

where $\boldsymbol{w} = (\mu_1, \ldots, \mu_K)$. We can compute the posterior in closed form to get

$$p(\boldsymbol{w}|\mathcal{D}_t) = \prod_a \text{Beta}(\mu_a| \underbrace{\alpha_0^a + N_t^0(a)}_{\alpha_t^a}, \underbrace{\beta_0^a + N_t^1(a)}_{\beta_t^a}) \tag{1.23}$$

where

$$N_t^r(a) = \sum_{i=1}^{t-1} \mathbb{I}\left(a_i = a, r_i = r\right) \tag{1.24}$$

This is illustrated in Figure 1.4 for a two-armed Bernoulli bandit. We can use a similar method for a **Gaussian bandit**, where $p_R(r|a) = \mathcal{N}(r|\mu_a, \sigma_a^2)$.

In the case of contextual bandits, the problem is conceptually the same, but becomes more complicated computationally. If we assume a **linear regression bandit**, $p_R(r|s,a;\boldsymbol{w}) = \mathcal{N}(r|\boldsymbol{\phi}(s,a)^\mathsf{T}\boldsymbol{w}, \sigma^2)$, we can use Bayesian linear regression to compute $p(\boldsymbol{w}|\mathcal{D}_t)$ exactly in closed form. If we assume a **logistic regression bandit**, $p_R(r|s,a;\boldsymbol{w}) = \text{Ber}(r|\sigma(\boldsymbol{\phi}(s,a)^\mathsf{T}\boldsymbol{w}))$, we have to use approximate methods for approximate Bayesian logistic regression to compute $p(\boldsymbol{w}|\mathcal{D}_t)$. If we have a **neural bandit** of the form $p_R(r|s,a;\boldsymbol{w}) = \mathcal{N}(r|f(s,a;\boldsymbol{w}))$ for some nonlinear function f, then posterior inference is even more challenging (this is equivalent to the problem of inference in Bayesian neural networks, see e.g., [Arb+23] for a review paper for the offline case, and [DMKM22; JCM24] for some recent online methods).

We can generalize the above methods to compute the belief state for the parameters of an MDP in the obvious way, but modeling both the reward function and state transition function.

Once we have computed the belief state, we can derive a policy with optimal regret using the methods like UCB (Section 1.4.3) or Thompson sampling (Section 1.4.4).

1.2.6 Optimization problems

The bandit problem is an example of a problem where the agent must interact with the world in order to collect information, but it does not otherwise affect the environment. Thus the agents internal belief state

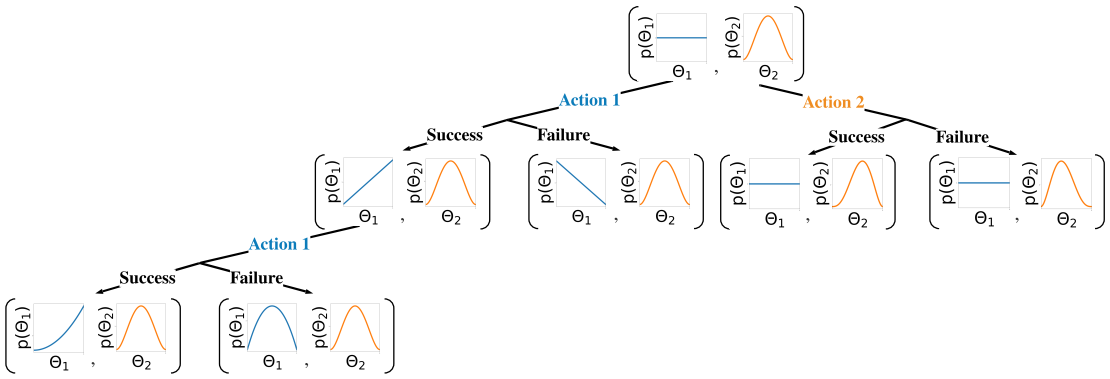


图 1.4：两臂贝塔 - 伯努利赌博机的顺序信念更新示意图。动作 1 的奖励先验是（蓝色）均匀分布 Beta(1, v2), v3; 动作 2 的奖励先验是（橙色）单峰分布 Beta(2, v6), v7。我们根据所选动作和观察到的奖励是否成功（1）或失败（0）来更新信念状态参数。

如果我们能解决这个（PO） MDP，我们就得到了探索 - 利用问题的最优解。

作为一个简单的例子，考虑一个上下文无关的**伯努利赌博机**，其中 $p_R(r|a) =$ 服从贝 $(r|\mu_a)$，并且 $\mu_a = p_R($ $r = 1|a) = R(a)$ 是采取行动 a 的期望奖励。唯一未知的参数是 $\boldsymbol{w} = \mu_{1:A}$。假设我们使用一个分解的贝塔先验

$$p_0(\boldsymbol{w}) = \prod_a \text{Beta}(\mu_a|\alpha_0^a, \beta_0^a) \tag{1.22}$$

在 $\boldsymbol{w} = (\mu_1, \ldots, \mu_K)$ 中。我们可以通过闭式计算得到后验概率。

$$p(\boldsymbol{w}|\mathcal{D}_t) = \prod_a \text{Beta}(\mu_a| \underbrace{\alpha_0^a + N_t^0(a)}_{\alpha_t^a}, \underbrace{\beta_0^a + N_t^1(a)}_{\beta_t^a}) \tag{1.23}$$

在哪里

$$N_t^r(a) = \sum_{i=1}^{t-1} \mathbb{I}(a_i = a, r_i = r) \tag{1.24}$$

这如图 1.4 所示，对于双臂伯努利赌博机。我们可以用类似的方法来处理高斯赌博机，其中 $p_R(r|a) = \mathcal{N}(r|\mu_a, \sigma_a^2)$。

在上下文赌博机的情况下，问题在概念上是相同的，但在计算上变得更加复杂。如果我们假设一个**线性回归赌博机**，$p_R(r|s, a; \boldsymbol{w}) = \mathcal{N}(r|\boldsymbol{\phi}(s, a)^\mathsf{T}\boldsymbol{w}, \sigma^2)$），我们可以使用贝叶斯线性回归以闭式形式精确计算 $p(\boldsymbol{w}|\mathcal{D}_t)$ 。如果我们假设一个**逻辑回归赌博机**，$p_R(r|s, a; \boldsymbol{w}) =$ 伯努利 $(r|\sigma(\boldsymbol{\phi}(s, a)^\mathsf{T}\boldsymbol{w}))$），我们必须使用近似方法来计算近似贝叶斯逻辑回归 $p(\boldsymbol{w}|\mathcal{D}_t)$。如果我们有一个**神经赌博机**的形式 $p_R(r|s, a; \boldsymbol{w}) = \mathcal{N}(r|f(s, a; \boldsymbol{w}))$ 对于某个非线性函数 f，那么后验推理就更加具有挑战性（这相当于贝叶斯神经网络的推理问题，例如，参见 [Arb+23] 关于离线案例的综述论文，以及 [DMKM22； JCM 24] 关于一些最近在线方法）。

我们可以将上述方法推广到以明显的方式计算 MDP 参数的信念状态，但需要同时建模奖励函数和状态转移函数。

一旦我们计算出信念状态，我们可以使用 UCB （第 1.4.3 节）或 Thompson 抽样（第 1.4.4 节）等方法推导出具有最优后悔度的策略。

1.2.6 优化问题

强盗问题是这样一个问题的例子，其中智能体必须与世界互动以收集信息，但不会以其他方式影响环境。因此，智能体的内部信念状态

changes over time, but the environment state does not.[7] Such problems commomly arise when we are trying to optimize a fixed but unknown function R. We can "query" the function by evaluating it at different points (parameter values), and in some cases, the resulting observation may also include gradient information. The agent's goal is to find the optimum of the function in as few steps as possible. We give some examples of this problem setting below.

1.2.6.1 Best-arm identification

In the standard multi-armed bandit problem our goal is to maximize the sum of expected rewards. However, in some cases, the goal is to determine the best arm given a fixed budget of T trials; this variant is known as **best-arm identification** [ABM10]. Formally, this corresponds to optimizing the **final reward** criterion:

$$V_{\pi,\pi_T} = \mathbb{E}_{p(a_{1:T},r_{1:T}|s_0,\pi)}\left[R(\hat{a})\right] \tag{1.25}$$

where $\hat{a} = \pi_T(a_{1:T}, r_{1:T})$ is the estimated optimal arm as computed by the **terminal policy** π_T applied to the sequence of observations obtained by the exploration policy π. This can be solved by a simple adaptation of the methods used for standard bandits.

1.2.6.2 Bayesian optimization

Bayesian optimization is a gradient-free approach to optimizing expensive blackbox functions. That is, we want to find

$$\boldsymbol{w}^* = \operatorname*{argmax}_{\boldsymbol{w}} R(\boldsymbol{w}) \tag{1.26}$$

for some unknown function R, where $\boldsymbol{w} \in \mathbb{R}^N$, using as few actions (function evaluations of R) as possible. This is essentially an "infinite arm" version of the best-arm identification problem [Tou14], where we replace the discrete choice of arms $a \in \{1, \ldots, K\}$ with the parameter vector $\boldsymbol{w} \in \mathbb{R}^N$. In this case, the optimal policy can be computed if the agent's state s_t is a belief state over the unknown function, i.e., $s_t = p(R|\boldsymbol{h}_t)$. A common way to represent this distribution is to use Gaussian processes. We can then use heuristics like expected improvement, knowledge gradient or Thompson sampling to implement the corresponding policy, $\boldsymbol{w}_t = \pi(s_t)$. For details, see e.g., [Gar23].

1.2.6.3 Active learning

Active learning is similar to BayesOpt, but instead of trying to find the point at which the function is largest (i.e., $\boldsymbol{w}^*$), we are trying to learn the whole function R, again by querying it at different points $\boldsymbol{w}_t$. Once again, the optimal strategy again requires maintaining a belief state over the unknown function, but now the best policy takes a different form, such as choosing query points to reduce the entropy of the belief state. See e.g., [Smi+23].

1.2.6.4 Stochastic Gradient Descent (SGD)

Finally we discuss how to interpret SGD as a sequential decision making process, following [Pow22]. The action space consists of querying the unknown function R at locations $\boldsymbol{a}_t = \boldsymbol{w}_t$, and observing the function value $r_t = R(\boldsymbol{w}_t)$; however, unlike BayesOpt, now we also observe the corresponding gradient $\boldsymbol{g}_t = \nabla_{\boldsymbol{w}} R(\boldsymbol{w})|_{\boldsymbol{w}_t}$, which gives non-local information about the function. The environment state contains the true function R which is used to generate the observations given the agent's actions. The agent state contains the current parameter estimate $\boldsymbol{w}_t$, and may contain other information such as first and second moments $\boldsymbol{m}_t$ and $\boldsymbol{v}_t$, needed by methods such as Adam. The update rule (for vanilla SGD) takes the form $\boldsymbol{w}_{t+1} = \boldsymbol{w}_t + \alpha_t \boldsymbol{g}_t$, where the stepsize α_t is chosen by the policy, $\alpha_t = \pi(s_t)$. The terminal policy has the form $\pi(s_T) = \boldsymbol{w}_T$.

[7] In the contextual bandit problem, the environment state (context) does change, but not in response to the agent's actions. Thus $p(o_t)$ is usually assumed to be a static distribution.

随时间变化，但环境状态不变。[7] 当我们试图优化一个固定但未知的函数 R 时，这类问题通常会出现。我们可以通过在不同的点（参数值）评估函数来“查询”该函数，在某些情况下，观察结果也可能包括梯度信息。代理的目标是以尽可能少的步骤找到函数的最优值。以下是一些此类问题设置的示例。

1.2.6.1 最佳臂识别

在标准的多臂老虎机问题中，我们的目标是最大化期望奖励的总和。然而，在某些情况下，目标是给定固定预算的 T 次试验来确定最佳臂；这种变体被称为**最佳臂识别** [ABM10]。形式上，这对应于优化**最终奖励**标准：

$$V_{\pi,\pi_T} = \mathbb{E}_{p(a_{1:T}, r_{1:T}|s_0,\pi)}\left[R(\hat{a})\right] \tag{1.25}$$

其中 $\hat{a} = \pi_T(a_{1:T}, r_{1:T})$ 是通过应用探索策略 π 获得的观察序列计算得到的估计最优臂，由 **终端策略** π_T 计算。这可以通过对标准伯努利方法进行简单调整来解决。

1.2.6.2 贝叶斯优化

贝叶斯优化是一种无梯度方法，用于优化昂贵的黑盒函数。也就是说，我们想要找到

$$\boldsymbol{w}^* = \underset{\boldsymbol{w}}{\operatorname{argmax}}\, R(\boldsymbol{w}) \tag{1.26}$$

对于某些未知的函数 R，其中 $\boldsymbol{w} \in \mathbb{R}^N$，尽可能少地使用动作（$R$ 的函数评估）来实现。这本质上是一个“无限臂”的最佳臂识别问题 [Tou14]，版本，其中我们将臂的离散选择 $a \in \{1, \ldots, K\}$ 替换为参数向量 $\boldsymbol{w} \in \mathbb{R}^N$。在这种情况下，如果代理的状态 s_t 是关于未知函数的信念状态，即 $s_t = p(R|\boldsymbol{h}_t)$，则可以计算最优策略。一种常见的表示这种分布的方法是使用高斯过程。然后，我们可以使用期望改进、知识梯度或 Thompson 抽样等启发式方法来实现相应的策略，$\boldsymbol{w}_t = \pi(s_t)$。有关详细信息，请参阅例如 [Gar23]。

1.2.6.3 活跃学习

主动学习类似于贝叶斯优化，但不是试图找到函数最大的点（即 $\boldsymbol{w}^*$），而是试图学习整个函数 R，再次通过在不同点查询它 $\boldsymbol{w}_t$。再次，最优策略仍然需要维护对未知函数的信念状态，但现在最佳策略采取了不同的形式，例如选择查询点以减少信念状态的熵。例如，参见 [Smi+23]。

1.2.6.4 随机梯度下降（SGD）

最后，我们讨论如何将 SGD 解释为一种顺序决策过程，遵循 [Pow22]。动作空间包括查询未知函数 R 在位置 $\boldsymbol{a}_t = \boldsymbol{w}_t$，并观察函数值 $r_t = R(\boldsymbol{w}_t)$；然而，与 BayesOpt 不同，现在我们还观察到相应的梯度 $\boldsymbol{g}_t = \nabla_{\boldsymbol{w}} R(\boldsymbol{w})|_{\boldsymbol{w}_t}$，这提供了关于函数的非局部信息。环境状态包含用于根据代理的动作生成观察结果的真函数 R。代理状态包含当前的参数估计 $\boldsymbol{w}_t$，可能还包含其他信息，如一阶和二阶矩 $\boldsymbol{m}_t$ 和 $\boldsymbol{v}_t$，这些信息是 Adam 等方法所需的。更新规则（对于 vanilla SGD）的形式为 $\boldsymbol{w}_{t+1} = \boldsymbol{w}_t + \alpha_t \boldsymbol{g}_t$，其中步长 α_t 由策略选择，$\alpha_t = \pi(s_t)$。终端策略的形式为 $\pi(s_T) = \boldsymbol{w}_T$。

[7] 在上下文赌博问题中，环境状态（上下文）确实会变化，但并非对代理的动作做出反应。因此，通常假设 $p(o_t)$ 是一个静态分布。

Approach	Method	Functions learned	On/Off	Section
Value-based	SARSA	$Q(s,a)$	On	Section 2.4
Value-based	Q-learning	$Q(s,a)$	Off	Section 2.5
Policy-based	REINFORCE	$\pi(a\|s)$	On	Section 3.2
Policy-based	A2C	$\pi(a\|s)$, $V(s)$	On	Section 3.3.1
Policy-based	TRPO/PPO	$\pi(a\|s)$, $A(s,a)$	On	Section 3.4.3
Policy-based	DDPG	$a = \pi(s)$, $Q(s,a)$	Off	Section 3.6.1
Policy-based	Soft actor-critic	$\pi(a\|s)$, $Q(s,a)$	Off	Section 3.5.4
Model-based	MBRL	$p(s'\|s,a)$	Off	Chapter 4

Table 1.1: Summary of some popular methods for RL. On/off refers to on-policy vs off-policy methods.

Although in principle it is possible to learn the learning rate (stepsize) policy using RL (see e.g., [Xu+17]), the policy is usually chosen by hand, either using a **learning rate schedule** or some kind of manually designed **adaptive learning rate** policy (e.g., based on second order curvature information).

1.3 Reinforcement Learning

In this section, we give a brief overview of how to compute optimal policies when the model of the environment is unknown; this is the core problem tackled by RL. We mostly focus on the MDP case, but discuss the POMDP case in Section 1.3.4.

We may categorize RL methods along two main dimensions: (1) by what the agent represents and learns: the value function, and/or the policy, and/or the model; (2) and by how actions are selected: **on-policy** (actions must be selected by the agent's current policy), and **off-policy** (actions can be select by any kind of policy, including human demonstrations). Table 1.1 lists a few representative examples. More details are given in the subsequent sections.

1.3.1 Value-based RL (Approximate Dynamic Programming)

In this section, we give a brief introduction to **value-based RL**, also called **Approximate Dynamic Programming** or **ADP**; see Chapter 2 for more details.

We introduced the value function $V_\pi(s)$ in Equation (1.1), which we repeat here for convenience:

$$V_\pi(s) \triangleq \mathbb{E}_\pi\left[G_0|s_0 = s\right] = \mathbb{E}_\pi\left[\sum_{t=0}^{\infty} \gamma^t r_t|s_0 = s\right] \tag{1.27}$$

The value function for the optimal policy π^* is known to satisfy the following recursive condition, known as **Bellman's equation**:

$$V_*(s) = \max_a R(s,a) + \gamma \mathbb{E}_{p_S(s'|s,a)}\left[V_*(s')\right] \tag{1.28}$$

This follows from the principle of **dynamic programming**, which computes the optimal solution to a problem (here the value of state s by combining the optimal solution of various subproblems (here the values of the next states s'). This can be used to derive the following learning rule:

$$V(s) \leftarrow V(s) + \eta[r + \gamma V(s') - V(s)] \tag{1.29}$$

where $s' \sim p_S(\cdot|s,a)$ is the next state sampled from the environment, and $r = R(s,a)$ is the observed reward. This is called **Temporal Difference** or **TD** learning (see Section 2.3.2 for details). Unfortunately, it is not clear how to derive a policy if all we know is the value function. We now describe a solution to this problem.

Approach	Method	Functions learned	On/Off	Section
Value-based	SARSA	$Q(s,a)$	On	Section 2.4
Value-based	Q-learning	$Q(s,a)$	Off	Section 2.5
Policy-based	REINFORCE	$\pi(a\|s)$	On	Section 3.2
Policy-based	A2C	$\pi(a\|s)$, $V(s)$	On	Section 3.3.1
Policy-based	TRPO/PPO	$\pi(a\|s)$, $A(s,a)$	On	Section 3.4.3
Policy-based	DDPG	$a=\pi(s)$, $Q(s,a)$	Off	Section 3.6.1
Policy-based	Soft actor-critic	$\pi(a\|s)$, $Q(s,a)$	Off	Section 3.5.4
Model-based	MBRL	$p(s'\|s,a)$	Off	Chapter 4

表 1.1：强化学习（RL）一些流行方法的总结。开 / 关指代策略性方法与非策略性方法。

尽管在原则上可以使用强化学习（例如，参见 [Xu+17]）来学习学习率（步长）策略，但策略通常是由人工选择的，要么使用一种**学习率调度**，要么是某种手动设计的**自适应学习率**策略（例如，基于二阶曲率信息）。

1.3 强化学习

在本节中，我们简要概述了在环境模型未知时如何计算最优策略；这是强化学习要解决的核心问题。我们主要关注 MDP 情况，但在第 1.3.4 节中讨论了 POMDP 情况。

我们将根据两个主要维度对强化学习（RL）方法进行分类：（1）根据智能体表示和学习的对象：价值函数、策略和 / 或模型；（2）根据动作的选择方式：**基于策略**（动作必须由智能体的当前策略选择），以及**非基于策略**（动作可以选择任何类型的策略，包括人类演示）。表 1.1 列出了几个代表性示例。更多细节将在后续章节中给出。

1.3.1 基于价值的强化学习（近似动态规划）

在本节中，我们简要介绍了**基于价值的强化学习**，也称为**近似动态规划**或 **ADP**；更多详情请参阅第 2 章。

我们引入了值函数 $V_\pi(s)$，在方程（1.1）中重复此处以方便查阅：

$$V_\pi(s) \triangleq \mathbb{E}_\pi\left[G_0|s_0=s\right] = \mathbb{E}_\pi\left[\sum_{t=0}^{\infty}\gamma^t r_t|s_0=s\right] \tag{1.27}$$

最优策略的价值函数 π^* 被知满足以下递归条件，被称为 **贝尔曼方程**：

$$V_*(s) = \max_a R(s,a) + \gamma \mathbb{E}_{p_S(s'|s,a)}\left[V_*(s')\right] \tag{1.28}$$

这遵循**动态规划**的原则，该原则通过结合各个子问题的最优解（此处为下一个状态 s' 的值）来计算问题的最优解（此处为状态的值）。这可以用来推导以下学习规则：

$$V(s) \leftarrow V(s) + \eta[r + \gamma V(s') - V(s)] \tag{1.29}$$

其中 $s' \sim p_S(\cdot|s,a)$ 是环境采样的下一个状态，而 $r = R(s,a)$ 是观察到的奖励。这被称为 **时间差分**或 **TD** 学习（详见第 2.3.2 节）。不幸的是，如果我们只知道值函数，则不清楚如何推导策略。我们现在描述解决这个问题的一个方法。

We first generalize the notion of value function to assigning a value to a state and action pair, by defining the **Q function** as follows:

$$Q_\pi(s,a) \triangleq \mathbb{E}_\pi\left[G_0|s_0=s,a_0=a\right] = \mathbb{E}_\pi\left[\sum_{t=0}^{\infty}\gamma^t r_t|s_0=s,a_0=a\right] \tag{1.30}$$

This quantity represents the expected return obtained if we start by taking action a in state s, and then follow π to choose actions thereafter. The Q function for the optimal policy satisfies a modified Bellman equation

$$Q_*(s,a) = R(s,a) + \gamma \mathbb{E}_{p_S(s'|s,a)}\left[\max_{a'} Q_*(s',a')\right] \tag{1.31}$$

This gives rise to the following TD update rule:

$$Q(s,a) \leftarrow r + \gamma \max_{a'} Q(s',a') - Q(s,a) \tag{1.32}$$

where we sample $s' \sim p_S(\cdot|s,a)$ from the environment. The action is chosen at each step from the implicit policy

$$a = \operatorname*{argmax}_{a'} Q(s,a') \tag{1.33}$$

This is called **Q learning** (see Section 2.5 for details),

1.3.2 Policy-based RL

In this section we give a brief introductin to **Policy-based RL**; for details see Chapter 3.

In policy-based methods, we try to directly maximize $J(\pi_{\boldsymbol{\theta}}) = \mathbb{E}_{p(s_0)}\left[V_\pi(s_0)\right]$ wrt the parameter's $\boldsymbol{\theta}$; this is called **policy search**. If $J(\pi_{\boldsymbol{\theta}})$ is differentiable wrt $\boldsymbol{\theta}$, we can use stochastic gradient ascent to optimize $\boldsymbol{\theta}$, which is known as **policy gradient** (see Section 3.1).

Policy gradient methods have the advantage that they provably converge to a local optimum for many common policy classes, whereas Q-learning may diverge when approximation is used (Section 2.5.2.4). In addition, policy gradient methods can easily be applied to continuous action spaces, since they do not need to compute $\operatorname{argmax}_a Q(s,a)$. Unfortunately, the score function estimator for $\nabla_{\boldsymbol{\theta}} J(\pi_{\boldsymbol{\theta}})$ can have a very high variance, so the resulting method can converge slowly.

One way to reduce the variance is to learn an approximate value function, $V_{\boldsymbol{w}}(s)$, and to use it as a baseline in the score function estimator. We can learn $V_{\boldsymbol{w}}(s)$ using using TD learning. Alternatively, we can learn an advantage function, $A_{\boldsymbol{w}}(s,a)$, and use it as a baseline. These policy gradient variants are called **actor critic** methods, where the actor refers to the policy $\pi_{\boldsymbol{\theta}}$ and the critic refers to $V_{\boldsymbol{w}}$ or $A_{\boldsymbol{w}}$. See Section 3.3 for details.

1.3.3 Model-based RL

In this section, we give a brief introduction to **model-based RL**; for more details, see Chapter 4.

Value-based methods, such as Q-learning, and policy search methods, such as policy gradient, can be very **sample inefficient**, which means they may need to interact with the environment many times before finding a good policy, which can be problematic when real-world interactions are expensive. In model-based RL, we first learn the MDP, including the $p_S(s'|s,a)$ and $R(s,a)$ functions, and then compute the policy, either using approximate dynamic programming on the learned model, or doing lookahead search. In practice, we often interleave the model learning and planning phases, so we can use the partially learned policy to decide what data to collect, to help learn a better model.

我们首先将值函数的概念推广到为状态和动作对赋予值，通过定义如下 **Q 函数**：

$$Q_\pi(s,a) \triangleq \mathbb{E}_\pi\left[G_0|s_0=s,a_0=a\right] = \mathbb{E}_\pi\left[\sum_{t=0}^{\infty}\gamma^t r_t|s_0=s,a_0=a\right] \tag{1.30}$$

这个数量表示如果我们从状态 s 开始采取行动 a，然后遵循 π 选择后续行动，所获得的预期回报。最优策略的 Q 函数满足修改后的贝尔曼方程

$$Q_*(s,a) = R(s,a) + \gamma\mathbb{E}_{p_S(s'|s,a)}\left[\max_{a'} Q_*(s',a')\right] \tag{1.31}$$

这导致了以下 TD 更新规则：

$$Q(s,a) \leftarrow r + \gamma\max_{a'} Q(s',a') - Q(s,a) \tag{1.32}$$

我们在环境中采样 $s' \sim p_S(\cdot|s,a)$。在每个步骤中，从隐式策略中选择动作。

$$a = \operatorname*{argmax}_{a'} Q(s,a') \tag{1.33}$$

这被称为 **Q 学习**（详见第 2.5 节以获取详细信息），

1.3.2 基于策略的强化学习

本节简要介绍了**基于策略的强化学习**；详情请见第 3 章。

在基于策略的方法中，我们试图直接最大化 $J(\pi_{\boldsymbol{\theta}}) = \mathbb{E}_{p(s_0)}[V_\pi(s_0)]$) 关于参数的 $\boldsymbol{\theta}$；这被称为 **策略搜索**。如果 $J(\pi_{\boldsymbol{\theta}})$ 相对于 $\boldsymbol{\theta}$ 可微，我们可以使用随机梯度上升来优化 $\boldsymbol{\theta}$，这被称为 **策略梯度**（见第 3.1 节）

策略梯度方法具有优势，它们对于许多常见的策略类别可以证明收敛到局部最优，而 Q- 学习在近似使用时可能会发散（第 2.5.2.4 节）。此外，策略梯度方法可以轻松应用于连续动作空间，因为它们不需要计算 $\operatorname{argmax}_a Q(s,a)$。不幸的是，对于 $\nabla_{\boldsymbol{\theta}} J(\pi_{\boldsymbol{\theta}})$ 的得分函数估计器可能会有很高的方差，因此 resulting method can converge slowly。

降低方差的一种方法是通过学习一个近似值函数，$V_{\boldsymbol{w}}(s)$，并将其用作分数函数估计器的基线。我们可以使用 TD 学习来学习 $V_{\boldsymbol{w}}(s)$。或者，我们可以学习一个优势函数，$A_{\boldsymbol{w}}(s,a)$，并将其用作基线。这些策略梯度变体被称为**演员 - 评论家**方法，其中演员指的是策略 $\pi_{\boldsymbol{\theta}}$，评论家指的是 $V_{\boldsymbol{w}}$ 或 $A_{\boldsymbol{w}}$。有关详细信息，请参阅第 3.3 节。

1.3.3 基于模型的强化学习

本节中，我们简要介绍了基于**模型的强化学习**；更多详情请参阅第 4 章。

基于价值的算法，如 Q 学习，以及策略搜索方法，如策略梯度，可能非常低效，这意味着它们可能需要与环境的多次交互才能找到良好的策略，这在现实世界的交互成本高昂时可能是个问题。在基于模型的强化学习中，我们首先学习 MDP，包括状态 - 动作值函数（状态 - 动作值函数，状态 - 动作值函数）和状态 - 动作优势函数（状态 - 动作优势函数，状态 - 动作优势函数），然后计算策略，要么在学习的模型上使用近似动态规划，要么进行前瞻性搜索。在实践中，我们通常将模型学习和规划阶段交替进行，这样我们就可以使用部分学习的策略来决定要收集哪些数据，以帮助学习更好的模型。

1.3.4 Dealing with partial observability

In an MDP, we assume that the state of the environment s_t is the same as the observation o_t obtained by the agent. But in many problems, the observation only gives partial information about the underlying state of the world (e.g., a rodent or robot navigating in a maze). This is called **partial observability**. In this case, using a policy of the form $a_t = \pi(o_t)$ is suboptimal, since o_t does not give us complete state information. Instead we need to use a policy of the form $a_t = \pi(\boldsymbol{h}_t)$, where $\boldsymbol{h}_t = (a_1, o_1, \ldots, a_{t-1}, o_t)$ is the entire past history of observations and actions, plus the current observation. Since depending on the entire past is not tractable for a long-lived agent, various approximate solution methods have been developed, as we summarize below.

1.3.4.1 Optimal solution

If we know the true latent structure of the world (i.e., both $p(o|z)$ and $p(z'|z,a)$, to use the notation of Section 1.1.2), then we can use solution methods designed for POMDPs, discussed in Section 1.2.1. This requires using Bayesian inference to compute a belief state, $\boldsymbol{b}_t = p(\boldsymbol{z}_t|\boldsymbol{h}_t)$ (see Section 1.2.5), and then using this belief state to guide our decisions. However, learning the parameters of a POMDP (i.e., the generative latent world model) is very difficult, as is recursively computing and updating the belief state, as is computing the policy given the belief state. Indeed, optimally solving POMDPs is known to be computationally very difficult for any method [PT87; KLC98]. So in practice simpler approximations are used. We discuss some of these below. (For more details, see [Mur00].)

Note that it is possible to marginalize out the POMDP latent state z_t, to derive a prediction over the next observable state, $p(\boldsymbol{o}_{t+1}|\boldsymbol{h}_t, \boldsymbol{a}_t)$. This can then become a learning target for a model, that is trained to directly predict future observations, without explicitly invoking the concept of latent state. This is called a **predictive state representation** or **PSR** [LS01]. This is related to the idea of **observable operator models** [Jae00], and to the concept of successor representations which we discuss in Section 4.4.2.

1.3.4.2 Finite observation history

The simplest solution to the partial observability problem is to define the state to be a finite history of the last k observations, $\boldsymbol{s}_t = \boldsymbol{h}_{t-k:t}$; when the observations $\boldsymbol{o}_t$ are images, this is often called **frame stacking**. We can then use standard MDP methods. Unfortunately, this cannot capture long-range dependencies in the data.

1.3.4.3 Stateful (recurrent) policies

A more powerful approach is to use a stateful policy, that can remember the entire past, and not just respond to the current input or last k frames. For example, we can represent the policy by an RNN (recurrent neural network), as proposed in the **R2D2** paper [Kap+18], and used in many other papers. Now the hidden state $\boldsymbol{z}_t$ of the RNN will implicitly summarize the past observations, $\boldsymbol{h}_t$, and can be used in lieu of the state s_t in any standard RL algorithm.

RNNs policies are widely used, and this method is often effective in solving partially observed problems. However, they typically will not plan to perform information-gathering actions, since there is no explicit notion of belief state or uncertainty. However, such behavior can arise via meta-learning [Mik+20].

1.3.5 Software

Implementing RL algorithms is much trickier than methods for supervised learning, or generative methods such as language modeling and diffusion, all of which have stable (easy-to-optimize) loss functions. Therefore it is often wise to build on existing software rather than starting from scratch. We list some useful libraries in Section 1.3.5.

In addition, RL experiments can be very high variance, making it hard to draw valid conclusions. See [Aga+21b; Pat+24; Jor+24] for some recommended experimental practices. For example, when reporting performance across different environments, with different intrinsic difficulties (e.g., different kinds of Atari

1.3.4 处理部分可观测性

在马尔可夫决策过程（ MDP ）中，我们假设环境的当前状态 s_t 与智能体获得的观察 o_t 相同。但在许多问题中，观察只能提供关于世界潜在状态的局部信息（例如，老鼠或机器人在迷宫中导航）。这被称为**部分可观测性**。在这种情况下，使用形式为 $a_t = \pi(o_t)$ 的策略是不理想的，因为 o_t 没有给我们提供完整的状态信息。相反，我们需要使用形式为 $a_t = \pi(\boldsymbol{h}_t)$ 的策略，其中 $\boldsymbol{h}_t = (a_1, o_1, \ldots, a_{t-1}, o_t)$ 是整个观察和行动的历史，加上当前的观察。由于依赖于整个过去对于长期生存的智能体来说是不切实际的，因此已经开发出各种近似解法，如下所述。

1.3.4.1 最佳解决方案

如果我们知道世界的真实潜在结构（即，p （ $o|z$ ）和 p （ $z'|z$，a ），使用第 1.1.2 节中的符号），那么我们可以使用为 POMDPs 设计的解决方案，如第 1.2.1 节所述。这需要使用贝叶斯推理来计算信念状态，$\boldsymbol{b}_t = p$ （ $\boldsymbol{z}_t|\boldsymbol{h}_t$ ）（见第 1.2.5 节），然后使用这个信念状态来指导我们的决策。然而，学习 POMDP 的参数（即，生成潜在世界模型）非常困难，递归地计算和更新信念状态也是如此，给定信念状态计算策略也是如此。事实上，已知对于任何方法 [PT87； KLC98]，最优解决 POMDPs 在计算上非常困难。因此，在实践中，使用更简单的近似。我们下面讨论其中一些。（更多细节，见 [Mur00]。）

请注意，可以边缘化 POMDP 潜在状态 z_t，以推导出对下一个可观察状态的预测，$p(\boldsymbol{o}_{t+1}|\boldsymbol{h}_t,\boldsymbol{a}_t)$。这可以成为模型的预测目标，该模型被训练以直接预测未来的观察结果，而不明确调用潜在状态的概念。这被称为**预测状态表示**或 **PSR** [LS01]。这与**可观察算子模型** [Jae00]，以及我们在第 4.4.2 节中讨论的后继表示概念相关。

1.3.4.2 有限观测历史

最简单的解决部分可观测性问题的方法是将状态定义为最后 k 次观测的有限历史，$\boldsymbol{s}_t = \boldsymbol{h}_{t-k:t}$； 当观测 $\boldsymbol{o}_t$ 是图像时，这通常被称为 **帧堆叠**。然后我们可以使用标准的 MDP 方法。不幸的是，这无法捕捉数据中的长距离依赖关系。

1.3.4.3 有状态（循环）策略

更强大的方法是使用有状态的策略，它可以记住整个过去，而不仅仅是响应当前的输入或最后的 k 帧。例如，我们可以通过一个 RNN （循环神经网络）来表示策略，如 R2D2 **论文** 中提出，并在许多其他论文中使用。现在，RNN 的隐藏状态 $\boldsymbol{z}_t$ 将隐式总结过去的观察，$\boldsymbol{h}_t$，并可用于任何标准 RL 算法中的状态 s_t 。

RNNs 策略被广泛使用，这种方法通常在解决部分观察问题中非常有效。然而，它们通常不会计划执行信息收集行动，因为没有明确的信念状态或不确定性的概念。但是，这种行为可以通过元学习 [Mik+20] 出现。

1.3.5 软件

实现 RL 算法比监督学习方法或语言建模和扩散等生成方法要复杂得多，所有这些方法都具有稳定的（易于优化的）损失函数。因此，通常明智的做法是建立在现有的软件之上，而不是从头开始。我们在第 1.3.5 节中列出了一些有用的库。

此外，强化学习实验可能具有很高的方差，这使得很难得出有效的结论。参见 [Aga+21b； Pat+24； Jor+24] 关于一些推荐的实验实践。例如，在报告不同环境下的性能时，具有不同的内在难度（例如，不同类型的 Atari

URL	Language	Comments
Stoix	Jax	Mini-library with many methods (including MBRL)
PureJaxRL	Jax	Single files with DQN; PPO, DPO
JaxRL	Jax	Single files with AWAC, DDPG, SAC, SAC+REDQ
Stable Baselines Jax	Jax	Library with DQN, CrossQ, TQC; PPO, DDPG, TD3, SAC
Jax Baselines	Jax	Library with many methods
Rejax	Jax	Library with DDQN, PPO, (discrete) SAC, DDPG
Dopamine	Jax/TF	Library with many methods
Rlax	Jax	Library of RL utility functions (used by Acme)
Acme	Jax/TF	Library with many methods (uses rlax)
CleanRL	PyTorch	Single files with many methods
Stable Baselines 3	PyTorch	Library with DQN; A2C, PPO, DDPG, TD3, SAC, HER
TianShou	PyTorch	Library with many methods (including offline RL)

Table 1.2: Some open source RL software.

games), [Aga+21b] recommend reporting the **interquartile mean** (IQM) of the performance metric, which is the mean of the samples between the 0.25 and 0.75 percentiles, (this is a special case of a trimmed mean). Let this estimate be denoted by $\hat{\mu}(\mathcal{D}_i)$, where $\mathcal{D}$ is the empirical data (e.g., reward vs time) from the i'th run. We can estimate the uncertainty in this estimate using a nonparametric method, such as bootstrap resampling, or a parametric approximation, such as a Gaussian approximation. (This requires computing the standard error of the mean, $\frac{\hat{\sigma}}{\sqrt{n}}$, where n is the number of trials, and $\hat{\sigma}$ is the estimated standard deviation of the (trimmed) data.)

1.4 Exploration-exploitation tradeoff

A fundamental problem in RL with unknown transition and reward models is to decide between choosing actions that the agent knows will yield high reward, or choosing actions whose reward is uncertain, but which may yield information that helps the agent get to parts of state-action space with even higher reward. This is called the **exploration-exploitation tradeoff**. In this section, we discuss various solutions.

1.4.1 Simple heuristics

We start with a policy based on pure exploitation. This is known as the **greedy policy**, $a_t = \text{argmax}_a Q(s, a)$. We can add exploration to this by sometimes picking some other, non-greedy action.

One approach is to use an **ϵ-greedy** policy π_ϵ, parameterized by $\epsilon \in [0, 1]$. In this case, we pick the greedy action wrt the current model, $a_t = \text{argmax}_a \hat{R}_t(s_t, a)$ with probability $1 - \epsilon$, and a random action with probability ϵ. This rule ensures the agent's continual exploration of all state-action combinations. Unfortunately, this heuristic can be shown to be suboptimal, since it explores every action with at least a constant probability $\epsilon/|\mathcal{A}|$, although this can be solved by annealing ϵ to 0 over time.

Another problem with ϵ-greedy is that it can result in "dithering", in which the agent continually changes its mind about what to do. In [DOB21] they propose a simple solution to this problem, known as **ϵz-greedy**, that often works well. The idea is that with probability $1 - \epsilon$ the agent exploits, but with with probability ϵ the agent explores by repeating the sampled action for $n \sim z()$ steps in a row, where $z(n)$ is a distribution over the repeat duration. This can help the agent escape from local minima.

Another approach is to use **Boltzmann exploration**, which assigns higher probabilities to explore more promising actions, taking itno account the reward function. That is, we use a polocy of the form

$$\pi_\tau(a|s) = \frac{\exp(\hat{R}_t(s_t, a)/\tau)}{\sum_{a'} \exp(\hat{R}_t(s_t, a')/\tau)} \tag{1.34}$$

where $\tau > 0$ is a temperature parameter that controls how entropic the distribution is. As τ gets close to 0, π_τ becomes close to a greedy policy. On the other hand, higher values of τ will make $\pi(a|s)$ more uniform,

URL	Language	Comments
Stoix	Jax	Mini-library with many methods (including MBRL)
PureJaxRL	Jax	Single files with DQN; PPO, DPO
JaxRL	Jax	Single files with AWAC, DDPG, SAC, SAC+REDQ
Stable Baselines Jax	Jax	Library with DQN, CrossQ, TQC; PPO, DDPG, TD3, SAC
Jax Baselines	Jax	Library with many methods
Rejax	Jax	Library with DDQN, PPO, (discrete) SAC, DDPG
Dopamine	Jax/TF	Library with many methods
Rlax	Jax	Library of RL utility functions (used by Acme)
Acme	Jax/TF	Library with many methods (uses rlax)
CleanRL	PyTorch	Single files with many methods
Stable Baselines 3	PyTorch	Library with DQN; A2C, PPO, DDPG, TD3, SAC, HER
TianShou	PyTorch	Library with many methods (including offline RL)

表 1.2：一些开源强化学习软件。

游戏），[Aga+21b] 建议报告性能指标的**四分位数均值**（IQM），这是介于 0.25 和 0.75 百分位数之间的样本均值，（这是截断均值的特殊情况）。用 $\hat{\mu}(\mathcal{D}_i)$ 表示这个估计，其中 $\mathcal{D}$ 是第 i 次运行的经验数据（例如，奖励与时间），我们可以使用非参数方法（如自助重采样）或参数近似（如高斯近似）来估计这个估计的不确定性。（这需要计算平均值的标准误差 $\frac{\hat{\sigma}}{\sqrt{n}}$，其中 n 是试验次数，$\hat{\sigma}$ 是（截断的）数据的估计标准差。）

1.4 探索 - 利用权衡

强化学习（RL）中，未知转移和奖励模型的基本问题在于，在已知将产生高奖励的动作和可能产生不确定奖励但可能帮助代理到达更高奖励状态动作空间的动作之间做出选择。这被称为**探索 - 利用权衡**。在本节中，我们将讨论各种解决方案。

1.4.1 简单启发式

我们以基于纯粹剥削的策略开始。这被称为**贪婪策略**，$a_t = \text{argmax}_a Q(s, a)$。我们可以通过有时选择一些其他非贪婪动作来添加探索。

一种方法是使用一个由 $\epsilon \in [0, 1]$ 参数化的 ϵ- **贪婪** 策略 $\pi_{\epsilon o}$。在这种情况下，我们以概率 $1 - \epsilon$ 选择相对于当前模型 $a_t = \text{argmax}_a \hat{R}_t(s_t,a)$ 的贪婪动作，并以概率 ϵ 选择随机动作。此规则确保代理持续探索所有状态 - 动作组合。不幸的是，这个启发式方法可以证明是次优的，因为它至少以常数概率 $\epsilon/|\mathcal{A}|$ 探索每个动作，尽管这可以通过将 ϵ 在时间上退火到 0 来解决。

另一个与 ϵ- 贪婪相关的问题是，它可能导致“抖动”，在这种情况下，智能体不断改变其行动的决策。在 [DOB21] 中，他们提出了一个简单的解决方案，称为 ϵz- **贪婪**，通常效果良好。其想法是，以概率 $1 - \epsilon$ 智能体利用，但以概率 ϵ 智能体通过重复采样的动作，连续 $n \sim z()$ 步，其中 $z(n)$ 是重复持续时间的分布。这可以帮助智能体摆脱局部最小值。

另一种方法是使用**玻尔兹曼探索**，它为探索更有希望的动作分配更高的概率，而不考虑奖励函数。也就是说，我们采用以下形式的策略：

$$\pi_\tau(a|s) = \frac{\exp(\hat{R}_t(s_t, a)/\tau)}{\sum_{a'} \exp(\hat{R}_t(s_t, a')/\tau)} \tag{1.34}$$

$\tau > 0$ 是一个控制分布熵度的温度参数。当 τ 接近 0 时，π_τ 接近贪婪策略。另一方面，τ 的更高值将使 $\pi(a|s)$ 更均匀。

$\hat{R}(s,a_1)$	$\hat{R}(s,a_2)$	$\pi_\epsilon(a\|s_1)$	$\pi_\epsilon(a\|s_2)$	$\pi_\tau(a\|s_1)$	$\pi_\tau(a\|s_2)$
1.00	9.00	0.05	0.95	0.00	1.00
4.00	6.00	0.05	0.95	0.12	0.88
4.90	5.10	0.05	0.95	0.45	0.55
5.05	4.95	0.95	0.05	0.53	0.48
7.00	3.00	0.95	0.05	0.98	0.02
8.00	2.00	0.95	0.05	1.00	0.00

Table 1.3: Comparison of ϵ-greedy policy (with $\epsilon = 0.1$) and Boltzmann policy (with $\tau = 1$) for a simple MDP with 6 states and 2 actions. Adapted from Table 4.1 of [GK19].

and encourage more exploration. Its action selection probabilities can be much "smoother" with respect to changes in the reward estimates than ϵ-greedy, as illustrated in Table 1.3.

The Boltzmann policy explores equally widely in all states. An alternative approach is to try to explore (state,action) combinations where the consequences of the outcome might be uncertain. This can be achived using an **exploration bonus** $R_t^b(s,a)$, which is large if the number of times we have tried actioon a in state s is small. We can then add R^b to the regular reward, to bias the behavior in a way that will hopefully cause the agent to learn useful information about the world. This is called an **intrinsic reward** function (Section 5.2.4).

1.4.2 Methods based on the belief state MDP

We can compute an optimal solution to the exploration-exploitation tradeoff by adopting a Bayesian approach to the problem. We start by computing the belief state MDP, as discussed in Section 1.2.5. We then compute the optimal policy, as we explain below.

1.4.2.1 Bandit case (Gittins indices)

Suppose we have a way to compute the recursively compute the belief state over model parameters, $p(\boldsymbol{\theta}_t|\mathcal{D}_{1:t})$. How do we use this to solve for the policy in the resulting belief state MDP?

In the special case of context-free bandits with a finite number of arms, the optimal policy of this belief state MDP can be computed using dynamic programming. The result can be represented as a table of action probabilities, $\pi_t(a_1,\ldots,a_K)$, for each step; this are known as **Gittins indices** [Git89] (see [PR12; Pow22] for a detailed explanation). However, computing the optimal policy for general contextual bandits is intractable [PT87].

1.4.2.2 MDP case (Bayes Adaptive MDPs)

We can extend the above techniques to the MDP case by constructing a **BAMDP**, which stands for "Bayes-Adaptive MDP" [Duf02]. However, this is computationally intractable to solve, so various approximations are made (see e.g., [Zin+21; AS22; Mik+20]).

1.4.3 Upper confidence bounds (UCBs)

The optimal solution to explore-exploit is intractable. However, an intuitively sensible approach is based on the principle known as "**optimism in the face of uncertainty**" (OFU). The principle selects actions greedily, but based on optimistic estimates of their rewards. The optimality of this approach is proved in the **R-Max** paper of [Ten02], which builds on the earlier **E3** paper of [KS02].

The most common implementation of this principle is based on the notion of an **upper confidence bound** or **UCB**. We will initially explain this for the bandit case, then extend to the MDP case.

$\hat{R}(s,a_1)$	$\hat{R}(s,a_2)$	$\pi_\epsilon(a\|s_1)$	$\pi_\epsilon(a\|s_2)$	$\pi_\tau(a\|s_1)$	$\pi_\tau(a\|s_2)$
1.00	9.00	0.05	0.95	0.00	1.00
4.00	6.00	0.05	0.95	0.12	0.88
4.90	5.10	0.05	0.95	0.45	0.55
5.05	4.95	0.95	0.05	0.53	0.48
7.00	3.00	0.95	0.05	0.98	0.02
8.00	2.00	0.95	0.05	1.00	0.00

表 1.3：具有 6 个状态和 2 个动作的简单 MDP 中，ϵ- 贪婪策略（含 $\epsilon = 0.1$）与 Boltzmann 策略（含 $\tau = 1$）的比较。改编自 GK/ 第 4.1 表。*19*/

并鼓励更多探索。其动作选择概率相对于奖励估计的变化可以比 ϵ- 贪婪策略 “ 更平滑 ”，如表 1.3 所示。

玻尔兹曼策略在所有状态下探索范围相同。一种替代方法是尝试探索（状态，动作）组合，其中结果可能不确定。这可以通过使用**探索奖励** $R_t^b(s,a)$ 来实现，如果我们在状态中尝试动作 a 的次数很少，这个奖励就很大。然后我们可以将 R^b 添加到常规奖励中，以偏置行为，从而希望使智能体学习有关世界的有用信息。这被称为**内在奖励**函数（第 5.2.4 节）。

1.4.2 基于信念状态 MDP 的方法

我们可以通过采用贝叶斯方法来解决这个问题，计算出探索 - 利用权衡的最优解。我们首先计算信念状态 MDP，如第 1.2.5 节所述。然后，我们计算最优策略，如下所述。

1.4.2.1 布兰德特案例（吉丁斯指数）

假设我们有一种方法来递归地计算信念状态模型参数，$p(\boldsymbol{\theta}_t|\mathcal{D}_{1:t})$。我们如何利用这个方法来解决在结果信念状态 MDP 中的策略?

在具有有限数量臂的上下文无关赌博机的特殊情况下，可以使用动态规划计算此信念状态 MDP 的最优策略。结果可以用动作概率表表示，$\pi_t(a_1,\ldots,a_K)$，对于每一步；这些被称为**吉丁斯指数** [Git89]（见 [PR12； Pow22] 以获取详细解释）。然而，计算一般上下文赌博机的最优策略是不可行的 [PT87]。

1.4.2.2 MDP 案例（贝叶斯自适应 MDP）

我们可以通过构建 **BAMDP** 来将上述技术扩展到 MDP 情况，它代表 “ 贝叶斯自适应 MDP”[Duf02]。然而，这难以在计算上解决，因此做出了各种近似（例如，参见 [Zin+21； AS22； Mik+20]）。

1.4.3 上界置信度（ UCBs ）

探索 - 利用的最佳方案是不可行的。然而，一个直观合理的方法是基于被称为 “ 面对不确定性的乐观主义 ”（ OFU ）的原则。该原则贪婪地选择行动，但基于对它们奖励的乐观估计。这种方法的最优性在 R-Max 论文中得到证明，该论文基于 KS 早期 E3 论文。

这种原则最常见的形式是基于**上置信界**或 **UCB** 的概念。我们首先解释这个概念在多臂老虎机案例中的应用，然后扩展到马尔可夫决策过程案例。

1.4.3.1 Basic idea

To use a UCB strategy, the agent maintains an optimistic reward function estimate $\tilde{R}_t$, so that $\tilde{R}_t(s_t, a) \geq R(s_t, a)$ for all a with high probability, and then chooses the greedy action accordingly:

$$a_t = \operatorname*{argmax}_a \tilde{R}_t(s_t, a) \tag{1.35}$$

UCB can be viewed a form of **exploration bonus**, where the optimistic estimate encourages exploration. Typically, the amount of optimism, $\tilde{R}_t - R$, decreases over time so that the agent gradually reduces exploration. With properly constructed optimistic reward estimates, the UCB strategy has been shown to achieve near-optimal regret in many variants of bandits [LS19]. (We discuss regret in Section 1.1.4.)

The optimistic function $\tilde{R}$ can be obtained in different ways, sometimes in closed forms, as we discuss below.

1.4.3.2 Bandit case: Frequentist approach

A frequentist approach to computing a confidence bound can be based on a **concentration inequality** [BLM16] to derive a high-probability upper bound of the estimation error: $|\hat{R}_t(s, a) - R_t(s, a)| \leq \delta_t(s, a)$, where $\hat{R}_t$ is a usual estimate of R (often the MLE), and δ_t is a properly selected function. An optimistic reward is then obtained by setting $\tilde{R}_t(s, a) = \hat{R}_t(s, a) + \delta_t(s, a)$.

As an example, consider again the context-free Bernoulli bandit, $R(a) \sim \text{Ber}(\mu(a))$. The MLE $\hat{R}_t(a) = \hat{\mu}_t(a)$ is given by the empirical average of observed rewards whenever action a was taken:

$$\hat{\mu}_t(a) = \frac{N_t^1(a)}{N_t(a)} = \frac{N_t^1(a)}{N_t^0(a) + N_t^1(a)} \tag{1.36}$$

where $N_t^r(a)$ is the number of times (up to step $t-1$) that action a has been tried and the observed reward was r, and $N_t(a)$ is the total number of times action a has been tried:

$$N_t(a) = \sum_{s=1}^{t-1} \mathbb{I}(a_t = a) \tag{1.37}$$

Then the **Chernoff-Hoeffding inequality** [BLM16] leads to $\delta_t(a) = c/\sqrt{N_t(a)}$ for some constant c, so

$$\tilde{R}_t(a) = \hat{\mu}_t(a) + \frac{c}{\sqrt{N_t(a)}} \tag{1.38}$$

1.4.3.3 Bandit case: Bayesian approach

We can also derive an upper confidence about using Bayesian inference. If we use a beta prior, we can compute the posterior in closed form, as shown in Equation (1.23). The posterior mean is $\hat{\mu}_t(a) = \mathbb{E}[\mu(a)|\boldsymbol{h}_t] = \frac{\alpha_t^a}{\alpha_t^a + \beta_t^a}$, and the posterior standard deviation is approximately

$$\hat{\sigma}_t(a) = \sqrt{\mathbb{V}[\mu(a)|\boldsymbol{h}_t]} \approx \sqrt{\frac{\hat{\mu}_t(a)(1 - \hat{\mu}_t(a))}{N_t(a)}} \tag{1.39}$$

We can use similar techniques for a Gaussian bandit, where $p_R(R|a, \boldsymbol{\theta}) = \mathcal{N}(R|\mu_a, \sigma_a^2)$, μ_a is the expected reward, and σ_a^2 the variance. If we use a conjugate prior, we can compute $p(\mu_a, \sigma_a|\mathcal{D}_t)$ in closed form. Using an uninformative version of the conjugate prior, we find $\mathbb{E}[\mu_a|\boldsymbol{h}_t] = \hat{\mu}_t(a)$, which is just the empirical mean of rewards for action a. The uncertainty in this estimate is the standard error of the mean, i.e., $\sqrt{\mathbb{V}[\mu_a|\boldsymbol{h}_t]} = \hat{\sigma}_t(a)/\sqrt{N_t(a)}$, where $\hat{\sigma}_t(a)$ is the empirical standard deviation of the rewards for action a.

Once we have computed the mean and posterior standard deviation, we define the optimistic reward estimate as

$$\tilde{R}_t(a) = \hat{\mu}_t(a) + c\hat{\sigma}_t(a) \tag{1.40}$$

for some constant c that controls how greedy the policy is. See Figure 1.5 for an illustration. We see that this is similar to the frequentist method based on concentration inequalities, but is more general.

1.4.3.1 基本思想

为了使用 UCB 策略，智能体维护一个乐观的奖励函数估计 $\tilde{R}_t$，以便在所有 a 中，以高概率满足 $\tilde{R}_t(s_t,a) \geq R(s_t,a)$，然后相应地选择贪婪动作：

$$a_t = \operatorname*{argmax}_a \tilde{R}_t(s_t, a) \tag{1.35}$$

UCB 可以被视为一种**探索奖励**的形式，其中乐观估计鼓励探索。通常，乐观程度 $\tilde{R}_t - R$会随着时间的推移而降低，从而使代理逐渐减少探索。通过适当构建的乐观奖励估计，UCB 策略已被证明在许多带枪变体中实现了接近最优的遗憾 [LS19]。（我们在第 1.1.4 节中讨论遗憾。）

乐观函数 $\tilde{R}$ 可以通过不同的方式获得，有时是闭式形式，如下文所述。

1.4.3.2 布兰德特案例：频率主义方法

基于频率主义方法计算置信界限可以基于一个**集中不等式** [BLM16] 推导出估计误差的高概率上界：$|\hat{R}_t(s,a) - R_t(s,a)| \leq \delta_t(s,a)$，其中 $\hat{R}_t$ 是 R （通常是 MLE ）的常规估计，而 δ_t 是一个正确选择的函数。然后通过设置 $\tilde{R}_t(s,a) = \hat{R}_t(s,a) + \delta_t(s,a)$. 获得乐观奖励。

作为一个例子，再次考虑无上下文伯努利赌博机，$R(a) \sim \text{Ber}(\mu(a))$。当采取行动 a 时，最大似然估计 $\hat{R}_t(a)=\hat{\mu}_t(a)$ 由观察到的奖励的实证平均值给出：

$$\hat{\mu}_t(a) = \frac{N_t^1(a)}{N_t(a)} = \frac{N_t^1(a)}{N_t^0(a) + N_t^1(a)} \tag{1.36}$$

$N_t^r(a)$ 表示在步骤 $t-1$ 之前尝试动作 a 的次数以及观察到的奖励为 r，而 $N_t(a)$ 表示尝试动作 a 的总次数：

$$N_t(a) = \sum_{s=1}^{t-1} \mathbb{I}(a_t = a) \tag{1.37}$$

Then the **Chernoff-Hoeffding 不等式** [BLM16] 导致 $\delta_t(a) = c/\sqrt{N_t(a)}$) for some constant c, so

$$\tilde{R}_t(a) = \hat{\mu}_t(a) + \frac{c}{\sqrt{N_t(a)}} \tag{1.38}$$

1.4.3.3 带刀案例：贝叶斯方法

我们可以推导出使用贝叶斯推断的上限置信度。如果我们使用贝塔先验，我们可以以封闭形式计算后验，如方程（ 1.23 ）所示。后验均值是 $\hat{\mu}_t(a) = \mathbb{E}[\mu(a)|\boldsymbol{h}_t] = \frac{\alpha^a}{\alpha_t^a + \beta_t^a}$，后验标准差大约为

$$\hat{\sigma}_t(a) = \sqrt{\mathbb{V}[\mu(a)|\boldsymbol{h}_t]} \approx \sqrt{\frac{\hat{\mu}_t(a)(1-\hat{\mu}_t(a))}{N_t(a)}} \tag{1.39}$$

我们可以使用类似的技术来处理高斯探险者，其中 $p_R(R|a,\boldsymbol{\theta}) = \mathcal{N}(R|\mu_a, \sigma_a^2)$，$\mu_a$ 是期望奖励，σ_a^2 是方差。如果我们使用共轭先验，我们可以以闭式计算 $p(\mu_a,\sigma_a|\mathcal{D}_t)$。使用非信息性版本的共轭先验，我们找到 $\mathbb{E}[\mu_a|\boldsymbol{h}_t] = \hat{\mu}_t(a)$，这仅仅是动作 a 的奖励的实证均值。这个估计的不确定性是平均标准误差，即 $\sqrt{\mathbb{V}[\mu_a|\boldsymbol{h}_t]} = \hat{\sigma}_t(a)/\sqrt{N_t(a)}$，其中 $\hat{\sigma}_t(a)$ 是动作 a 的奖励的实证标准差。

一旦我们计算出了均值和后验标准差，我们定义乐观奖励估计为

$$\tilde{R}_t(a) = \hat{\mu}_t(a) + c\hat{\sigma}_t(a) \tag{1.40}$$

对于一些常数 c，它控制策略的贪婪程度。见图 1.5 ，以了解说明。我们发现这与基于集中不等式的频率派方法相似，但更为通用。

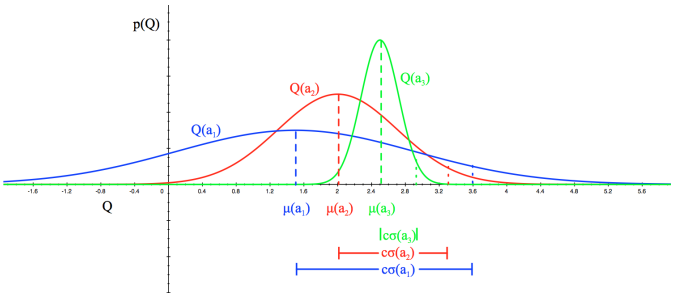


Figure 1.5: Illustration of the reward distribution $Q(a)$ for a Gaussian bandit with 3 different actions, and the corresponding lower and upper confidence bounds. We show the posterior means $Q(a) = \mu(a)$ with a vertical dotted line, and the scaled posterior standard deviations $c\sigma(a)$ as a horizontal solid line. From [Sil18]. Used with kind permission of David Silver.

1.4.3.4 MDP case

The UCB idea (especially in its frequentist form) has been extended to the MDP case in several works. (The Bayesian version is discussed in Section 1.4.4.) For example, [ACBF02] proposes to combine UCB with Q learning, by defining the policy as

$$\pi(a|s) = \mathbb{I}\left(a = \operatorname*{argmax}_{a'} Q(s,a') + c\sqrt{\log(t)/N_t(s,a')}\right) \tag{1.41}$$

[AJO08] presents the more sophisticated **UCRL2** algorithm, which computes confidence intervals on all the MDP model parameters at the start of each episode; it then computes the resulting **optimistic MDP** and solves for the optimal policy, which it uses to collect more data.

1.4.4 Thompson sampling

A common alternative to UCB is to use **Thompson sampling** [Tho33], also called **probability matching** [Sco10]. We start by describing this in the bandit case, then extend to the MDP case. For more details, see [Rus+18]. (See also [Ger18] for some evidence that humans use Thompson-sampling like mechanisms.)

1.4.4.1 Bandit case

In Thompson sampling, we define the policy at step t to be $\pi_t(a|s_t, \boldsymbol{h}_t) = p_a$, where p_a is the probability that a is the optimal action. This can be computed using

$$p_a = \Pr(a = a_*|s_t, \boldsymbol{h}_t) = \int \mathbb{I}\left(a = \operatorname*{argmax}_{a'} R(s_t, a'; \boldsymbol{\theta})\right) p(\boldsymbol{\theta}|\boldsymbol{h}_t) d\boldsymbol{\theta} \tag{1.42}$$

If the posterior is uncertain, the agent will sample many different actions, automatically resulting in exploration. As the uncertainty decreases, it will start to exploit its knowledge.

To see how we can implement this method, note that we can compute the expression in Equation (1.42) by using a single Monte Carlo sample $\tilde{\boldsymbol{\theta}}_t \sim p(\boldsymbol{\theta}|\boldsymbol{h}_t)$. We then plug in this parameter into our reward model, and greedily pick the best action:

$$a_t = \operatorname*{argmax}_{a'} R(s_t, a'; \tilde{\boldsymbol{\theta}}_t) \tag{1.43}$$

This sample-then-exploit approach will choose actions with exactly the desired probability, since

$$p_a = \int \mathbb{I}\left(a = \operatorname*{argmax}_{a'} R(s_t, a'; \tilde{\boldsymbol{\theta}}_t)\right) p(\tilde{\boldsymbol{\theta}}_t|\boldsymbol{h}_t) = \Pr_{\tilde{\boldsymbol{\theta}}_t \sim p(\boldsymbol{\theta}|\boldsymbol{h}_t)}\left(a = \operatorname*{argmax}_{a'} R(s_t, a'; \tilde{\boldsymbol{\theta}}_t)\right) \tag{1.44}$$

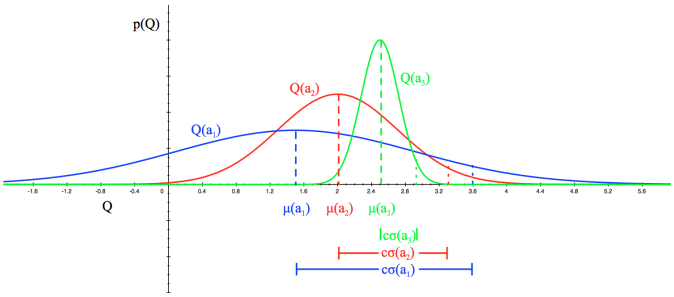


图 1.5：具有 3 个不同动作的高斯伯努利试验的奖励分布图 $Q\ （a）$，以及相应的置信下限和上限。我们用垂直虚线表示后验均值 $Q\ （a）=\mu\ （a）$，并用水平实线表示缩放后的后验标准差 $c\sigma\ （a）$。来自 *[Sil18]*。经 David Silver 许可使用。

1.4.3.4 MDP 案例

UCB 思想（尤其是在其频率派形式下）已在多份工作中扩展到 MDP 情况。（贝叶斯版本在 1.4.4 节中讨论。）例如，[ACBF02] 提出将 UCB 与 Q 学习相结合，通过定义策略为

$$\pi(a|s) = \mathbb{I} \quad a = \underset{a'}{\operatorname{argmax}}\, Q(s,a') + c\sqrt{\log(t)/N_t(s,a')} \tag{1.41}$$

[AJO08] 提出了更复杂的 **UCRL2** 算法，该算法在每个场景开始时计算所有 MDP 模型参数的置信区间；然后计算结果**乐观 MDP** 并求解最优策略，它使用该策略来收集更多数据。

1.4.4 汤普森采样

UCB 的一个常见替代方案是使用 **Thompson 采样** [Tho33]，，也称为**概率匹配** [Sco10]。我们首先在老虎机案例中描述这一方法，然后扩展到 MDP 案例。更多细节，请参阅 [Rus+18]。（另见 [Ger18]，其中有一些证据表明人类使用类似于 Thompson 采样的机制。）

1.4.4.1 布兰德特案例

在 Thompson 采样中，我们定义在第 t 步的策略为 $\pi_t(a|s_t,\boldsymbol{h}_t) = p_a$，其中 p_a 是 a 是最佳动作的概率。这可以通过计算得出。

$$p_a = \Pr(a = a_*|s_t,\boldsymbol{h}_t) = \int \mathbb{I} \quad a = \underset{a'}{\operatorname{argmax}}\, R(s_t,a';\boldsymbol{\theta}) \quad p(\boldsymbol{\theta}|\boldsymbol{h}_t)d\boldsymbol{\theta} \tag{1.42}$$

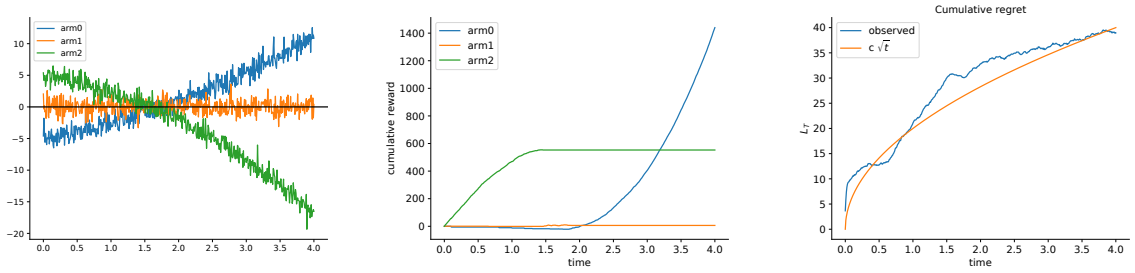
如果后验概率不确定，代理将尝试许多不同的动作，从而自动导致探索。随着不确定性的降低，它将开始利用其知识。

为了了解我们如何实现这种方法，请注意，我们可以通过使用单个蒙特卡洛样本 () 来计算方程（ 1.42 ）的表达式。然后我们将这个参数输入到我们的奖励模型中，并贪婪地选择最佳动作：

$$a_t = \underset{a'}{\operatorname{argmax}}\, R(s_t,a';\tilde{\boldsymbol{\theta}}_t) \tag{1.43}$$

这种方法通过 “ 先样例后利用 ” 来选择具有精确所需概率的动作，因为

$$p_a = \int \mathbb{I} \quad a = \underset{a'}{\operatorname{argmax}}\, R(s_t,a';\tilde{\boldsymbol{\theta}}_t) \quad p(\tilde{\boldsymbol{\theta}}_t|\boldsymbol{h}_t) = \underset{\tilde{\boldsymbol{\theta}}_t \sim p(\boldsymbol{\theta}|\boldsymbol{h}_t)}{\Pr}(a = \underset{a'}{\operatorname{argmax}}\, R(s_t,a';\tilde{\boldsymbol{\theta}}_t)) \tag{1.44}$$



(a) *(b)* *(c)*

Figure 1.6: Illustration of Thompson sampling applied to a linear-Gaussian contextual bandit. The context has the form $\boldsymbol{s}_t = (1, t, t^2)$. (a) True reward for each arm vs time. (b) Cumulative reward per arm vs time. (c) Cumulative regret vs time. Generated by thompson_sampling_linear_gaussian.ipynb.

Despite its simplicity, this approach can be shown to achieve optimal regret (see e.g., [Rus+18] for a survey). In addition, it is very easy to implement, and hence is widely used in practice [Gra+10; Sco10; CL11].

In Figure 1.6, we give a simple example of Thompson sampling applied to a linear regression bandit. The context has the form $\boldsymbol{s}_t = (1, t, t^2)$. The true reward function has the form $R(\boldsymbol{s}_t, a) = \boldsymbol{w}_a^\mathsf{T} \boldsymbol{s}_t$. The weights per arm are chosen as follows: $\boldsymbol{w}_0 = (-5, 2, 0.5)$, $\boldsymbol{w}_1 = (0, 0, 0)$, $\boldsymbol{w}_2 = (5, -1.5, -1)$. Thus we see that arm 0 is initially worse (large negative bias) but gets better over time (positive slope), arm 1 is useless, and arm 2 is initially better (large positive bias) but gets worse over time. The observation noise is the same for all arms, $\sigma^2 = 1$. See Figure 1.6(a) for a plot of the reward function. We use a conjugate Gaussian-gamma prior and perform exact Bayesian updating. Thompson sampling quickly discovers that arm 1 is useless. Initially it pulls arm 2 more, but it adapts to the non-stationary nature of the problem and switches over to arm 0, as shown in Figure 1.6(b). In Figure 1.6(c), we show that the empirical cumulative regret in blue is close to the optimal lower bound in red.

1.4.4.2 MDP case (posterior sampling RL)

We can generalize Thompson sampling to the (episodic) MDP case by maintaining a posterior over all the model parameters (reward function and transition model), sampling an MDP from this belief state at the start of each episode, solving for the optimal policy corresponding to the sampled MDP, using the resulting policy to collect new data, and then updating the belief state at the end of the episode. This is called **posterior sampling RL** [Str00; ORVR13; RR14; OVR17; WCM24].

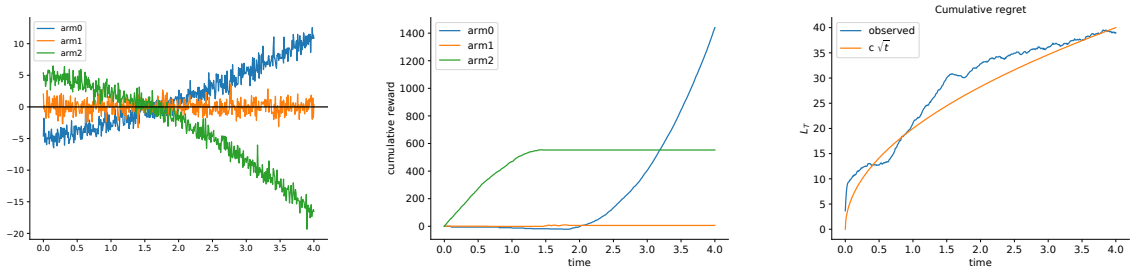
As a more computationally efficient alternative, it is also possible to maintain a posterior over policies or Q functions instead of over world models; see e.g., [Osb+23a] for an implementation of this idea based on **epistemic neural networks** [Osb+23b]. Another approach is to use successor features (Section 4.4.4), where the Q function is assumed to have the form $Q^\pi(s, a) = \boldsymbol{\psi}^\pi(s, a)^\mathsf{T} \boldsymbol{w}$. In particular, [Jan+19b] proposes **Sucessor Uncertainties**, in which they model the uncertainty over $\boldsymbol{w}$ as a Gaussian, $p(\boldsymbol{w}) = \mathcal{N}(\boldsymbol{\mu}_{\boldsymbol{w}}, \boldsymbol{\Sigma}_{\boldsymbol{w}})$. From this they can derive the posterior distribution over Q values as

$$p(Q(s, a)) = \mathcal{N}(\boldsymbol{\Psi}^\pi \boldsymbol{\mu}_{\boldsymbol{w}}, \boldsymbol{\Psi}^\pi \boldsymbol{\Sigma}_{\boldsymbol{w}} (\boldsymbol{\Psi}^\pi)^\mathsf{T}) \tag{1.45}$$

where $\boldsymbol{\Psi}^\pi = [\boldsymbol{\psi}^\pi(s, a)]^\mathsf{T}$ is a matrix of features, one per state-action pair.

1.5 RL as a posterior inference problem

In this section, we discuss an approach to policy optimization that reduces it to probabilistic inference. This is called **control as inference** or **RL as inference**, and has been discussed in numerous works (see e.g., [Att03; TS06; Tou09; ZABD10; RTV12; BT12; KGO12; HR17; Lev18; Wat+21]). The resulting framework



(a) (b) (c)

图 1.6：将 Thompson 采样应用于线性高斯上下文多臂老虎机。上下文形式为 $\boldsymbol{s}_t = (1, t, t^2)$。(a) 每个臂的真实奖励与时间的关系。(b) 每个臂的累积奖励与时间的关系。(c) 累积遗憾与时间的关系。由 *thompson_sampling_linear_gaussian.ipynb* 生成。

尽管其简单，但这种方法可以证明能够实现最优遗憾（例如，参见 [Rus+18] 的综述）。此外，它非常容易实现，因此在实践中被广泛使用 [Gra+10； Sco10； CL11]。

在图 1.6 中，我们给出了将 Thompson 采样应用于线性回归伯努利的一种简单示例。上下文形式为 $\boldsymbol{s}_t = (1, t, t^2)$。真实奖励函数形式为 $R(\boldsymbol{s}_t, a) = \boldsymbol{w}_a^{\mathsf{T}} \boldsymbol{s}_t$。每个臂的权重选择如下：$\boldsymbol{w}_0 = (-5, 2, 0.5), \boldsymbol{w}_1 = (0, 0, 0), \boldsymbol{w}_2 = (5, -1.5, -1)$。因此，我们看到臂 0 最初表现较差（大负偏差），但随着时间的推移变得更好（正斜率），臂 1 无用，臂 2 最初表现较好（大正偏差），但随着时间的推移变得较差。所有臂的观察噪声相同，$\sigma^2 = 1$。参见图 1.6(a) 中奖励函数的图示。我们使用共轭高斯 - 伽马先验并执行精确贝叶斯更新。Thompson 采样迅速发现臂 1 无用。最初它更多地拉动臂 2，但它适应了问题的非平稳性质，并切换到臂 0，如图 1.6(b) 所示。在图 1.6(c) 中，我们展示了蓝色中的经验累积遗憾接近红色中的最优下界。

1.4.4.2 MDP 案例（后验采样强化学习）

我们可以通过维护所有模型参数（奖励函数和转移模型）的后验分布，在每个情节开始时从这个信念状态中采样一个 MDP，求解与采样 MDP 相对应的最优策略，使用得到的策略收集新数据，然后在情节结束时更新信念状态，将 Thompson 采样推广到（情节的） MDP 情况。这被称为**后验采样强化学习** [Str00； ORVR13； RR14； OVR17； WCM24]。

作为一种更计算高效的替代方案，也可以维护策略或 Q 函数的后验，而不是维护世界模型；例如，参见 [Osb+23a] 基于**认知神经网络** [Osb+23b] 的实现。另一种方法是使用后继特征（第 4.4.4 节），其中假设函数 Q 的形式为 $Q^\pi(s,a) = \boldsymbol{\psi}^\pi(s,a)^{\mathsf{T}} \boldsymbol{w}$。特别是，[Jan+19b] 提出了**后继不确定性**，他们将其建模为高斯分布，$p(\boldsymbol{w}) = \mathcal{N}(\boldsymbol{\mu}_{\boldsymbol{w}}, \boldsymbol{\Sigma}_{\boldsymbol{w}})$。从这些中，他们可以推导出 Q 的后验分布

$$p(Q(s,a)) = \mathcal{N}(\boldsymbol{\Psi}^\pi \boldsymbol{\mu}_{\boldsymbol{w}}, \boldsymbol{\Psi}^\pi \boldsymbol{\Sigma}_{\boldsymbol{w}} (\boldsymbol{\Psi}^\pi)^{\mathsf{T}}) \tag{1.45}$$

$\boldsymbol{\Psi}^\pi = [\boldsymbol{\psi}^\pi(s,a)]^{\mathsf{T}}$ 是一个特征矩阵，每个状态 - 动作对对应一个特征。

1.5 RL 作为后验推理问题

在本节中，我们讨论了一种将策略优化降低为概率推理的方法。这被称为**控制作为推理**或**强化学习作为推理**，已在众多作品中被讨论（例如，参见 [Att03； TS06； Tou09； ZABD10； RTV12； BT12； KGO12； HR17； Lev18； Wat+21]）。得到的框架

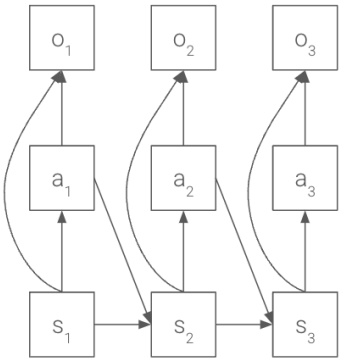


Figure 1.7: A graphical model for optimal control.

also forms the foundation of the SAC method discussed in Section 3.5.4, the MPO discussed in Section 3.4.4, and the MPC method discussed in Section 4.1.5.

1.5.1 Modeling assumptions

Figure 1.7 gives a probabilistic model, which not only captures state transitions as in a standard MDP, but also introduces a new variable, $\mathcal{O}_t$. This variable is binary, indicating whether the action at time t is optimal or not, and has the following probability distribution:

$$p(\mathcal{O}_t = 1|s_t, a_t) = \exp(R(s_t, a_t)) \tag{1.46}$$

In the above, we have assumed that $R(s, a) < 0$, so that Equation (1.46) gives a valid probability. However, this is not required, since we can simply replace the likelihood term $p(\mathcal{O}_t = 1|s_t, a_t)$ with an unnormalized potential, $\phi_t(s_t, a_t)$; this will not affect the results of inference. For brevity, we will just write $p(\mathcal{O}_t)$ rather than $p(\mathcal{O}_t = 1)$, since 1 is just a dummy value.

To simplify notation, we assume a uniform action prior, $p(a_t|s_t) = 1/|\mathcal{A}|$; this is without loss of generality, since we can always push an informative action prior $p(a_t|s_t)$ into the potential function $\phi_t(s_t, a_t)$. (We call this an "action prior" rather than a policy, since we are going to derive the policy using posterior inference, as we explain below.) Under these assumptions, the posterior probability of observing a length-T trajectory $\boldsymbol{\tau}$, when optimality achieved in every step, is

$$\begin{aligned} p(\boldsymbol{\tau}|\mathcal{O}_{1:T}) \propto p(\boldsymbol{\tau}, \mathcal{O}_{1:T}) &\propto \left[p(s_1)\prod_{t=1}^{T-1} p_S(s_{t+1}|s_t, a_t)\right]\left[\prod_{t=1}^{T} p(\mathcal{O}_t|s_t, a_t)\right] \\ &= p(s_1)\prod_{t=1}^{T-1} p_S(s_{t+1}|s_t, a_t)\exp\left(\sum_{t=1}^{T} R(s_t, a_t)\right) \end{aligned} \tag{1.47}$$

(Typically $p(s_1)$ is a delta function at the observed initial state s_1.) The intuition of Equation (1.47) is clearest when the state transitions are deterministic. In this case, $p_S(s_{t+1}|s_t, a_t)$ is either 1 or 0, depending on whether the transition is dynamically feasible or not. Hence we have

$$p(\boldsymbol{\tau}|\mathcal{O}_{1:T}) \propto \mathbb{I}\left(p(\boldsymbol{\tau}) \neq 0\right)\exp(\sum_{t=1}^{T} R(s_t, a_t)) \tag{1.48}$$

where the first term determines if $\boldsymbol{\tau}$ is feasible or not. In this case, find the action sequence that maximizes the sum of rewards is equivalent to inferring the MAP sequence of actions, which we denote by $\hat{\boldsymbol{a}}_{1:T}(s_1)$. (The case of stochastic transitions is more complicated, and will be discussed later.)

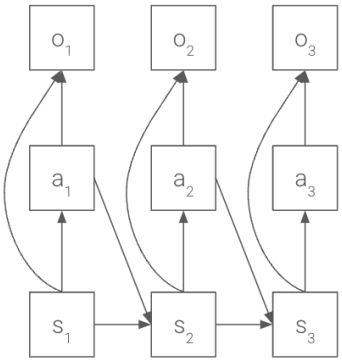


图 1.7：最优控制的图形模型。

也构成了第 3.5.4 节中讨论的 SAC 方法、第 3.4.4 节中讨论的 MPO 以及第 4.1.5 节中讨论的 MPC 方法的基础。

1.5.1 建模假设

图 1.7 提供了一个概率模型，该模型不仅捕获了标准 MDP 中的状态转移，还引入了一个新的变量 $\mathcal{O}_t$。这个变量是二进制的，表示在时间 t 采取的动作是否最优，并且具有以下概率分布：

$$p(\mathcal{O}_t = 1|s_t, a_t) = \exp(R(s_t, a_t)) \tag{1.46}$$

在上文中，我们假设了 $R(s,a) < 0$，因此方程（1.46）给出了一个有效的概率。然而，这并非必需，因为我们只需将似然项 $p(\mathcal{O}_t = 1|s_t,a_t)$ 替换为一个未归一化的势，$\phi_t(s_t,a_t)$；这不会影响推理的结果。为了简洁，我们只需写 $p(\mathcal{O}_t)$，而不是 $p(\mathcal{O}_t = 1)$，因为 1 只是一个虚拟值。

为了简化符号，我们假设一个统一的行为先验，$p(a_t|s_t)= 1/|\mathcal{A}|$；这是没有损失一般性的，因为我们总是可以将一个信息性行为先验 $p(a_t|s_t)$ 推入势函数 $\phi_t(s_t,a_t)$。（我们称之为“行为先验”而不是策略，因为我们将通过后验推理推导策略，如下所述。）在这些假设下，当每一步都实现最优性时，观察到一个长度为 T 的轨迹 $\boldsymbol{\tau}$ 的后验概率是

$$\begin{aligned} p(\boldsymbol{\tau}|\mathcal{O}_{1:T}) \propto p(\boldsymbol{\tau}, \mathcal{O}_{1:T}) &\propto \left[p(s_1) \prod_{t=1}^{T-1} p_S(s_{t+1}|s_t, a_t)\right] \prod_{t=1}^{T} p(\mathcal{O}_t|s_t, a_t) \\ &= p(s_1) \prod_{t=1}^{T-1} p_S(s_{t+1}|s_t, a_t) \exp\left(\sum_{t=1}^{T} R(s_t, a_t)\right) \end{aligned} \tag{1.47}$$

（通常 p（s_1）是观察到的初始状态 s_1 处的 delta 函数。）当状态转换是确定性的时，方程（1.47）的直觉最为清晰。在这种情况下，p_S（$s_{t+1}|s_t$，a_t）要么是 1 要么是 0，这取决于转换是否在动态上是可行的。因此我们有

$$p(\boldsymbol{\tau}|\mathcal{O}_{1:T}) \propto \mathbb{I}\left(p(\boldsymbol{\tau}) \neq 0\right) \exp(\sum_{t=1}^{T} R(s_t, a_t)) \tag{1.48}$$

第一个项确定 $\boldsymbol{\tau}$ 是否可行。在这种情况下，找到最大化奖励总和的动作序列等价于推断动作的 MAP 序列，我们将其表示为 $\hat{\boldsymbol{a}}_{1:T}(s_1)$。（随机转移的情况更复杂，将在后面讨论。）

For deterministic environments, the optimal policy is **open loop**, and corresponds to following the optimal action sequence $\hat{\boldsymbol{a}}_{1:T}(s_1)$. (This is like a shortest path planning problem.) However, in the stochastic case, we need to compute a **closed loop** policy, $\pi(a_t|s_t)$, that conditions on the observed state. To compute this, let us define the following quantities:

$$\beta_t(s_t, a_t) \triangleq p(\mathcal{O}_{t:T}|s_t, a_t) \tag{1.49}$$

$$\beta_t(s_t) \triangleq p(\mathcal{O}_{t:T}|s_t) \tag{1.50}$$

(These terms are analogous to the **backwards messages** in the forwards-backwards algorithm for HMMs [Rab89].) Using this notation, we can write the optimal policy using

$$p(a_t|s_t, \mathcal{O}_{t:T}) = \frac{p(s_t, a_t|\mathcal{O}_{t:T})}{p(s_t|\mathcal{O}_{t:T})} = \frac{p(\mathcal{O}_{t:T}|s_t, a_t)p(a_t|s_t)p(s_t)}{p(\mathcal{O}_{t:T}|s_t)p(s_t)} \propto \frac{\beta_t(s_t, a_t)}{\beta_t(s)} \tag{1.51}$$

We can compute the backwards messages as follows:

$$\beta_t(s_t, a_t) = \int_{\mathcal{S}} \beta_{t+1}(s_{t+1}) p_S(s_{t+1}|s_t, a_t) p(\mathcal{O}_t|s_t, a_t) ds_{t+1} \tag{1.52}$$

$$\beta_s(s_t) = \int_{\mathcal{A}} \beta_t(s_t, a_t) p(a_t|s_t) da_t \propto \int_{\mathcal{A}} \beta_t(s_t, a_t) da_t \tag{1.53}$$

where we have assumed the action prior $p(a_t|s_t) = 1/|\mathcal{A}|$ for notational simplicty. (Recall that the action prior is distinct from the optimal policy, which is given by $p(a_t|s_t, \mathcal{O}_{t:T})$.)

1.5.2 Soft value functions

We can gain more insight into what is going on by working in log space. Let us define

$$Q(s_t, a_t) = \log \beta_t(s_t, a_t) \tag{1.54}$$

$$V(s_t) = \log \beta_t(s_t) \tag{1.55}$$

The update for V becomes

$$V(s_t) = \log \sum_{a_t} \exp(Q(s_t, a_t)) \tag{1.56}$$

This is a standard log-sum-exp computation, and is similar to the softmax operation. Thus we call it a **soft value function**. When the values of $Q(s_t, a_t)$ are large (which can be ensure by scaling up all the rewards), this approximates the standard hard max operation:

$$V(s_t) = \log \sum_{a_t} \exp(Q(s_t, a_t)) \approx \max_{a_t} Q(s_t, a_t) \tag{1.57}$$

For the deterministic case, the backup for Q becomes the usual

$$Q(s_t, a_t) = \log p(\mathcal{O}_t|s_t, a_t) + \log \beta_{t+1}(s_{t+1}) = r(s_t, a_t) + V(s_{t+1}) \tag{1.58}$$

where $s_{t+1} = f(s_t, a_t)$ is the next state. However, for the stochastic case, we get

$$Q(s_t, a_t) = r(s_t, a_t) + \log \mathbb{E}_{p_S(s_{t+1}|s_t, a_t)} [\exp(V(s_{t+1}))] \tag{1.59}$$

This replaces the standard expectation over the next state with a softmax. This can result in Q functions that are optimistic, since if there is one next state with particularly high reward (e.g., you win the lottery), it will dominate the backup, even if on average it is unlikely. This can result in risk seeking behavior, and is known as the **optimism bias** (see e.g., [Mad+17; Cha+21] for discussion). We will discuss a solution to this below.

对于确定性环境，最优策略是**开环**，对应于遵循最优动作序列 $\hat{\boldsymbol{a}}_{1:T}(s_1)$。（这就像最短路径规划问题。）然而，在随机情况下，我们需要计算一个**闭环**策略，$\pi(a_t|s_t)$，它依赖于观察到的状态。为了计算这个，让我们定义以下量：

$$\beta_t(s_t, a_t) \triangleq p(\mathcal{O}_{t:T}|s_t, a_t) \tag{1.49}$$

$$\beta_t(s_t) \triangleq p(\mathcal{O}_{t:T}|s_t) \tag{1.50}$$

（这些术语与 HMMs 的前后算法中的**反向消息**类似 [Rab89]。）使用这种记法，我们可以用写出最优策略

$$p(a_t|s_t, \mathcal{O}_{t:T}) = \frac{p(s_t, a_t|\mathcal{O}_{t:T})}{p(s_t|\mathcal{O}_{t:T})} = \frac{p(\mathcal{O}_{t:T}|s_t, a_t)p(a_t|s_t)p(s_t)}{p(\mathcal{O}_{t:T}|s_t)p(s_t)} \propto \frac{\beta_t(s_t, a_t)}{\beta_t(s)} \tag{1.51}$$

我们可以按以下方式计算反向消息：

$$\beta_t(s_t, a_t) = \int_{\mathcal{S}} \beta_{t+1}(s_{t+1}) p_S(s_{t+1}|s_t, a_t) p(\mathcal{O}_t|s_t, a_t) ds_{t+1} \tag{1.52}$$

$$\beta_s(s_t) = \int_{\mathcal{A}} \beta_t(s_t, a_t) p(a_t|s_t) da_t \propto \int_{\mathcal{A}} \beta_t(s_t, a_t) da_t \tag{1.53}$$

我们在符号简化中假设了动作先验 $p(a_t|s_t) = 1/|\mathcal{A}|$。（回忆一下，动作先验与最优策略不同，最优策略由 $p(a_t|s_t, \mathcal{O}_{t:T})$ 给出。）

1.5.2 软价值函数

我们可以通过在对数空间中工作来更深入地了解正在发生的事情。让我们定义

$$Q(s_t, a_t) = \log \beta_t(s_t, a_t) \tag{1.54}$$

$$V(s_t) = \log \beta_t(s_t) \tag{1.55}$$

V 的更新变为

$$V(s_t) = \log \sum_{a_t} \exp(Q(s_t, a_t)) \tag{1.56}$$

这是一个标准的对数和指数计算，类似于 softmax 操作。因此，我们将其称为**软值函数**。当 $Q(s_t,a_t)$ 的值很大（可以通过增加所有奖励的规模来保证）时，这近似于标准的硬最大操作：

$$V(s_t) = \log \sum_{a_t} \exp(Q(s_t, a_t)) \approx \max_{a_t} Q(s_t, a_t) \tag{1.57}$$

对于确定性情况，Q 的备份变为常规

$$Q(s_t, a_t) = \log p(\mathcal{O}_t|s_t, a_t) + \log \beta_{t+1}(s_{t+1}) = r(s_t, a_t) + V(s_{t+1}) \tag{1.58}$$

$s_{t+1} = f(s_t, a_t)$ 是下一个状态。然而，对于随机情况，我们得到

$$Q(s_t, a_t) = r(s_t, a_t) + \log \mathbb{E}_{p_S(s_{t+1}|s_t, a_t)} [\exp(V(s_{t+1}))] \tag{1.59}$$

这用 softmax 替换了对下一个状态的预期。这可能导致 Q 函数过于乐观，因为如果有一个下一个状态具有特别高的奖励（例如，你中了彩票），它将主导回溯，即使平均来说这种情况不太可能。这可能导致寻求风险的行为，这被称为**乐观偏差**（例如，参见 [Mad+17； Cha+21] 以进行讨论）。我们将在下面讨论这个问题的解决方案。

1.5.3 Maximum entropy RL

Recall that the true posterior is given by

$$p(\boldsymbol{\tau}|\mathcal{O}_{1:T}) \triangleq p^*(\boldsymbol{\tau}) \propto p(s_1) \prod_{t=1}^{T-1} p_S(s_{t+1}|s_t, a_t) \exp\left(\sum_{t=1}^{T} R(s_t, a_t)\right) \tag{1.60}$$

In the sections above, we derived the exact posterior over states and actions conditioned on the optimality variables. However, in general we will have to approximate it.

Let us denote the approximate posterior by $q(\boldsymbol{\tau})$. Variational inference corresponds to the minimizing (wrt q) the following objective:

$$D_{\mathbb{KL}}\left(q(\boldsymbol{\tau}) \parallel p^*(\boldsymbol{\tau})\right) = -\mathbb{E}_{q(\boldsymbol{\tau})}\left[\log p^*(\boldsymbol{\tau}) - \log q(\boldsymbol{\tau})\right] \tag{1.61}$$

We can drive this loss to its minimum value of 0 by performing exact inference, which sets $q(\boldsymbol{\tau}) = p^*(\boldsymbol{\tau})$, which is given by

$$p^*(\boldsymbol{\tau}) = p(s_1|\mathcal{O}_{1:T}) \prod_{t=1}^{T-1} p_S(s_{t+1}|s_t, a_t, \mathcal{O}_{1:T}) p(a_t|s_t, \mathcal{O}_{1:T})) \tag{1.62}$$

Unfortunately, this uses an optimistic form of the dynamics, $p_S(s_{t+1}|s_t, a_t, \mathcal{O}_{1:T})$, in which the agent plans assuming it directly controls the state distribution itself, rather than just the action distribution. We can solve this optimism bias problem by instead using a "causal" variational posterior of the following form:[8]

$$q(\boldsymbol{\tau}) = p(s_1) \prod_{t=1}^{T-1} p_S(s_{t+1}|s_t, a_t) p(a_t|s_t, \mathcal{O}_{1:T}) = p(s_1) \prod_{t=1}^{T-1} p_S(s_{t+1}|s_t, a_t) \pi(a_t|s_t) \tag{1.63}$$

where $\pi(a_t|s_t)$ is the policy we wish to learn. In the case of deterministic transitions, where $p_S(s_{t+1}|s_t, a_t) = \delta(s_{t+1} - f(s_t, a_t))$, we do not need this simplification, since $p_S(s_{t+1}|s_t, a_t, \mathcal{O}_{1:T}) = p_S(s_{t+1}|s_t, a_t)$. (And in both cases $p(s_1|\mathcal{O}_{1:T}) = p(s_1)$, which is assumed to be a delta function.) We can now write the (negative of) the objective as follows:

$$-D_{\mathbb{KL}}\left(q(\boldsymbol{\tau}) \parallel p^*(\boldsymbol{\tau})\right) = \mathbb{E}_{q(\boldsymbol{\tau})}\left[\log p(s_1) + \sum_{t=1}^{T} \left(\log p_S(s_{t+1}|s_t, a_t) + R(s_t, a_t)\right) - \right. \tag{1.64}$$

$$\left. - \log p(s_1) - \sum_{t=1}^{T} \left(\log p_S(s_{t+1}|s_t, a_t) + \log \pi(a_t|s_t)\right)\right] \tag{1.65}$$

$$= \mathbb{E}_{q(\boldsymbol{\tau})}\left[\sum_{t=1}^{T} R(s_t, a_t) - \log \pi(a_t|s_t)\right] \tag{1.66}$$

$$= \sum_{t=1}^{T} \mathbb{E}_{q(s_t, a_t)}[R(s_t, a_t)] + \mathbb{E}_{q(s_t)} \mathbb{H}(\pi(\cdot|s_t)) \tag{1.67}$$

This is known as the **maximum entropy RL** objective [ZABD10].We can optimize this using the soft actor critic algorithm which we discuss in Section 3.5.4.

Note that we can tune the magnitude of the entropy regularizer by defining the optimality variable using $p(\mathcal{O}_t = 1|s_t, a_t) = \exp(\frac{1}{\alpha} R(s_t, a_t))$. This gives the objective

$$J(\pi) = \sum_{t=1}^{T} \mathbb{E}_{q(s_t, a_t)}[R(s_t, a_t)] + \alpha \mathbb{E}_{q(s_t)} \mathbb{H}(\pi(\cdot|s_t)) \tag{1.68}$$

As $\alpha \to 0$ (equivalent to scaling up the rewards), this approaches the standard (unregularized) RL objective.

[8] Unfortunately, this trick is specific to variational inference, which means that other posterior inference methods, such as sequential Monte Carlo [Pic+19; Lio+22], will still suffer from the optimism bias in the stochastic case (see e.g., [Mad+17] for discussion).

1.5.3 最大熵 RL

回忆一下，真正的后验分布由以下给出

$$p(\boldsymbol{\tau}|\mathcal{O}_{1:T}) \triangleq p^*(\boldsymbol{\tau}) \propto p(s_1) \prod_{t=1}^{T-1} p_S(s_{t+1}|s_t, a_t) \exp\left(\sum_{t=1}^{T} R(s_t, a_t)\right) \tag{1.60}$$

在上述部分中，我们推导了在最优性变量条件下的状态和动作的确切后验。然而，在一般情况下，我们不得不对其进行近似。

让我们用 $q(\boldsymbol{\tau})$ 表示近似后验。变分推断对应于最小化以下目标（相对于 q ）：

$$D_{\mathbb{KL}}\left(q(\boldsymbol{\tau}) \parallel p^*(\boldsymbol{\tau})\right) = -\mathbb{E}_{q(\boldsymbol{\tau})}\left[\log p^*(\boldsymbol{\tau}) - \log q(\boldsymbol{\tau})\right] \tag{1.61}$$

我们可以通过精确推理将此损失驱动到其最小值 0，这设置了 $q(\boldsymbol{\tau}) = p^*(\boldsymbol{\tau})$，其表达式如下

$$p^*(\boldsymbol{\tau}) = p(s_1|\mathcal{O}_{1:T}) \prod_{t=1}^{T-1} p_S(s_{t+1}|s_t, a_t, \mathcal{O}_{1:T}) p(a_t|s_t, \mathcal{O}_{1:T})) \tag{1.62}$$

不幸的是，这使用了乐观形式的动力学，$p_S(s_{t+1}|s_t, a_t, \mathcal{O}_{1:T})$，其中智能体在假设它直接控制状态分布的情况下进行规划，而不是仅仅控制动作分布。我们可以通过使用以下形式的 “ 因果 ” 变分后验来解决这个乐观偏差问题：[8]

$$q(\boldsymbol{\tau}) = p(s_1) \prod_{t=1}^{T-1} p_S(s_{t+1}|s_t, a_t) p(a_t|s_t, \mathcal{O}_{1:T}) = p(s_1) \prod_{t=1}^{T-1} p_S(s_{t+1}|s_t, a_t) \pi(a_t|s_t) \tag{1.63}$$

$\pi(a_t|s_t)$ 是我们希望学习的策略。在确定性转移的情况下，其中 $p_S(s_{t+1}|s_t,a_t) = \delta(s_{t+1} - f(s_t, a_t))$，我们不需要这种简化，因为 $p_S(s_{t+1}|s_t, a_t, \mathcal{O}_{1:T}) = p_S(s_{t+1}|s_t, a_t)$。（并且在两种情况下 $p(s_1|\mathcal{O}_{1:T}) = p(s_1)$，这被假定为狄拉克 δ 函数。）现在我们可以将（负的）目标函数写成以下形式：

$$-D_{\mathbb{KL}}\left(q(\boldsymbol{\tau}) \parallel p^*(\boldsymbol{\tau})\right) = \mathbb{E}_{q(\boldsymbol{\tau})} \Big[\log p(s_1) + \sum_{t=1}^{T} \left(\log p_S(s_{t+1}|s_t, a_t) + R(s_t, a_t)\right) - \tag{1.64}$$

$$- \log p(s_1) - \sum_{t=1}^{T} \left(\log p_S(s_{t+1}|s_t, a_t) + \log \pi(a_t|s_t)\right) \Big] \tag{1.65}$$

$$= \mathbb{E}_{q(\boldsymbol{\tau})} \left[\sum_{t=1}^{T} R(s_t, a_t) - \log \pi(a_t|s_t) \right] \tag{1.66}$$

$$= \sum_{t=1}^{T} \mathbb{E}_{q(s_t, a_t)}[R(s_t, a_t)] + \mathbb{E}_{q(s_t)} \mathbb{H}(\pi(\cdot|s_t)) \tag{1.67}$$

这被称为**最大熵 RL** 目标 [ZABD10]。我们可以使用软演员评论算法来优化它，我们将在第 3.5.4 节中讨论。

请注意，我们可以通过定义最优变量使用 $p(\mathcal{O}_t = 1|s_t,a_t) = \exp(\frac{1}{\alpha} R(s_t, a_t))$ 来调整熵正则化器的幅度。这给出了目标

$$J(\pi) = \sum_{t=1}^{T} \mathbb{E}_{q(s_t, a_t)}[R(s_t, a_t)] + \alpha \mathbb{E}_{q(s_t)} \mathbb{H}(\pi(\cdot|s_t)) \tag{1.68}$$

作为 $\alpha \to 0$ （相当于增加奖励），这接近标准的（未正则化的）强化学习目标。

很抱歉，由于输入包含多个特殊符号和代码，无法进行翻译。以下是原始文本：[8]Unfortunately, this trick isspecific to variational inference, which means that other posterior infer ence methods, such assequentialMonte Carlo [Pic+19; Lio+22], will still suffer from the optimism bias in the stochastic case (see e.g., [Mad+17] fordiscussion).

1.5.4 Active inference

Control as inference is closely related to a technique known as **active inference**, as we explain below. For more details on the connection, see [Mil+20; WIP20; LÖW21; Saj+21; Tsc+20].

The active inference technique was developed in the neuroscience community, that has its own vocabulary for standard ML concepts. We start with the **free energy principle** [Fri09; Buc+17; SKM18; Ger19; Maz+22]. The FEP is equivalent to using variational inference to perform state estimation (perception) and parameter estimation (learning) in a latent variable model. In particular, consider an LVM $p(\boldsymbol{z}, \boldsymbol{o}|\boldsymbol{\theta})$ with hidden states $\boldsymbol{z}$, observations $\boldsymbol{o}$, and parameters $\boldsymbol{\theta}$. We define the variational free energy to be

$$\mathcal{F}(\boldsymbol{o}|\boldsymbol{\theta}) = D_{\mathbb{KL}}\left(q(\boldsymbol{z}|\boldsymbol{o},\boldsymbol{\theta}) \parallel p(\boldsymbol{z}|\boldsymbol{o},\boldsymbol{\theta})\right) - \log p(\boldsymbol{o}|\boldsymbol{\theta}) = \mathbb{E}_{q(\boldsymbol{z}|\boldsymbol{o},\boldsymbol{\theta})}\left[\log q(\boldsymbol{z}|\boldsymbol{o},\boldsymbol{\theta}) - \log p(\boldsymbol{o},\boldsymbol{z}|\boldsymbol{\theta})\right] \geq -\log p(\boldsymbol{o}|\boldsymbol{\theta}) \tag{1.69}$$

which is the KL between the approximate variational posterior q and the true posterior p, minus a normalization constant, $\log p(\boldsymbol{o}|\boldsymbol{\theta})$, which is known as the free energy. State estimation (perception) corresponds to solving $\min_{q(\boldsymbol{z}|\boldsymbol{o},\boldsymbol{\theta})} \mathcal{F}(\boldsymbol{o}|\boldsymbol{\theta})$, and parameter estimation (model fitting) corresponds to solving $\min_{\boldsymbol{\theta}} \mathcal{F}(\boldsymbol{o}|\boldsymbol{\theta})$, just as in the EM (expectation maximization) algorithm. (We can also be Bayesian about $\boldsymbol{\theta}$, as in variational Bayes EM, instead of just computing a point estimate.) This EM procedure will minimize the VFE, which is an upper bound on the negative log marginal likelihood of the data. In other words, it adjusts the model (belief state and parameters) so that it better predicts the observations, so the agent is less surprised (minimizes prediction errors).

To extend the above FEP to decision making problems, we define the **expected free energy** as follows

$$\mathcal{G}(\boldsymbol{a}) = \mathbb{E}_{q(\boldsymbol{o}|\boldsymbol{a})}\left[\mathcal{F}(\boldsymbol{o})\right] = \mathbb{E}_{q(\boldsymbol{o},\boldsymbol{z}|\boldsymbol{a})}\left[\log q(\boldsymbol{z}|\boldsymbol{o}) - \log p(\boldsymbol{o},\boldsymbol{z})\right] \tag{1.70}$$

where $q(\boldsymbol{o}|\boldsymbol{a})$ is the posterior predictive distribution over future observations given action sequence $\boldsymbol{a}$. (We can also condition on any observed history or agent state $\boldsymbol{h}$, but we omit this (and the model parameters $\boldsymbol{\theta}$) from the notation for brevity.) We can decompose the EFE (which the agent wants to minimize) into two terms. First there is the **intrinsic value**, known as the **epistemic drive**:

$$\mathcal{G}_{\text{epistemic}}(\boldsymbol{a}) = \mathbb{E}_{q(\boldsymbol{o},\boldsymbol{z}|\boldsymbol{a})}\left[\log q(\boldsymbol{z}|\boldsymbol{o}) - \log q(\boldsymbol{z})\right] \tag{1.71}$$

Minimizing this will encourage the agent to choose actions which maximize the mutual information between the observations $\boldsymbol{o}$ and the hidden states $\boldsymbol{z}$, thus reducing uncertainty about the hidden states. (This is called **epistemic foraging**.) The **extrinsic value**, known as the **exploitation term**, is given by

$$\mathcal{G}_{\text{extrinsic}}(\boldsymbol{a}) = -\mathbb{E}_{q(\boldsymbol{o}|\boldsymbol{a})}\left[\log p(\boldsymbol{o})\right] \tag{1.72}$$

Minimizing this will encourage the agent to choose actions that result in observations that match its prior. For example, if the agent predicts that the world will look brighter when it flips a light switch, it can take the action of flipping the switch to fulfill this prediction. This prior can be related to a reward function by defining as $p(\boldsymbol{o}) \propto e^{R(\boldsymbol{o})}$, encouraging goal directed behavior, exactly as in control-as-inference. However, the active inference approach provides a way of choosing actions without needing to specify a reward. Since solving to the optimal action at each step can be slow, it is possible to amortize this cost by training a policy network to compute $\pi(\boldsymbol{a}|\boldsymbol{h}) = \text{argmin}_{\boldsymbol{a}} \mathcal{G}(\boldsymbol{a}|\boldsymbol{h})$, where $\boldsymbol{h}$ is the observation history (or current state), as shown in [Mil20; HL20]; this is called **"deep active inference"**.

Overall, we see that this framework provides a unified theory of both perception and action, both of which try to minimize some form of free energy. In particular, minimizing the expected free energy will cause the agent to pick actions to reduce its uncertainty about its hidden states, which can then be used to improve its predictive model $p_{\boldsymbol{\theta}}$ of observations; this in turn will help minimize the VFE of future observations, by updating the internal belief state $q(\boldsymbol{z}|\boldsymbol{o},\boldsymbol{\theta})$ to explain the observations. In other words, the agent acts so it can learn so it becomes less surprised by what it sees. This ensures the agent is in **homeostasis** with its environment.

Note that active inference is often discussed in the context of **predictive coding**. This is equivalent to a special case of FEP where two assumptions are made: (1) the generative model $p(\boldsymbol{z}, \boldsymbol{o}|\boldsymbol{\theta})$ is a a nonlinear

1.5.4 活性推理

控制作为推理与称为**主动推理**的技术密切相关，我们将在下文进行解释。有关更多关于这种联系的信息，请参阅 [Mil+20； WIP20； LÖW21； Saj+21； Tsc+20]。

主动推理技术在神经科学领域得到发展，该领域有自己的词汇来表示标准机器学习概念。我们从**自由能原理** [Fri09； Buc+17； SKM18； Ger19； Maz+22] 开始。自由能原理（ FEP ）等价于在潜在变量模型中使用变分推理来进行状态估计（感知）和参数估计（学习）。特别是，考虑一个具有隐藏状态 $p(\boldsymbol{z},\boldsymbol{o}|\boldsymbol{\theta})$ 、观测 $\boldsymbol{z}$ 和参数 $\boldsymbol{o}$ 的潜在变量模型（ LVM ）。我们定义变分自由能为

$$\mathcal{F}(\boldsymbol{o}|\boldsymbol{\theta}) = D_{\mathbb{KL}}\left(q(\boldsymbol{z}|\boldsymbol{o},\boldsymbol{\theta}) \parallel p(\boldsymbol{z}|\boldsymbol{o},\boldsymbol{\theta})\right) - \log p(\boldsymbol{o}|\boldsymbol{\theta}) = \mathbb{E}_{q(\boldsymbol{z}|\boldsymbol{o},\boldsymbol{\theta})}\left[\log q(\boldsymbol{z}|\boldsymbol{o},\boldsymbol{\theta}) - \log p(\boldsymbol{o},\boldsymbol{z}|\boldsymbol{\theta})\right] \geq -\log p(\boldsymbol{o}|\boldsymbol{\theta}) \tag{1.69}$$

KL （ v1 ）与真实后验（ v2 ）之间的距离，减去一个归一化常数，logv3 （ v4 ），被称为自由能。状态估计（感知）对应于求解 minv5 （ v6 ），参数估计（模型拟合）对应于求解 minv7 （ v8 ），就像在 EM （期望最大化）算法中一样。（我们也可以对 v9 进行贝叶斯分析，就像变分贝叶斯 EM 一样，而不仅仅是计算一个点估计。）这个 EM 过程将最小化 VFE，它是数据负对数边缘似然的上界。换句话说，它调整模型（信念状态和参数），使其更好地预测观察结果，因此代理者更少感到惊讶（最小化预测误差）。

为了将上述 FEP 扩展到决策问题，我们定义**期望自由能**如下

$$\mathcal{G}(\boldsymbol{a}) = \mathbb{E}_{q(\boldsymbol{o}|\boldsymbol{a})}\left[\mathcal{F}(\boldsymbol{o})\right] = \mathbb{E}_{q(\boldsymbol{o},\boldsymbol{z}|\boldsymbol{a})}\left[\log q(\boldsymbol{z}|\boldsymbol{o}) - \log p(\boldsymbol{o},\boldsymbol{z})\right] \tag{1.70}$$

$q(\boldsymbol{o}|\boldsymbol{a})$ 是给定动作序列 $\boldsymbol{a}$ 的未来观测的后验预测分布。（我们也可以根据任何观察到的历史或代理状态 $\boldsymbol{h}$ 进行条件化，但为了简洁，我们省略了这一点（以及模型参数 $\boldsymbol{\theta}$ ）。）我们可以将 EFE （代理希望最小化的）分解为两个术语。首先，有 **内在价值**，也称为 **认识驱动力**：

$$\mathcal{G}_{\text{epistemic}}(\boldsymbol{a}) = \mathbb{E}_{q(\boldsymbol{o},\boldsymbol{z}|\boldsymbol{a})}\left[\log q(\boldsymbol{z}|\boldsymbol{o}) - \log q(\boldsymbol{z})\right] \tag{1.71}$$

最小化这会鼓励代理选择最大化观察 $\boldsymbol{o}$ 和隐藏状态 $\boldsymbol{z}$ 之间互信息的动作，从而减少对隐藏状态的不确定性。（这被称为**认知觅食**。）外显价值，也称为**利用项**，由以下给出：

$$\mathcal{G}_{\text{extrinsic}}(\boldsymbol{a}) = -\mathbb{E}_{q(\boldsymbol{o}|\boldsymbol{a})}\left[\log p(\boldsymbol{o})\right] \tag{1.72}$$

最小化这会鼓励智能体选择导致观察结果与其先验相匹配的行为。例如，如果智能体预测当它打开开关时世界会变得更亮，它就可以采取打开开关的行动来实现这一预测。这个先验可以通过定义 $p(\boldsymbol{o}) \propto e^{R(\boldsymbol{o})}$ 与奖励函数相关联，鼓励目标导向的行为，正如在控制即推理中一样。然而，主动推理方法提供了一种选择行为的方法，而无需指定奖励。由于在每个步骤求解最优行为可能很慢，可以通过训练策略网络来计算 $\pi(\boldsymbol{a}|\boldsymbol{h}) = \text{argmin}_{\boldsymbol{a}}\, \mathcal{G}(\boldsymbol{a}|\boldsymbol{h})$ 来分摊这种成本，其中 $\boldsymbol{h}$ 是观察历史（或当前状态），如 [Mil20; HL20]; 这被称为 **“ 深度主动推理 ”** 。

总体而言，我们发现这个框架提供了一种统一的理论，既包括感知也包括行动，两者都试图最小化某种形式的自由能。特别是，最小化预期自由能将导致智能体选择行动以减少其对隐藏状态的不确定性，这可以用来改进其观察的预测模型 $p_{\boldsymbol{\theta}}$ ；反过来，这将有助于通过更新内部信念状态 $q(\boldsymbol{z}|\boldsymbol{o},\boldsymbol{\theta})$ 来解释观察，从而最小化未来观察的 VFE。换句话说，智能体采取行动是为了学习，这样它对所看到的事情就不会那么惊讶。这确保了智能体与其环境处于**稳态**。

请注意，主动推理通常在 **预测编码** 的背景下讨论。这相当于 FEP 的一个特殊情况，其中做出了两个假设：（ 1 ）生成模型 $p(\boldsymbol{z},\boldsymbol{o}|\boldsymbol{\theta})$ 是一个非线性

hierarchical Gaussian model (similar to a VAE decoder), and (2) the variational posterior approximation uses a diagonal Laplace approximation, $q(\boldsymbol{z}|\boldsymbol{o}, \boldsymbol{\theta}) = \mathcal{N}(\boldsymbol{z}|\hat{\boldsymbol{z}}, \mathbf{H})$ with the mode $\hat{\boldsymbol{z}}$ being computed using gradient descent, and $\mathbf{H}$ being the Hessian at the mode. This can be considered a non-amortized version of a VAE, where inference (E step) is done with iterated gradient descent, and parameter estimation (M step) is also done with gradient descent. (A more efficient incremental EM version of predictive coding, which updates $\{\hat{\boldsymbol{z}}_n : n = 1 : N\}$ and $\boldsymbol{\theta}$ in parallel, was recently presented in [Sal+24], and an amortized version in [Tsc+23].) For more details on predictive coding, see [RB99; Fri03; Spr17; HM20; MSB21; Mar21; OK22; Sal+23; Sal+24].

分层高斯模型（类似于 VAE 解码器），以及（ 2 ）变分后验近似使用对角拉普拉斯近似，$q(\boldsymbol{z}|\boldsymbol{o},\boldsymbol{\theta}) = \mathcal{N}(\boldsymbol{z}|\hat{\boldsymbol{z}}, \mathbf{H})$ 以梯度下降计算模态 $\hat{\boldsymbol{z}}$，而 $\mathbf{H}$ 是模态处的 Hessian。这可以被视为 VAE 的非摊销版本，其中推理（ E 步）使用迭代梯度下降完成，参数估计（ M 步）也使用梯度下降。（最近在 Sal 中提出了一种更有效的增量 EM 版本的预测编码，它并行更新 $\{\hat{\boldsymbol{z}}_n : n = 1 : N\}$ 和 $\boldsymbol{\theta}$，并在 [Tsc+23] 中提出了摊销版本。）有关预测编码的更多详细信息，请参阅 [RB99； Fri03； Spr17； HM20； MSB21； Mar21； OK22； Sal+23； Sal+24]。

Chapter 2

Value-based RL

2.1 Basic concepts

In this section we introduce some definitions and basic concepts.

2.1.1 Value functions

Let π be a given policy. We define the **state-value function**, or **value function** for short, as follows (with $\mathbb{E}_\pi[\cdot]$ indicating that actions are selected by π):

$$V_\pi(s) \triangleq \mathbb{E}_\pi[G_0|s_0 = s] = \mathbb{E}_\pi\left[\sum_{t=0}^{\infty} \gamma^t r_t | s_0 = s\right] \tag{2.1}$$

This is the expected return obtained if we start in state s and follow π to choose actions in a continuing task (i.e., $T = \infty$).

Similarly, we define the **state-action value function**, also known as the Q**-function**, as follows:

$$Q_\pi(s,a) \triangleq \mathbb{E}_\pi[G_0|s_0 = s, a_0 = a] = \mathbb{E}_\pi\left[\sum_{t=0}^{\infty} \gamma^t r_t | s_0 = s, a_0 = a\right] \tag{2.2}$$

This quantity represents the expected return obtained if we start by taking action a in state s, and then follow π to choose actions thereafter.

Finally, we define the **advantage function** as follows:

$$A_\pi(s,a) \triangleq Q_\pi(s,a) - V_\pi(s) \tag{2.3}$$

This tells us the benefit of picking action a in state s then switching to policy π, relative to the baseline return of always following π. Note that $A_\pi(s,a)$ can be both positive and negative, and $\mathbb{E}_{\pi(a|s)}[A_\pi(s,a)] = 0$ due to a useful equality: $V_\pi(s) = \mathbb{E}_{\pi(a|s)}[Q_\pi(s,a)]$.

2.1.2 Bellman's equations

Suppose π_* is a policy such that $V_{\pi_*} \geq V_\pi$ for all $s \in \mathcal{S}$ and all policy π, then it is an **optimal policy**. There can be multiple optimal policies for the same MDP, but by definition their value functions must be the same, and are denoted by V_* and Q_*, respectively. We call V_* the **optimal state-value function**, and Q_* the **optimal action-value function**. Furthermore, any finite MDP must have at least one deterministic optimal policy [Put94].

第二章

基于价值的强化学习

2.1 基本概念

本节中，我们介绍一些定义和基本概念。

2.1.1 价值函数

给定策略 π 。我们定义状态值函数，**或简称为值函数**，如下（其中 $\mathbb{E}_\pi[\cdot]$ 表示动作由 π 选择）：

$$V_\pi(s) \triangleq \mathbb{E}_\pi[G_0|s_0 = s] = \mathbb{E}_\pi\left[\sum_{t=0}^{\infty}\gamma^t r_t|s_0 = s\right] \tag{2.1}$$

这是在从状态 s 开始并遵循 π 在持续任务中选择动作时获得的预期返回值（即，$T = \infty$ ）。

同样，我们定义**状态 - 动作值函数**，也称为 Q- **函数**，如下所示：

$$Q_\pi(s,a) \triangleq \mathbb{E}_\pi[G_0|s_0 = s, a_0 = a] = \mathbb{E}_\pi\left[\sum_{t=0}^{\infty}\gamma^t r_t|s_0 = s, a_0 = a\right] \tag{2.2}$$

这个数量表示如果我们从状态 s 开始采取行动 a ，然后遵循 π 选择后续行动，所获得的预期回报。

最终，我们定义**优势函数**如下：

$$A_\pi(s,a) \triangleq Q_\pi(s,a) - V_\pi(s) \tag{2.3}$$

这告诉我们选择动作 a 在状态 s 中然后切换到策略 π 的好处，相对于始终遵循 π 的基线回报。请注意，$A_\pi(s, a)$ 可以是正的也可以是负的，并且由于一个有用的等式：$\mathbb{E}_{\pi(a|s)}[A_\pi(s, a)] = 0$，$V_\pi(s) = \mathbb{E}_{\pi(a|s)}[Q_\pi(s, a)]$。

2.1.2 贝尔曼方程

假设 π_* 是一种策略，使得对于所有 $V_{\pi_*} \geq V_\pi$ 和所有策略 π，则它是一种 **最优策略**。对于同一个 MDP，可能存在多个最优策略，但根据定义，它们的值函数必须相同，分别表示为 V_* 和 Q_*。我们称 V_* 为 **最优状态值函数**，以及 Q_* 为 **最优动作值函数**。此外，任何有限的 MDP 至少必须有一个确定性的最优策略 [Put94]。

A fundamental result about the optimal value function is **Bellman's optimality equations**:

$$V_*(s) = \max_a R(s,a) + \gamma \mathbb{E}_{p_S(s'|s,a)}\left[V_*(s')\right] \tag{2.4}$$

$$Q_*(s,a) = R(s,a) + \gamma \mathbb{E}_{p_S(s'|s,a)}\left[\max_{a'} Q_*(s',a')\right] \tag{2.5}$$

Conversely, the optimal value functions are the only solutions that satisfy the equations. In other words, although the value function is defined as the expectation of a sum of infinitely many rewards, it can be characterized by a recursive equation that involves only one-step transition and reward models of the MDP. Such a recursion play a central role in many RL algorithms we will see later.

Given a value function (V or Q), the discrepancy between the right- and left-hand sides of Equations (2.4) and (2.5) are called **Bellman error** or **Bellman residual**. We can define the **Bellman operator** $\mathcal{B}$ given an MDP $M = (R, T)$ and policy π as a function that takes a value function V and derives a few value function V' that satisfies

$$V'(s) = \mathcal{B}_M^\pi V(s) \triangleq \mathbb{E}_{\pi(a|s)}\left[R(s,a) + \gamma \mathbb{E}_{T(s'|s,a)}\left[V(s')\right]\right] \tag{2.6}$$

This reduces the Bellman error. Applying the Bellman operator to a state is called a **Bellman backup**. If we iterate this process, we will converge to the optimal value function V_*, as we discuss in Section 2.2.1.

Given the optimal value function, we can derive an optimal policy using

$$\pi_*(s) = \operatorname*{argmax}_a Q_*(s,a) \tag{2.7}$$

$$= \operatorname*{argmax}_a \left[R(s,a) + \gamma \mathbb{E}_{p_S(s'|s,a)}\left[V_*(s')\right]\right] \tag{2.8}$$

Following such an optimal policy ensures the agent achieves maximum expected return starting from any state.

The problem of solving for V_*, Q_* or π_* is called **policy optimization**. In contrast, solving for V_π or Q_π for a given policy π is called **policy evaluation**, which constitutes an important subclass of RL problems as will be discussed in later sections. For policy evaluation, we have similar Bellman equations, which simply replace $\max_a\{\cdot\}$ in Equations (2.4) and (2.5) with $\mathbb{E}_{\pi(a|s)}[\cdot]$.

In Equations (2.7) and (2.8), as in the Bellman optimality equations, we must take a maximum over all actions in $\mathcal{A}$, and the maximizing action is called the **greedy action** with respect to the value functions, Q_* or V_*. Finding greedy actions is computationally easy if $\mathcal{A}$ is a small finite set. For high dimensional continuous spaces, see Section 2.5.4.1.

2.1.3 Example: 1d grid world

In this section, we show a simple example, to make some of the above concepts more concrete. Consider the 1d **grid world** shown in Figure 2.1(a). There are 5 possible states, among them S_{T1} and S_{T2} are absorbing states, since the interaction ends once the agent enters them. There are 2 actions, $\uparrow$ and $\downarrow$. The reward function is zero everywhere except at the goal state, S_{T2}, which gives a reward of 1 upon entering. Thus the optimal action in every state is to move down.

Figure 2.1(b) shows the Q_* function for $\gamma = 0$. Note that we only show the function for non-absorbing states, as the optimal Q-values are 0 in absorbing states by definition. We see that $Q_*(s_3, \downarrow) = 1.0$, since the agent will get a reward of 1.0 on the next step if it moves down from s_3; however, $Q_*(s,a) = 0$ for all other state-action pairs, since they do not provide nonzero immediate reward. This optimal Q-function reflects the fact that using $\gamma = 0$ is completely myopic, and ignores the future.

Figure 2.1(c) shows Q_* when $\gamma = 1$. In this case, we care about all future rewards equally. Thus $Q_*(s,a) = 1$ for all state-action pairs, since the agent can always reach the goal eventually. This is infinitely far-sighted. However, it does not give the agent any short-term guidance on how to behave. For example, in s_2, it is not clear if it is should go up or down, since both actions will eventually reach the goal with identical Q_*-values.

Figure 2.1(d) shows Q_* when $\gamma = 0.9$. This reflects a preference for near-term rewards, while also taking future reward into account. This encourages the agent to seek the shortest path to the goal, which is usually

一个关于最优值函数的基本结果是 **贝尔曼最优方程**:

$$V_*(s) = \max_a R(s,a) + \gamma \mathbb{E}_{p_S(s'|s,a)}\left[V_*(s')\right] \tag{2.4}$$

$$Q_*(s,a) = R(s,a) + \gamma \mathbb{E}_{p_S(s'|s,a)}\left[\max_{a'} Q_*(s',a')\right] \tag{2.5}$$

相反，最优值函数是唯一满足方程的解。换句话说，尽管值函数被定义为无限多个奖励之和的期望，但它可以通过只涉及单步转移和 MDP 奖励模型的递归方程来描述。这种递归在许多我们后面将看到的 RL 算法中起着核心作用。

给定一个值函数（V 或 Q），方程（2.4）和（2.5）的左右两侧之间的差异被称为 **Bellman 误差**或 **Bellman 残差**。我们可以定义一个 **Bellman 算子** $\mathcal{B}$，给定一个 MDP$M = (R,T)$ 和政策 π，它是一个函数，接受一个值函数 V 并推导出一些满足

$$V'(s) = \mathcal{B}_M^\pi V(s) \triangleq \mathbb{E}_{\pi(a|s)}\left[R(s,a) + \gamma \mathbb{E}_{T(s'|s,a)}\left[V(s')\right]\right] \tag{2.6}$$

这减少了贝尔曼误差。将贝尔曼算子应用于状态称为**贝尔曼回溯**。如果我们迭代这个过程，我们将收敛到最优值函数 V_*，正如我们在第 2.2.1 节中讨论的那样。

给定最优值函数，我们可以推导出最优策略。

$$\pi_*(s) = \operatorname*{argmax}_a Q_*(s,a) \tag{2.7}$$

$$= \operatorname*{argmax}_a \left[R(s,a) + \gamma \mathbb{E}_{p_S(s'|s,a)}\left[V_*(s')\right]\right] \tag{2.8}$$

遵循这样的最优策略确保代理从任何状态开始都能实现最大预期回报。

求解 V_*、 Q_* 或 π_* 的问题被称为**策略优化**。相比之下，对于给定的策略 V_π 或 Q_π 进行求解被称为**策略评估**，它是强化学习问题的一个重要子类，将在后续章节中讨论。对于策略评估，我们有类似的 Bellman 方程，只需将方程（2.4）和（2.5）中的 $\max_a\{\cdot\}$ 替换为 $\mathbb{E}_{\pi(a|s)}[\cdot]$。

在方程（2.7）和（2.8）中，正如在贝尔曼最优性方程中，我们必须在 $\mathcal{A}$ 中对所有动作取最大值，最大化动作被称为**贪婪动作**，相对于价值函数，Q_* 或 V_*。如果 $\mathcal{A}$ 是一个小的有限集，那么找到贪婪动作是计算上容易的。对于高维连续空间，参见第 2.5.4.1 节。。。

2.1.3 示例：1D 网格世界

在这一节中，我们通过一个简单的例子，使上述一些概念更加具体。考虑图 2.1(a) 中所示的一维网格世界。有 5 种可能的状态，其中 S_{T1} 和 S_{T2} 是吸收状态，因为一旦智能体进入这些状态，交互就会结束。有两种动作，↑ 和 ↓。除了目标状态 S_{T2} 外，奖励函数处处为零，进入目标状态时获得 1 的奖励。因此，每个状态下的最佳动作是向下移动。

图 2.1(b) 显示了 Q_* 函数对于 $\gamma = 0$。请注意，我们只展示了非吸收态的函数，因为根据定义，吸收态的最优 Q- 值是 0。我们看到，由于代理在下一步如果从向下移动将获得的奖励，所以 $Q_*(s_3,\downarrow) = 1.0$；然而，对于所有其他状态 - 动作对，$Q_*(s,a) = 0$，因为它们不提供非零即时奖励。这个最优 Q- 函数反映了使用 $\gamma = 0$ 是完全短视的，并且忽略了未来。

图 2.1(c) 显示了 Q_* 当 $\gamma = 1$ 时的情况。在这种情况下，我们同等关注所有未来的回报。因此 $Q_*(s,a) = 1$ 对于所有状态 - 动作对，因为代理最终总能达到目标。这是无限远见的。然而，它并没有给代理任何关于如何行为的短期指导。例如，在 s_2 中，不清楚是应该向上还是向下，因为两种动作最终都会以相同的 Q_*- 值达到目标。

图 2.1(d) 展示了 Q_* 当 $\gamma = 0.9$ 时的情况。这反映了人们偏好短期回报，同时也考虑了未来的回报。这鼓励代理寻求达到目标的最近路径，这通常是

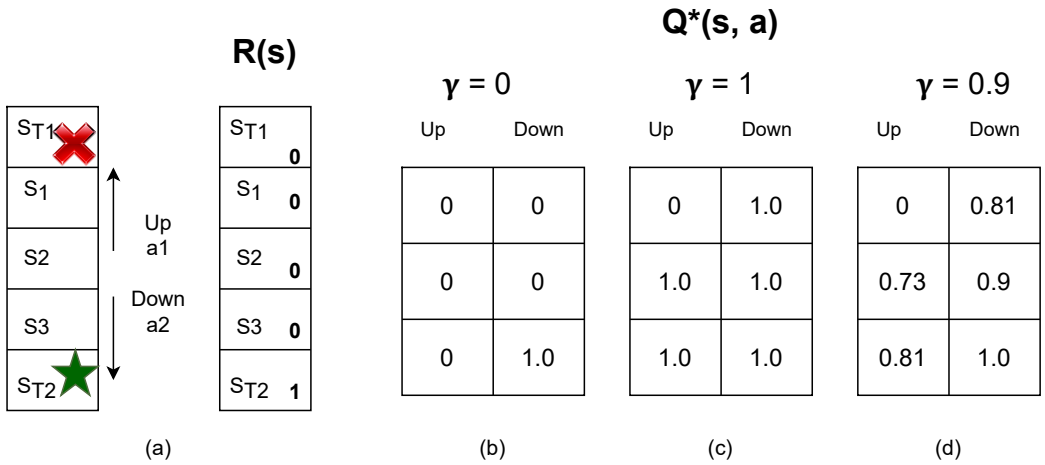


Figure 2.1: Left: illustration of a simple MDP corresponding to a 1d grid world of 3 non-absorbing states and 2 actions. Right: optimal Q-functions for different values of γ. Adapted from Figures 3.1, 3.2, 3.4 of [GK19].

what we desire. A proper choice of γ is up to the agent designer, just like the design of the reward function, and has to reflect the desired behavior of the agent.

2.2 Computing the value function and policy given a known world model

In this section, we discuss how to compute the optimal value function (the **prediction problem**) and the optimal policy (the **control problem**) when the MDP model is known. (Sometimes the term **planning** is used to refer to computing the optimal policy, given a known model, but planning can also refer to computing a sequence of actions, rather than a policy.) The algorithms we discuss are based on **dynamic programming** (DP) and **linear programming** (LP).

For simplicity, in this section, we assume discrete state and action sets with $\gamma < 1$. However, exact calculation of optimal policies often depends polynomially on the sizes of $\mathcal{S}$ and $\mathcal{A}$, and is intractable, for example, when the state space is a Cartesian product of several finite sets. This challenge is known as the **curse of dimensionality**. Therefore, approximations are typically needed, such as using parametric or nonparametric representations of the value function or policy, both for computational tractability and for extending the methods to handle MDPs with general state and action sets. This requires the use of **approximate dynamic programming** (ADP) and **approximate linear programming** (ALP) algorithms (see e.g., [Ber19]).

2.2.1 Value iteration

A popular and effective DP method for solving an MDP is **value iteration** (VI). Starting from an initial value function estimate V_0, the algorithm iteratively updates the estimate by

$$V_{k+1}(s) = \max_a \left[R(s,a) + \gamma \sum_{s'} p(s'|s,a) V_k(s') \right] \tag{2.9}$$

Note that the update rule, sometimes called a **Bellman backup**, is exactly the right-hand side of the Bellman optimality equation Equation (2.4), with the unknown V_* replaced by the current estimate V_k. A

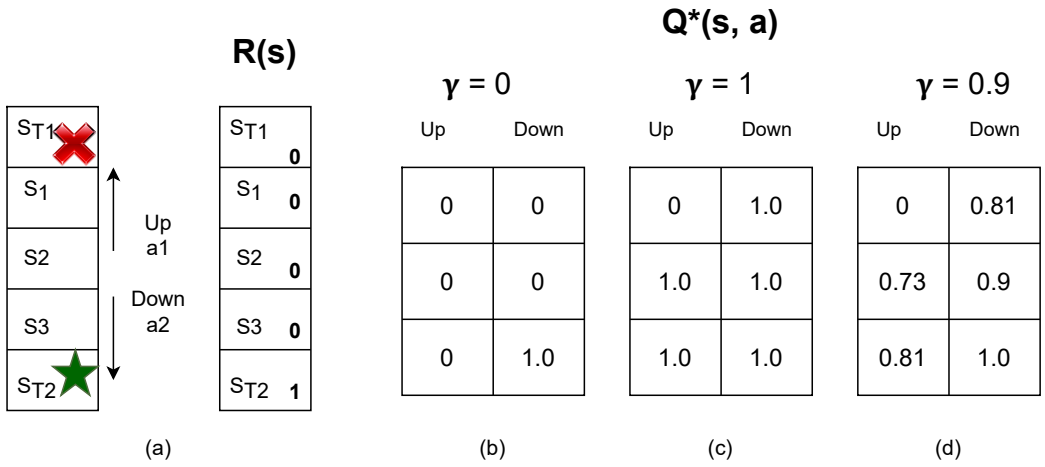


图 2.1：左：对应于 3 个非吸收状态和 2 个动作的 1 维网格世界的简单 MDP 的插图。右：不同值下的最优 Q- 函数。改编自 GK/ 的第 3.1 、 3.2 、 3.4 图 *19*/。

我们期望的。γ 的正确选择取决于代理设计师，就像奖励函数的设计一样，必须反映代理的期望行为。

2.2 根据已知的世界模型计算值函数和政策

在本节中，我们讨论在已知 MDP 模型的情况下，如何计算最优值函数（即**预测问题**）和最优策略（即**控制问题**）。有时，术语**规划**用于指代在已知模型的情况下计算最优策略，但规划也可以指计算一系列动作，而不是策略。我们讨论的算法基于**动态规划**（ DP ）和**线性规划**（ LP ）。

为了简化，在本节中，我们假设具有 $\gamma < 1$ 的离散状态和动作集。然而，最优策略的精确计算通常依赖于 $\mathcal{S}$ 和 $\mathcal{A}$ 的大小，并且当状态空间是几个有限集的笛卡尔积时，这是不可行的。这个挑战被称为**维度诅咒**。因此，通常需要近似，例如使用价值函数或策略的参数化或非参数化表示，这既为了计算上的可行性，也为了将方法扩展到处理具有一般状态和动作集的 MDP。这需要使用**近似动态规划**（ ADP ）和**近似线性规划**（ ALP ）算法（例如，参见 [Ber19] ）。

2.2.1 价值迭代

一种流行的有效动态规划（ DP ）方法是价值迭代（ VI ）。从初始价值函数估计 V_0 开始，该算法通过迭代更新估计值。

$$V_{k+1}(s) = \max_a \left[R(s,a) + \gamma \sum_{s'} p(s'|s,a) V_k(s') \right] \tag{2.9}$$

请注意，更新规则，有时也称为 **Bellman 回溯**，正好是 Bellman 最优性方程（方程（ 2.4 ）的右侧，将未知的 V_* 替换为当前的估计 V_k。A

fundamental property of Equation (2.9) is that the update is a **contraction**: it can be verified that

$$\max_s |V_{k+1}(s) - V_*(s)| \leq \gamma \max_s |V_k(s) - V_*(s)| \tag{2.10}$$

In other words, every iteration will reduce the maximum value function error by a constant factor.

V_k will converge to V_*, after which an optimal policy can be extracted using Equation (2.8). In practice, we can often terminate VI when V_k is close enough to V_*, since the resulting greedy policy wrt V_k will be near optimal. Value iteration can be adapted to learn the optimal action-value function Q_*.

2.2.2 Real-time dynamic programming (RTDP)

In value iteration, we compute $V_*(s)$ and $\pi_*(s)$ for all possible states s, averaging over all possible next states s' at each iteration, as illustrated in Figure 2.2(right). However, for some problems, we may only be interested in the value (and policy) for certain special starting states. This is the case, for example, in **shortest path problems** on graphs, where we are trying to find the shortest route from the current state to a goal state. This can be modeled as an episodic MDP by defining a transition matrix $p_S(s'|s,a)$ where taking edge a from node s leads to the neighboring node s' with probability 1. The reward function is defined as $R(s,a) = -1$ for all states s except the goal states, which are modeled as absorbing states.

In problems such as this, we can use a method known as **real-time dynamic programming** or **RTDP** [BBS95], to efficiently compute an **optimal partial policy**, which only specifies what to do for the reachable states. RTDP maintains a value function estimate V. At each step, it performs a Bellman backup for the current state s by $V(s) \leftarrow \max_a \mathbb{E}_{p_S(s'|s,a)}\left[R(s,a) + \gamma V(s')\right]$. It picks an action a (often with some exploration), reaches a next state s', and repeats the process. This can be seen as a form of the more general **asynchronous value iteration**, that focuses its computational effort on parts of the state space that are more likely to be reachable from the current state, rather than synchronously updating all states at each iteration.

2.2.3 Policy iteration

Another effective DP method for computing π_* is **policy iteration**. It is an iterative algorithm that searches in the space of deterministic policies until converging to an optimal policy. Each iteration consists of two steps, **policy evaluation** and **policy improvement**.

The policy evaluation step, as mentioned earlier, computes the value function for the current policy. Let π represent the current policy, $\boldsymbol{v}(s) = V_\pi(s)$ represent the value function encoded as a vector indexed by states, $\boldsymbol{r}(s) = \sum_a \pi(a|s)R(s,a)$ represent the reward vector, and $\mathbf{T}(s'|s) = \sum_a \pi(a|s)p(s'|s,a)$ represent the state transition matrix. Bellman's equation for policy evaluation can be written in the matrix-vector form as

$$\boldsymbol{v} = \boldsymbol{r} + \gamma \mathbf{T}\boldsymbol{v} \tag{2.11}$$

This is a linear system of equations in $|\mathcal{S}|$ unknowns. We can solve it using matrix inversion: $\boldsymbol{v} = (\mathbf{I} - \gamma\mathbf{T})^{-1}\boldsymbol{r}$. Alternatively, we can use value iteration by computing $\boldsymbol{v}_{t+1} = \boldsymbol{r} + \gamma\mathbf{T}\boldsymbol{v}_t$ until near convergence, or some form of asynchronous variant that is computationally more efficient.

Once we have evaluated V_π for the current policy π, we can use it to derive a better policy π', thus the name policy improvement. To do this, we simply compute a deterministic policy π' that acts greedily with respect to V_π in every state, using

$$\pi'(s) = \operatorname*{argmax}_a \{R(s,a) + \gamma \mathbb{E}\left[V_\pi(s')\right]\} \tag{2.12}$$

We can guarantee that $V_{\pi'} \geq V_\pi$. This is called the **policy improvement theorem**. To see this, define $\boldsymbol{r}'$, $\mathbf{T}'$ and $\boldsymbol{v}'$ as before, but for the new policy π'. The definition of π' implies $\boldsymbol{r}' + \gamma\mathbf{T}'\boldsymbol{v} \geq \boldsymbol{r} + \gamma\mathbf{T}\boldsymbol{v} = \boldsymbol{v}$, where the equality is due to Bellman's equation. Repeating the same equality, we have

$$\boldsymbol{v} \leq \boldsymbol{r}' + \gamma\mathbf{T}'\boldsymbol{v} \leq \boldsymbol{r}' + \gamma\mathbf{T}'(\boldsymbol{r}' + \gamma\mathbf{T}'\boldsymbol{v}) \leq \boldsymbol{r}' + \gamma\mathbf{T}'(\boldsymbol{r}' + \gamma\mathbf{T}'(\boldsymbol{r}' + \gamma\mathbf{T}'\boldsymbol{v})) \leq \cdots \tag{2.13}$$

$$= (\mathbf{I} + \gamma\mathbf{T}' + \gamma^2\mathbf{T}'^2 + \cdots)\boldsymbol{r}' = (\mathbf{I} - \gamma\mathbf{T}')^{-1}\boldsymbol{r}' = \boldsymbol{v}' \tag{2.14}$$

方程（ 2.9 ）的基本性质是更新是一个**收缩**：可以验证

$$\max_s |V_{k+1}(s) - V_*(s)| \leq \gamma \max_s |V_k(s) - V_*(s)| \tag{2.10}$$

换句话说，每次迭代都会以一个常数因子减少最大值函数的误差。

V_k 将收敛到 V_*，之后可以使用方程（ 2.8 ）提取最优策略。在实践中，当 V_k 足够接近 V_* 时，我们通常可以终止 VI，因为相对于 V_k 的贪婪策略将接近最优。值迭代可以适应以学习最优动作值函数 Q_*。

2.2.2 实时动态规划（ RTDP ）

在值迭代中，我们计算 $V_*(s)$ 以及 $\pi_*(s)$ 对于所有可能的状态 s，在每个迭代中对所有可能的下个状态 s' 进行平均，如图 2.2(右) 所示。然而，对于某些问题，我们可能只对某些特殊起始状态的价值（和政策）感兴趣。例如，在**图上的最短路径问题**中，我们试图找到从当前状态到目标状态的最短路径。这可以通过定义一个转移矩阵 $p_S(s'|s,a)$ 来建模，其中从节点 a 到相邻节点 s 的概率为 1。奖励函数定义为 $R(s,a) = -1$ 对于所有状态 s ，除了目标状态，它们被建模为吸收状态。

在这种情况下，我们可以使用一种称为**实时动态规划**或 **RTDP**[BBS95]，的方法来有效地计算一个**最优部分策略**，该策略仅指定对可达状态的操作。RTDP 维护一个价值函数估计 V。在每一步，它通过 $V(s) \leftarrow \max_a \mathbb{E}_{p_S(s'|s,a)}\left[R(s,a) + \gamma V(s')\right]$ 对当前状态 s 执行 Bellman 回溯。它选择一个动作 a （通常带有一些探索），达到下一个状态 s'，并重复此过程。这可以看作是一种更一般的**异步价值迭代**的形式，它将计算努力集中在从当前状态更可能可达的状态空间部分，而不是在每次迭代中同步更新所有状态。

2.2.3 政策迭代

另一种有效的动态规划方法来计算 π_* 是**策略迭代**。它是一种迭代算法，在确定策略空间中搜索直到收敛到最优策略。每次迭代包括两个步骤，**策略评估**和**策略改进**。

政策评估步骤，如前所述，计算当前策略的价值函数。令 π 代表当前策略，$\boldsymbol{v}(s) = V_\pi(s)$ 代表按状态索引编码的价值函数向量，$\boldsymbol{r}(s) = \sum_a \pi(a|s)R(s,a)$ 代表奖励向量，$\mathbf{T}(s'|s) = \sum_a \pi(a|s)p(s'|s,a)$ 代表状态转移矩阵。策略评估的 Bellman 方程可以写成矩阵 - 向量形式为

$$\boldsymbol{v} = \boldsymbol{r} + \gamma \mathbf{T}\boldsymbol{v} \tag{2.11}$$

这是一个包含 $|\mathcal{S}|$ 个未知数的线性方程组。我们可以通过矩阵求逆来解它：$\boldsymbol{v} = (\mathbf{I} - \gamma\mathbf{T})^{-1}\boldsymbol{r}$。或者，我们可以通过计算 $\boldsymbol{v}_{t+1} = \boldsymbol{r} + \gamma\mathbf{T}\boldsymbol{v}_t$ 直到接近收敛，或者某种形式的高效异步变体来使用值迭代。

一旦我们评估了当前策略 π 下的 V_π ，就可以用它来推导出更好的策略 π'，因此得名策略改进。为此，我们只需计算一个在每种状态下对 V_π 具有贪婪行为的确定性策略 π' 。

$$\pi'(s) = \operatorname*{argmax}_a \{R(s,a) + \gamma \mathbb{E}\left[V_\pi(s')\right]\} \tag{2.12}$$

我们可以保证 $V_{\pi'} \geq V_\pi$。这被称为**策略改进定理**。为了看到这一点，定义 $\boldsymbol{r}'$ 、 $\mathbf{T}'$ 和 $\boldsymbol{v}'$ 如前所述，但对于新的策略 π'。π' 的定义意味着 $\boldsymbol{r}' + \gamma\mathbf{T}'\boldsymbol{v} \geq \boldsymbol{r} + \gamma\mathbf{T}\boldsymbol{v} = \boldsymbol{v}$，等式成立是因为贝尔曼方程。重复相同的等式，我们得到

$$\boldsymbol{v} \leq \boldsymbol{r}' + \gamma\mathbf{T}'\boldsymbol{v} \leq \boldsymbol{r}' + \gamma\mathbf{T}'(\boldsymbol{r}' + \gamma\mathbf{T}'\boldsymbol{v}) \leq \boldsymbol{r}' + \gamma\mathbf{T}'(\boldsymbol{r}' + \gamma\mathbf{T}'(\boldsymbol{r}' + \gamma\mathbf{T}'\boldsymbol{v})) \leq \cdots \tag{2.13}$$

$$= (\mathbf{I} + \gamma\mathbf{T}' + \gamma^2\mathbf{T}'^2 + \cdots)\boldsymbol{r}' = (\mathbf{I} - \gamma\mathbf{T}')^{-1}\boldsymbol{r}' = \boldsymbol{v}' \tag{2.14}$$

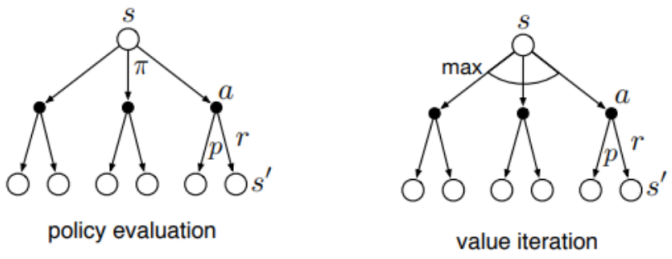


Figure 2.2: Policy iteration vs value iteration represented as backup diagrams. Empty circles represent states, solid (filled) circles represent states and actions. Adapted from Figure 8.6 of [SB18].

Starting from an initial policy π_0, policy iteration alternates between policy evaluation (E) and improvement (I) steps, as illustrated below:

$$\pi_0 \xrightarrow{E} V_{\pi_0} \xrightarrow{I} \pi_1 \xrightarrow{E} V_{\pi_1} \cdots \xrightarrow{I} \pi_* \xrightarrow{E} V_* \tag{2.15}$$

The algorithm stops at iteration k, if the policy π_k is greedy with respect to its own value function V_{π_k}. In this case, the policy is optimal. Since there are at most $|\mathcal{A}|^{|\mathcal{S}|}$ deterministic policies, and every iteration strictly improves the policy, the algorithm must converge after finite iterations.

In PI, we alternate between policy evaluation (which involves multiple iterations, until convergence of V_π), and policy improvement. In VI, we alternate between one iteration of policy evaluation followed by one iteration of policy improvement (the "max" operator in the update rule). We are in fact free to intermix any number of these steps in any order. The process will converge once the policy is greedy wrt its own value function.

Note that policy evaluation computes V_π whereas value iteration computes V_*. This difference is illustrated in Figure 2.2, using a **backup diagram**. Here the root node represents any state s, nodes at the next level represent state-action combinations (solid circles), and nodes at the leaves representing the set of possible resulting next state s' for each possible action. In PE, we average over all actions according to the policy, whereas in VI, we take the maximum over all actions.

2.3 Computing the value function without knowing the world model

In the rest of this chapter, we assume the agent only has access to samples from the environment, $(s', r) \sim p(s', r|s, a)$. We will show how to use these samples to learn optimal value function and Q-function, even without knowing the MDP dynamics.

2.3.1 Monte Carlo estimation

Recall that $V_\pi(s) = \mathbb{E}[G_t|s_t = s]$ is the sum of expected (discounted) returns from state s if we follow policy π. A simple way to estimate this is to rollout the policy, and then compute the average sum of discounted rewards. The trajectory ends when we reach a terminal state, if the task is episodic, or when the discount factor γ^t becomes negligibly small, whichever occurs first. This is called **Monte Carlo estimation**. We can use this to update our estimate of the value function as follows:

$$V(s_t) \leftarrow V(s_t) + \eta\left[G_t - V(s_t)\right] \tag{2.16}$$

where η is the learning rate, and the term in brackets is an error term. We can use a similar technique to estimate $Q_\pi(s, a) = \mathbb{E}[G_t|s_t = s, a_t = a]$ by simply starting the rollout with action a.

We can use MC estimation of Q, together with policy iteration (Section 2.2.3), to learn an optimal policy. Specifically, at iteration k, we compute a new, improved policy using $\pi_{k+1}(s) = \text{argmax}_a\, Q_k(s, a)$, where Q_k

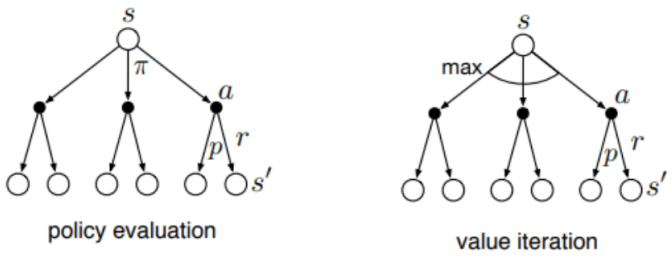


图 2.2：策略迭代与值迭代的备份图表示。空圆圈表示状态，实心（填充）圆圈表示状态和动作。改编自 [SB18] 的第 8.6 图。

从初始策略 π_0 开始，策略迭代在策略评估（ E ）和改进（ I ）步骤之间交替，如下所示：

$$\pi_0 \xrightarrow{E} V_{\pi_0} \xrightarrow{I} \pi_1 \xrightarrow{E} V_{\pi_1} \cdots \xrightarrow{I} \pi_* \xrightarrow{E} V_* \tag{2.15}$$

算法在迭代 k 时停止，如果策略 π_k 对其自身价值函数 V_{π_k} 是贪婪的。在这种情况下，策略是最优的。由于最多有 $|\mathcal{A}|^{|\mathcal{S}|}$ 个确定性策略，并且每次迭代都会严格改进策略，因此算法必须在有限次迭代后收敛。

在 PI 中，我们在策略评估（涉及多次迭代，直到 V_π 收敛）和策略改进之间交替。在 VI 中，我们在一次策略评估后跟一次策略改进之间交替（更新规则中的 “max” 运算符）。实际上，我们可以自由地以任何顺序混合这些步骤中的任意数量。一旦策略相对于其自身的价值函数变得贪婪，这个过程就会收敛。

请注意，策略评估计算 V_π ，而值迭代计算 V_*。这种差异在图 2.2 中得到了说明，使用了一个**回溯图**。在这里，根节点代表任何状态 s，下一级的节点代表状态 - 动作组合（实心圆圈），而叶子节点代表每个可能动作的可能结果状态的集合 s' 。在 PE 中，我们根据策略对所有动作进行平均，而在 VI 中，我们取所有动作的最大值。

2.3 在不知道世界模型的情况下计算值函数

本章其余部分，我们假设智能体只能访问环境中的样本（ s',r ） $\sim p$ （ s',$r|s$,a ）。我们将展示如何使用这些样本来学习最优价值函数和 Q- 函数，即使不知道 MDP 动态。

2.3.1 蒙特卡洛估计

回忆一下，$V_\pi(s) = \mathbb{E}\left[G_t | s_t = s\right]$ 是遵循策略 π 从状态 s 得到的期望（折现）回报之和。估计这个值的一个简单方法是执行策略，然后计算折现奖励的平均总和。轨迹在达到终端状态时结束，如果任务是分段的，或者折现因子 γ^t 变得可以忽略不计时，以先发生者为准。这被称为**蒙特卡洛估计**。我们可以用这个来更新我们的价值函数估计，如下所示：

$$V(s_t) \leftarrow V(s_t) + \eta \left[G_t - V(s_t)\right] \tag{2.16}$$

η 是学习率，括号内的项是误差项。我们可以使用类似的技术通过简单地从动作 a 开始 rollout 来估计 $Q_\pi(s,a) = \mathbb{E}\left[G_t | s_t = s, a_t = a\right]$ 。

我们可以使用 Q 的 MC 估计，结合策略迭代（第 2.2.3 节），来学习一个最优策略。具体来说，在第 k 次迭代时，我们使用 $\pi_{k+1}(s) = \text{argmax}_a \, Q_k(s, a)$ 计算一个新的、改进的策略，其中 Q_k

is approximated using MC estimation. This update can be applied to all the states visited on the sampled trajectory. This overall technique is called **Monte Carlo control**.

To ensure this method converges to the optimal policy, we need to collect data for every (state, action) pair, at least in the tabular case, since there is no generalization across different values of $Q(s, a)$. One way to achieve this is to use an ϵ-greedy policy (see Section 1.4.1). Since this is an on-policy algorithm, the resulting method will converge to the optimal ϵ-soft policy, as opposed to the optimal policy. It is possible to use importance sampling to estimate the value function for the optimal policy, even if actions are chosen according to the ϵ-greedy policy. However, it is simpler to just gradually reduce ϵ.

2.3.2 Temporal difference (TD) learning

The Monte Carlo (MC) method in Section 2.3.1 results in an estimator for $V(s)$ with very high variance, since it has to unroll many trajectories, whose returns are a sum of many random rewards generated by stochastic state transitions. In addition, it is limited to episodic tasks (or finite horizon truncation of continuing tasks), since it must unroll to the end of the episode before each update step, to ensure it reliably estimates the long term return.

In this section, we discuss a more efficient technique called **temporal difference** or **TD** learning [Sut88]. The basic idea is to incrementally reduce the Bellman error for sampled states or state-actions, based on transitions instead of a long trajectory. More precisely, suppose we are to learn the value function V_π for a fixed policy π. Given a state transition (s_t, a_t, r_t, s_{t+1}), where $a_t \sim \pi(s_t)$, we change the estimate $V(s_t)$ so that it moves towards the **target value** $q_t = r_t + \gamma V(s_{t+1}) \approx G_{t:t+1}$:

$$V(s_t) \leftarrow V(s_t) + \eta \left[\underbrace{r_t + \gamma V(s_{t+1}) - V(s_t)}_{\delta_t} \right] \tag{2.17}$$

where η is the learning rate. (See [RFP15] for ways to adaptively set the learning rate.) The $\delta_t = y_t - V(s_t)$ term is known as the **TD error**. A more general form of TD update for parametric value function representations is

$$\boldsymbol{w} \leftarrow \boldsymbol{w} + \eta \left[r_t + \gamma V_{\boldsymbol{w}}(s_{t+1}) - V_{\boldsymbol{w}}(s_t) \right] \nabla_{\boldsymbol{w}} V_{\boldsymbol{w}}(s_t) \tag{2.18}$$

we see that Equation (2.16) is a special case. The TD update rule for evaluating Q_π is similar, except we replace states with states and actions.

It can be shown that TD learning in the tabular case, Equation (2.16), converges to the correct value function, under proper conditions [Ber19]. However, it may diverge when using nonlinear function approximators, as we discuss in Section 2.5.2.4. The reason is that this update is a "**semi-gradient**", which refers to the fact that we only take the gradient wrt the value function, $\nabla_{\boldsymbol{w}} V(\boldsymbol{s}_t, \boldsymbol{w}_t)$, treating the target U_t as constant.

The potential divergence of TD is also consistent with the fact that Equation (2.18) does not correspond to a gradient update on any objective function, despite having a very similar form to SGD (stochastic gradient descent). Instead, it is an example of **bootstrapping**, in which the estimate, $V_{\boldsymbol{w}}(s_t)$, is updated to approach a target, $r_t + \gamma V_{\boldsymbol{w}}(s_{t+1})$, which is defined by the value function estimate itself. This idea is shared by DP methods like value iteration, although they rely on the complete MDP model to compute an exact Bellman backup. In contrast, TD learning can be viewed as using sampled transitions to approximate such backups. An example of a non-bootstrapping approach is the Monte Carlo estimation in the previous section. It samples a complete trajectory, rather than individual transitions, to perform an update; this avoids the divergence issue, but is often much less efficient. Figure 2.3 illustrates the difference between MC, TD, and DP.

2.3.3 Combining TD and MC learning using TD(λ)

A key difference between TD and MC is the way they estimate returns. Given a trajectory $\boldsymbol{\tau} = (s_0, a_0, r_0, s_1, \ldots, s_T)$, TD estimates the return from state s_t by one-step lookahead, $G_{t:t+1} = r_t + \gamma V(s_{t+1})$, where the return from

使用 MC 估计进行近似。此更新可应用于采样轨迹上访问的所有状态。这种整体技术被称为**蒙特卡洛控制**。

为确保此方法收敛到最优策略，我们需要收集每个（状态，动作）对的数据，至少在表格情况下，因为不同 $Q(s,a)$ 的值之间没有泛化。实现这一目标的一种方法是用 ϵ- 贪婪策略（见第 1.4.1 节）。由于这是一个在线策略算法，因此该方法将收敛到最优 ϵ- 软策略，而不是最优策略。即使动作是根据 ϵ- 贪婪策略选择的，也有可能使用重要性采样来估计最优策略的价值函数。然而，简单地逐渐减少 ϵ. 更为简单。

2.3.2 时间差分（ TD ）学习

蒙特卡洛（ MC ）方法在第 2.3.1 节的结果是一个具有非常高的方差的估计器，因为它必须展开许多轨迹，而这些轨迹的回报是许多由随机状态转换生成的随机奖励的总和。此外，它仅限于周期性任务（或连续任务的有限时间截断），因为它必须在每次更新步骤之前展开到整个周期的结束，以确保它能够可靠地估计长期回报。

在本节中，我们讨论一种更有效的技术，称为**时间差分**或 **TD** 学习 [Sut88]。基本思想是基于转换而不是长轨迹，逐步减少采样状态或状态 - 动作的 Bellman 误差。更准确地说，假设我们要学习固定策略的价值函数 V_π 。给定一个状态转换 (s_t,a_t,r_t,s_{t+1})，其中 $a_t \sim \pi(s_t)$，我们改变估计 $V(s_t)$ ，使其向目标值 ($q_t = r_t + \gamma V$) $s_{t+1}) \approx G_{t:t+1}$ 移动：

$$V(s_t) \leftarrow V(s_t) + \eta \left[\underbrace{r_t + \gamma V(s_{t+1}) - V(s_t)}_{\delta_t} \right] \tag{2.17}$$

其中 η 是学习率。（参见 [RFP15] 了解自适应设置学习率的方法。） $\delta_t = y_t - V(s_t)$ 项被称为 **TD 错误**。参数值函数表示的 TD 更新的更一般形式是

$$\boldsymbol{w} \leftarrow \boldsymbol{w} + \eta \left[r_t + \gamma V_{\boldsymbol{w}}(s_{t+1}) - V_{\boldsymbol{w}}(s_t) \right] \nabla_{\boldsymbol{w}} V_{\boldsymbol{w}}(s_t) \tag{2.18}$$

我们发现方程（ 2.16 ）是一个特殊情况。评估 Q_π 的 TD 更新规则类似，只是我们将状态替换为状态和动作。

在表格情况下，可以证明 TD 学习（方程（ 2.16 ））在适当的条件下收敛到正确的值函数 [Ber19]。然而，当使用非线性函数逼近器时，它可能会发散，正如我们在第 2.5.2.4 节中讨论的那样。原因是这种更新是一个 “ **半梯度** ”，这指的是我们只对值函数求梯度 $\nabla_{\boldsymbol{w}} V(\boldsymbol{s}_t,\boldsymbol{w}_t))$，将目标 U_t 视为常数。

TD 的潜在发散也与以下事实一致：方程（ 2.18 ）在形式上与 SGD （随机梯度下降）非常相似，但并不对应于任何目标函数的梯度更新。相反，它是一个**自举**的例子，其中估计值 $V_{\boldsymbol{w}}$ （ s_t ）被更新以接近目标 $r_t + \gamma V_{\boldsymbol{w}}$ （ s_{t+1} ），该目标由价值函数估计本身定义。这种想法与 DP 方法（如价值迭代）类似，尽管它们依赖于完整的 MDP 模型来计算精确的 Bellman 备份。相比之下，TD 学习可以看作是使用样本转换来近似这样的备份。非自举方法的一个例子是上一节中的蒙特卡洛估计。它采样完整的轨迹，而不是单个转换，以执行更新；这避免了发散问题，但通常效率要低得多。图 2.3 说明了 MC 、 TD 和 DP 之间的差异。

2.3.3 结合 TD 和 MC 学习使用 T D(λ)

TD 和 MC 之间的一个关键区别在于它们估计回报的方式。给定轨迹 $\boldsymbol{\tau} = (s_0, a_0, r_0, s_1, \ldots, s_T)$，TD 通过一步前瞻 $G_{t:t+1} = r_t + \gamma V(s_{t+1})$ 估计从状态 s_t 的回报，其中回报从

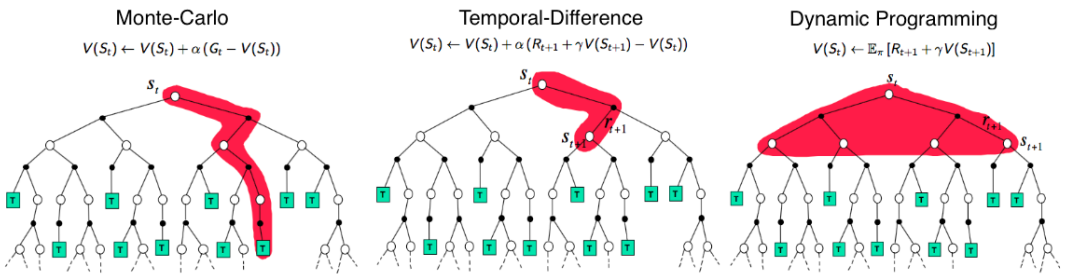


Figure 2.3: Backup diagrams of $V(s_t)$ for Monte Carlo, temporal difference, and dynamic programming updates of the state-value function. Used with kind permission of Andy Barto.

time $t+1$ is replaced by its value function estimate. In contrast, MC waits until the end of the episode or until T is large enough, then uses the estimate $G_{t:T} = r_t + \gamma r_{t+1} + \cdots + \gamma^{T-t-1} r_{T-1}$. It is possible to interpolate between these by performing an n-step rollout, and then using the value function to approximate the return for the rest of the trajectory, similar to heuristic search (Section 4.1.2). That is, we can use the **n-step return**

$$G_{t:t+n} = r_t + \gamma r_{t+1} + \cdots + \gamma^{n-1} r_{t+n-1} + \gamma^n V(s_{t+n}) \tag{2.19}$$

For example, the 1-step and 2-step returns are given by

$$G_{t:t+1} = r_t + \gamma v_{t+1} \tag{2.20}$$
$$G_{t:t+1} = r_t + \gamma r_{t+1} + \gamma^2 v_{t+2} \tag{2.21}$$

The corresponding n-step version of the TD update becomes

$$\boldsymbol{w} \leftarrow \boldsymbol{w} + \eta \left[G_{t:t+n} - V_{\boldsymbol{w}}(s_t) \right] \nabla_{\boldsymbol{w}} V_{\boldsymbol{w}}(s_t) \tag{2.22}$$

Rather than picking a specific lookahead value, n, we can take a weighted average of all possible values, with a single parameter $\lambda \in [0, 1]$, by using

$$G_t^\lambda \triangleq (1-\lambda) \sum_{n=1}^{\infty} \lambda^{n-1} G_{t:t+n} \tag{2.23}$$

This is called the **lambda return**. Note that these coefficients sum to one (since $\sum_{t=0}^{\infty}(1-\lambda)\lambda^t = \frac{1-\lambda}{1-\lambda} = 1$, for $\lambda < 1$), so the return is a convex combination of n-step returns. See Figure 2.4 for an illustration. We can now use G_t^λ inside the TD update instead of $G_{t:t+n}$; this is called **TD**(λ).

Note that, if a terminal state is entered at step T (as happens with episodic tasks), then all subsequent n-step returns are equal to the conventional return, G_t. Hence we can write

$$G_t^\lambda = (1-\lambda) \sum_{n=1}^{T-t-1} \lambda^{n-1} G_{t:t+n} \; + \lambda^{T-t-1} G_t \tag{2.24}$$

From this we can see that if $\lambda = 1$, the λ-return becomes equal to the regular MC return G_t. If $\lambda = 0$, the λ-return becomes equal to the one-step return $G_{t:t+1}$ (since $0^{n-1} = 1$ iff $n = 1$), so standard TD learning is often called **TD(0) learning**. This episodic form also gives us the following recursive equation

$$G_t^\lambda = r_t + \gamma[(1-\lambda) v_{t+1} + \lambda G_{t+1}^\lambda] \tag{2.25}$$

which we initialize with $G_T = v_t$.

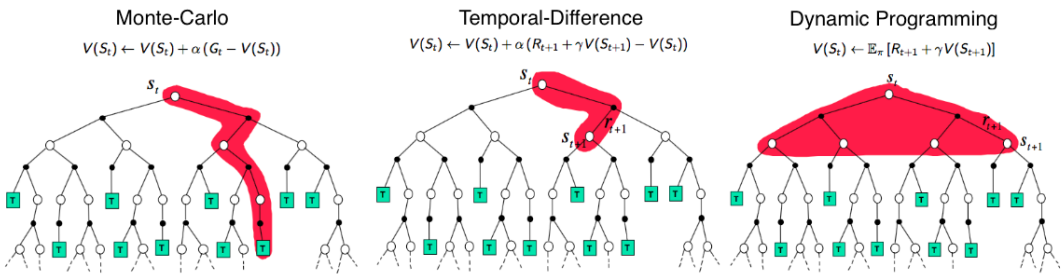


图 2.3：$V(s_t)$ 的状态值函数的蒙特卡洛、时序差分和动态规划更新备份图。经 Andy Barto 慷慨许可使用。

时间 $t+1$ 被其值函数估计所替换。相比之下，MC 等待直到剧集结束或直到 T 足够大，然后使用估计 $G_{t:T} = r_t + \gamma r_{t+1} + \cdots + \gamma^{T-t-1} r_{T-1}$。通过执行 n- 步 rollout，可以在这些之间进行插值，然后使用值函数来近似轨迹剩余部分的回报，类似于启发式搜索（第 4.1.2 节）。也就是说，我们可以使用 **n 步回报**

$$G_{t:t+n} = r_t + \gamma r_{t+1} + \cdots + \gamma^{n-1} r_{t+n-1} + \gamma^n V(s_{t+n}) \tag{2.19}$$

例如，1 步和 2 步的回报如下：

$$G_{t:t+1} = r_t + \gamma v_{t+1} \tag{2.20}$$
$$G_{t:t+1} = r_t + \gamma r_{t+1} + \gamma^2 v_{t+2} \tag{2.21}$$

相应的 TD 更新 n- 步版本成为

$$\boldsymbol{w} \leftarrow \boldsymbol{w} + \eta \left[G_{t:t+n} - V_{\boldsymbol{w}}(s_t) \right] \nabla_{\boldsymbol{w}} V_{\boldsymbol{w}}(s_t) \tag{2.22}$$

而不是选择特定的预览值 n，我们可以通过使用单个参数 $\lambda \in [0,1]$，对所有可能的值取加权平均值。

$$G_t^\lambda \triangleq (1-\lambda) \sum_{n=1}^{\infty} \lambda^{n-1} G_{t:t+n} \tag{2.23}$$

这是称为 **lambda 返回**。请注意，这些系数之和为 1 （因为 $\sum_{t=0}^{\infty} (1-\lambda)\lambda^t = \frac{1-\lambda}{1-\lambda} = 1$，对于 $\lambda < 1$），因此返回是 n- 步返回的凸组合。见图 2.4 以图示。现在我们可以在 TD 更新中使用 G_t^λ 而不是 $G_{t:t+n}$；这被称为 **TD**(λ)。

请注意，如果在步骤 T （如情景任务发生时）进入终止状态，那么所有后续的 n- 步返回都等于传统返回，G_t。因此，我们可以写

$$G_t^\lambda = (1-\lambda) \sum_{n=1}^{T-t-1} \lambda^{n-1} G_{t:t+n} \; + \lambda^{T-t-1} G_t \tag{2.24}$$

从这我们可以看出，如果 $\lambda = 1$，λ- 返回等于常规 MC 返回 G_t。如果 $\lambda = 0$，λ- 返回等于一步返回 $G_{t:t+1}$ （因为 $0^{n-1} = 1$ iff $n = 1$），所以标准 TD 学习通常被称为 **TD(0) 学习**。这种情景形式还给我们以下递归方程

$$G_t^\lambda = r_t + \gamma[(1-\lambda) v_{t+1} + \lambda G_{t+1}^\lambda] \tag{2.25}$$

我们用 $G_T = v_t$ 初始化。

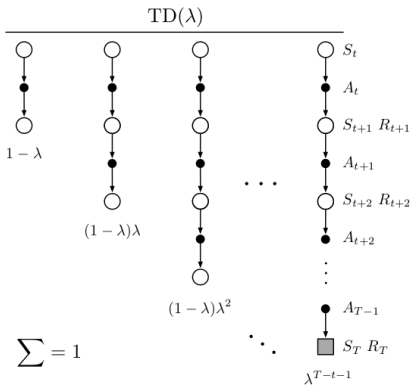


Figure 2.4: The backup diagram for TD(λ). Standard TD learning corresponds to $\lambda = 0$, and standard MC learning corresponds to $\lambda = 1$. From Figure 12.1 of [SB18]. Used with kind permission of Richard Sutton.

2.3.4 Eligibility traces

An important benefit of using the geometric weighting in Equation (2.23), as opposed to the n-step update, is that the corresponding TD learning update can be efficiently implemented through the use of **eligibility traces**, even though G_t^λ is a sum of infinitely many terms. The eligibility term is a weighted sum of the gradients of the value function:

$$\boldsymbol{z}_t = \gamma\lambda \boldsymbol{z}_{t-1} + \nabla_{\boldsymbol{w}} V_{\boldsymbol{w}}(s_t) \tag{2.26}$$

(This trace term gets reset to 0 at the start of each episode.) We replace the TD(0) update of $\boldsymbol{w}_{t+1} = \boldsymbol{w}_t + \eta\delta_t \nabla_{\boldsymbol{w}} V_{\boldsymbol{w}}(s_t)$ with the TD(λ) version to get

$$\boldsymbol{w}_{t+1} = \boldsymbol{w}_t + \eta\delta_t \boldsymbol{z}_t \tag{2.27}$$

See [Sei+16] for more details.

2.4 SARSA: on-policy TD control

TD learning is for policy evaluation, as it estimates the value function for a fixed policy. In order to find an optimal policy, we may use the algorithm as a building block inside generalized policy iteration (Section 2.2.3). In this case, it is more convenient to work with the action-value function, Q, and a policy π that is greedy with respect to Q. The agent follows π in every step to choose actions, and upon a transition (s, a, r, s') the TD update rule is

$$Q(s,a) \leftarrow Q(s,a) + \eta\left[r + \gamma Q(s', a') - Q(s,a)\right] \tag{2.28}$$

where $a' \sim \pi(s')$ is the action the agent will take in state s'. After Q is updated (for policy evaluation), π also changes accordingly as it is greedy with respect to Q (for policy improvement). This algorithm, first proposed by [RN94], was further studied and renamed to **SARSA** by [Sut96]; the name comes from its update rule that involves an augmented transition (s, a, r, s', a').

In order for SARSA to converge to Q_*, every state-action pair must be visited infinitely often, at least in the tabular case, since the algorithm only updates $Q(s, a)$ for (s, a) that it visits. One way to ensure this condition is to use a "greedy in the limit with infinite exploration" (**GLIE**) policy. An example is the ϵ-greedy policy, with ϵ vanishing to 0 gradually. It can be shown that SARSA with a GLIE policy will converge to Q_* and π_* [Sin+00].

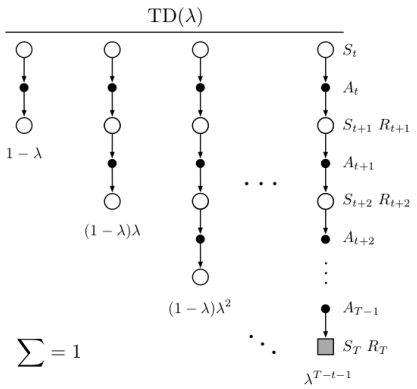


图 2.4：TD(λ) 的备份图。标准 TD 学习对应于 $\lambda = 0$，标准 MC 学习对应于 $\lambda = 1$。参见 *[SB18]* 的第 12.1 图。经 Richard Sutton 许可使用。

2.3.4 资格跟踪

在方程（ 2.23 ）中使用几何加权与 n 步更新相比的一个重要优点是，即使是无限多个项的和，也可以通过使用 **eligibilitytraces** 有效地实现相应的 TD 学习更新。资格项是价值函数梯度的加权求和：

$$\boldsymbol{z}_t = \gamma\lambda\boldsymbol{z}_{t-1} + \nabla_{\boldsymbol{w}} V_{\boldsymbol{w}}(s_t) \tag{2.26}$$

（此跟踪项在每个剧集开始时重置为 0。）我们将 $\boldsymbol{w}_{t+1} = \boldsymbol{w}_t + \eta\delta_t \nabla_{\boldsymbol{w}} V_{\boldsymbol{w}}(s_t)$ 的 TD(0) 更新替换为 TD(λ) 版本以获得

$$\boldsymbol{w}_{t+1} = \boldsymbol{w}_t + \eta\delta_t \boldsymbol{z}_t \tag{2.27}$$

查看 [Sei+16] 以获取更多详细信息。

2.4 SARSA：基于策略的 TD 控制

TD 学习用于策略评估，因为它估计固定策略的价值函数。为了找到最优策略，我们可以在广义策略迭代（第 2.2.3 节）中将该算法作为构建块使用。在这种情况下，处理动作值函数，Q，以及一个相对于贪婪的策略 π 更为方便。代理在每个步骤中遵循 π 来选择动作，在转换 (s, a, r, s') 时，TD 更新规则是

$$Q(s,a) \leftarrow Q(s,a) + \eta\left[r + \gamma Q(s', a') - Q(s,a)\right] \tag{2.28}$$

其中 $a' \sim \pi(s')$ 是代理在状态 s' 中将要采取的动作。在 Q 更新（用于策略评估）后，π 也相应地改变，因为它对 Q （用于策略改进）是贪婪的。该算法最初由 [RN94]，提出，并由 Sut 进一步研究并更名为 **SARSA** ；其名称来源于其涉及增强转换 (s, a, r, s', a') 的更新规则。

为了使 SARSA 收敛到 Q_*，每个状态 - 动作对必须被无限次访问，至少在表格情况下，因为算法只更新 $Q(s,a)$ 对于它访问的 (s,a) 。确保这种条件的一种方法是用 “ 极限贪婪无限探索 ”(GLIE) 策略。一个例子是 ϵ- 贪婪策略，其中 ϵ 逐渐消失到 0 。可以证明，具有 GLIE 策略的 SARSA 将收敛到 Q_* 和 π_* [Sin+00]。

2.5 Q-learning: off-policy TD control

SARSA is an on-policy algorithm, which means it learns the Q-function for the policy it is currently using, which is typically not the optimal policy, because of the need to perform exploration. However, with a simple modification, we can convert this to an off-policy algorithm that learns Q_*, even if a suboptimal or exploratory policy is used to choose actions.

2.5.1 Tabular Q learning

Suppose we modify SARSA by replacing the sampled next action $a' \sim \pi(s')$ in Equation (2.28) with a greedy action: $a' = \text{argmax}_b Q(s', b)$. This results in the following update when a transition (s, a, r, s') happens

$$Q(s,a) \leftarrow Q(s,a) + \eta \left[r + \gamma \max_{a'} Q(s', a') - Q(s,a) \right] \tag{2.29}$$

This is the update rule of **Q-learning** for the tabular case [WD92].

Since it is off-policy, the method can use (s, a, r, s') triples coming from any data source, such as older versions of the policy, or log data from an existing (non-RL) system. If every state-action pair is visited infinitely often, the algorithm provably converges to Q_* in the tabular case, with properly decayed learning rates [Ber19]. Algorithm 1 gives a vanilla implementation of Q-learning with ϵ-greedy exploration.

Algorithm 1: Tabular Q-learning with ϵ-greedy exploration

Initialize value function Q
repeat
  Sample starting state s of new episode
  repeat
    Sample action $a = \begin{cases} \text{argmax}_b Q(s,b), & \text{with probability } 1-\epsilon \\ \text{random action}, & \text{with probability } \epsilon \end{cases}$
    $(s', r) = \text{env.step}(a)$
    Compute the TD error: $\delta = r + \gamma \max_{a'} Q(s', a') - Q(s, a)$
    $Q(s,a) \leftarrow Q(s,a) + \eta\delta$
    $s \leftarrow s'$
  until *state s is terminal*
until *converged*

For terminal states, $s \in \mathcal{S}^+$, we know that $Q(s, a) = 0$ for all actions a. Consequently, for the optimal value function, we have $V^*(s) = \max_{a'} Q^*(s, a) = 0$ for all terminal states. When performing online learning, we don't usually know which states are terminal. Therefore we assume that, whenever we take a step in the environment, we get the next state s' and reward r, but also a binary indicator $\text{done}(s')$ that tells us if s' is terminal. In this case, we set the target value in Q-learning to $V^*(s') = 0$ yielding the modified update rule:

$$Q(s,a) \leftarrow Q(s,a) + \eta \left[r + (1 - \text{done}(s'))\gamma \max_{a'} Q(s', a') - Q(s,a) \right] \tag{2.30}$$

For brevity, we will usually ignore this factor in the subsequent equations, but it needs to be implemented in the code.

Figure 2.5 gives an example of Q-learning applied to the simple 1d grid world from Figure 2.1, using $\gamma = 0.9$. We show the Q-functon at the start and end of each episode, after performing actions chosen by an ϵ-greedy policy. We initialize $Q(s, a) = 0$ for all entries, and use a step size of $\eta = 1$. At convergence, we have $Q_*(s, a) = r + \gamma Q_*(s', a_*)$, where $a_* = \downarrow$ for all states.

2.5 Q-learning：离策略 TD 控制

SARSA 是一种基于策略的算法，这意味着它学习当前使用的策略的 Q- 函数，这通常不是最优策略，因为需要执行探索。然而，通过简单的修改，我们可以将其转换为离策略算法，即使使用次优或探索性策略来选择动作，也能学习 Q_*。

2.5.1 表格 Q 学习

假设我们通过将方程（2.28）中的采样下一个动作 $a' \sim \pi(s')$ 替换为贪婪动作：$a' = \text{argmax}_b Q(s', b)$ 来修改 SARSA。当发生转换 (s, a, r, s') 时，这会导致以下更新：

$$Q(s,a) \leftarrow Q(s,a) + \eta \left[r + \gamma \max_{a'} Q(s', a') - Q(s,a) \right] \tag{2.29}$$

这是表格情况下的 Q-learning **更新规则**。[WD92]。

由于它是离策略的，该方法可以使用 (s, a, r, s') 三元组，这些三元组来自任何数据源，例如策略的旧版本，或现有（非 RL）系统的日志数据。如果每个状态 - 动作对都被无限次访问，则在表格情况下，该算法可以证明收敛到 Q_*，具有适当衰减的学习率 [Ber19]。算法 1 提供了一个具有 ϵ- 贪婪探索的 Q 学习的标准实现。

算法 1： 表格 Q 学习与 ϵ- 贪婪探索

Initialize value function Q
repeat
  Sample starting state s of new episode
  repeat
    Sample action $a = \begin{cases} \text{argmax}_b Q(s,b), & \text{with probability } 1-\epsilon \\ \text{random action}, & \text{with probability } \epsilon \end{cases}$
    $(s', r) = \text{env.step}(a)$
    Compute the TD error: $\delta = r + \gamma \max_{a'} Q(s', a') - Q(s, a)$
    $Q(s,a) \leftarrow Q(s,a) + \eta\delta$
    $s \leftarrow s'$
  until *state s is terminal*
until *converged*

对于终端状态，$s \in \mathcal{S}^+$，我们知道对于所有动作 a，有 $Q(s,a) = 0$。因此，对于最优值函数，我们有对于所有终端状态，$V^*(s) = \max_{a'} Q^*(s,a) = 0$。在进行在线学习时，我们通常不知道哪些状态是终端状态。因此，我们假设，每当我们在环境中迈出一步时，我们得到下一个状态 s' 和奖励 r，还有一个二元指示器 done(s')，告诉我们 s' 是否是终端状态。在这种情况下，我们将 Q 学习中的目标值设置为 $V^*(s') = 0$，从而得到修改后的更新规则：

$$Q(s,a) \leftarrow Q(s,a) + \eta \left[r + (1 - \text{done}(s'))\gamma \max_{a'} Q(s', a') - Q(s,a) \right] \tag{2.30}$$

为了简洁，我们通常会在后续方程中忽略这个因素，但需要在代码中实现。

图 2.5 提供了一个将 Q 学习应用于图 2.1 中的简单 1 维网格世界的例子，使用 $\gamma = 0.9$。我们展示了在每个剧集开始和结束时执行由 - 贪婪策略选择的动作后，Q- 函数。我们初始化 $Q(s,a) = 0$ 的所有条目，并使用步长 $\eta = 1$。在收敛时，我们有 $Q_*(s,a) = r + \gamma Q_*(s', a_*)$，其中 $a_* = \downarrow$ 对于所有状态。

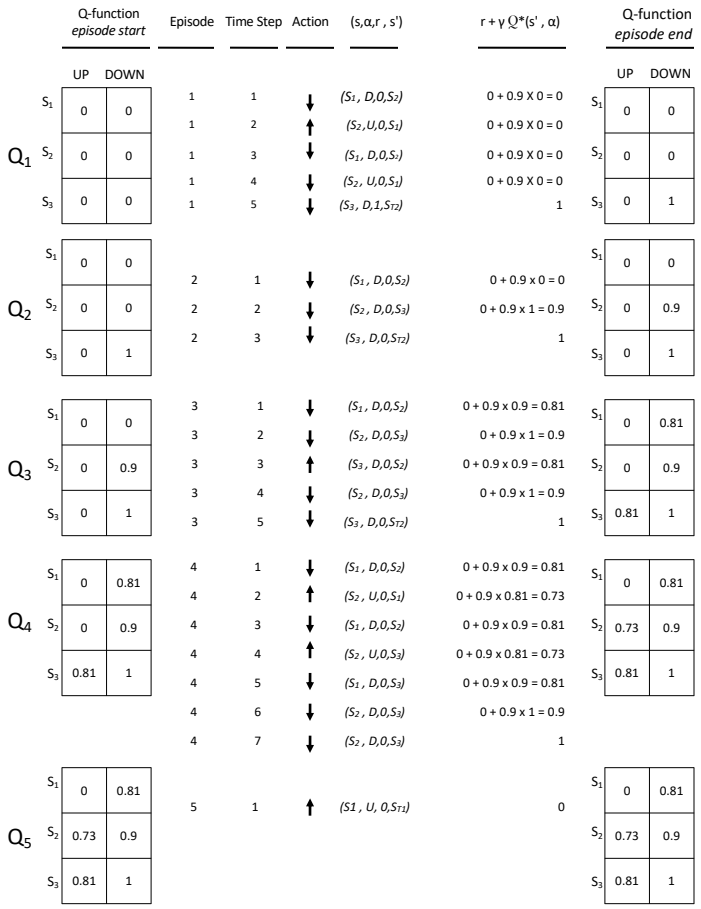


Figure 2.5: Illustration of Q learning for one random trajectory in the 1d grid world in Figure 2.1 using ϵ-greedy exploration. At the end of episode 1, we make a transition from S_3 to S_{T2} and get a reward of $r = 1$, so we estimate $Q(S_3, \downarrow) = 1$. In episode 2, we make a transition from S_2 to S_3, so S_2 gets incremented by $\gamma Q(S_3, \downarrow) = 0.9$. Adapted from Figure 3.3 of [GK19].

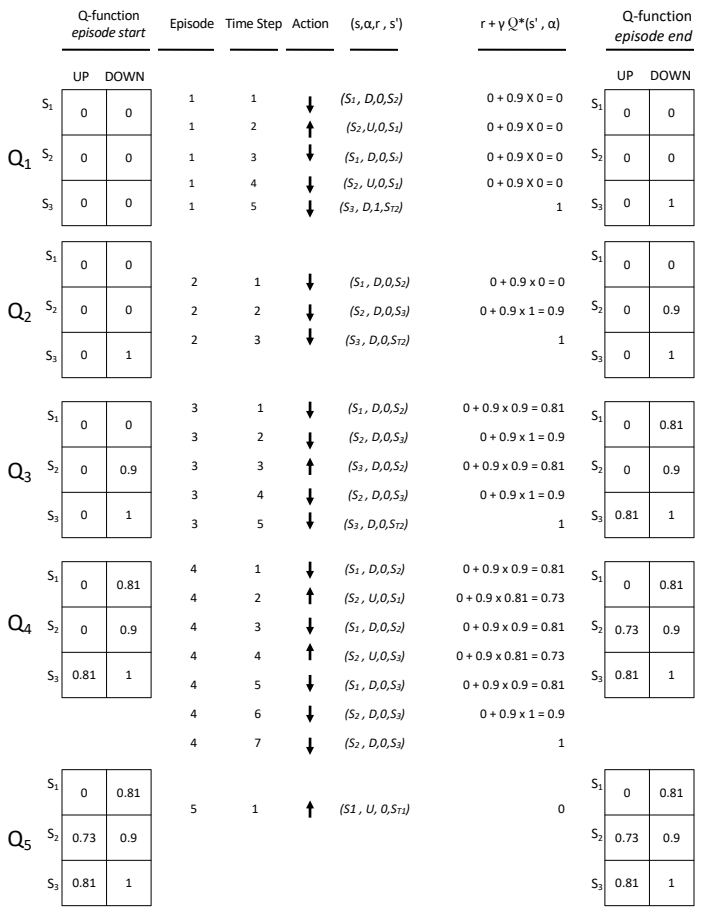


图 2.5：图 2.1 中 1 维网格世界中一条随机轨迹的 Q 学习示意图。在第 1 个回合结束时，我们从 S_3 转移到 S_{T2} 并获得 $r=1$ 的奖励，因此我们估计 $Q(S_3,\downarrow)$。在第 2 个回合中，我们从 S_2 转移到 S_3，因此 S_2 增加 $\gamma Q(S_3,\downarrow)$。改编自 GK/ 图 3.3。

2.5.2 Q learning with function approximation

To make Q learning work with high-dimensional state spaces, we have to replace the tabular (non-parametric) representation with a parametric approximation, denoted $Q_{\boldsymbol{w}}(s, a)$. We can update this function using one or more steps of SGD on the following loss function

$$\mathcal{L}(\boldsymbol{w}|\boldsymbol{s}, a, r, \boldsymbol{s}') = \left((r + \gamma \max_{a'} Q_{\boldsymbol{w}}(\boldsymbol{s}', a')) - Q_{\boldsymbol{w}}(\boldsymbol{s}, a)\right)^2 \tag{2.31}$$

Since nonlinear functions need to be trained on minibatches of data, we compute the average loss over multiple randomly sampled experience tuples (see Section 2.5.2.3 for discussion) to get

$$\mathcal{L}(\boldsymbol{w}) = \mathbb{E}_{(\boldsymbol{s},a,r,\boldsymbol{s}')\sim U(\mathcal{D})}\left[\mathcal{L}(\boldsymbol{w}|\boldsymbol{s}, a, r, \boldsymbol{s}')\right] \tag{2.32}$$

See Algorithm 2 for the pseudocode.

Algorithm 2: Q learning with function approximation and replay buffers

Initialize environment state $\boldsymbol{s}$, network parameters $\boldsymbol{w}_0$, replay buffer $\mathcal{D} = \emptyset$, discount factor γ, step size η, policy $\pi_0(a|s) = \epsilon \text{Unif}(a) + (1-\epsilon)\delta(a = \text{argmax}_a Q_{\boldsymbol{w}_0}(s, a))$
for *iteration* $k = 0, 1, 2, \ldots$ **do**
  for *environment step* $s = 0, 1, \ldots, S-1$ **do**
    Sample action: $a \sim \pi_k(a|s)$
    Interact with environment: $(s', r) = \text{env.step}(a)$
    Update buffer: $\mathcal{D} \leftarrow \mathcal{D} \cup \{(s, a, s', r)\}$
  $\boldsymbol{w}_{k,0} \leftarrow \boldsymbol{w}_k$
  for *gradient step* $g = 0, 1, \ldots, G-1$ **do**
    Sample batch: $B \subset \mathcal{D}$
    Compute error: $\mathcal{L}(B, \boldsymbol{w}_{k,g}) = \frac{1}{|B|} \sum_{(s,a,r,s')\in B} \left[Q_{\boldsymbol{w}_{k,g}}(s, a) - (r + \gamma \max_{a'} Q_{\boldsymbol{w}_k}(s', a'))\right]^2$
    Update parameters: $\boldsymbol{w}_{k,g} \leftarrow \boldsymbol{w}_{k,g} - \eta \nabla_{\boldsymbol{w}_{k,g}} \mathcal{L}(B, \boldsymbol{w}_{k,g})$
  $\boldsymbol{w}_{k+1} \leftarrow \boldsymbol{w}_{k,G}$

2.5.2.1 Neural fitted Q

The first approach of this kind is known as **neural fitted Q iteration** [Rie05], which corresponds to fully optimizing $\mathcal{L}(\boldsymbol{w})$ at each iteration (equivalent to using $G = \infty$ gradient steps).

2.5.2.2 DQN

The influential deep Q-network or DQN paper of [Mni+15] also used neural nets to represent the Q function, but performed a smaller number of gradient updates per iteration. Furthermore, they proposed to modify the target value when fitting the Q function in order to avoid instabilities during training (see Section 2.5.2.4 for details).

The DQN method became famous since it was able to train agents that can outperform humans when playing various Atari games from the **ALE** (Atari Learning Environment) benchmark [Bel+13]. Here the input is a small color image, and the action space corresponds to moving left, right, up or down, plus an optional shoot action.[1]

Since 2015, many more extensions to DQN have been proposed, with the goal of improving performance in various ways, either in terms of peak reward obtained, or sample efficiency (e.g., reward obtained after only

[1] For more discussion of ALE, see [Mac+18a], and for a recent extension to continuous actions (representing joystick control), see the CALE benchmark of [FC24]. Note that DQN was not the first deep RL method to train an agent from pixel input; that honor goes to [LR10], who trained an autoencoder to embed images into low-dimensional latents, and then used neural fitted Q learning (Section 2.5.2.1) to fit the Q function.

2.5.2 使用函数逼近的 Q 学习

为了使 Q 学习能够在高维状态空间中工作，我们必须用参数近似来替换表格（非参数）表示，记为 $Q_{\boldsymbol{w}}(s,a)$。我们可以通过以下损失函数 $\left(\quad\right)$ 使用一个或多个步骤的随机梯度下降来更新这个函数。

$$\mathcal{L}(\boldsymbol{w}|\boldsymbol{s},a,r,\boldsymbol{s}') = \left(r+\gamma\max_{a'}Q_{\boldsymbol{w}}(\boldsymbol{s}',a')\right)-Q_{\boldsymbol{w}}(\boldsymbol{s},a)\Big)^2 \tag{2.31}$$

由于非线性函数需要在数据的小批量上进行训练，我们计算多个随机采样的经验元组的平均损失（参见第 2.5.2.3 节进行讨论）以获得

$$\mathcal{L}(\boldsymbol{w}) = \mathbb{E}_{(\boldsymbol{s},a,r,\boldsymbol{s}')\sim U(\mathcal{D})}\left[\mathcal{L}(\boldsymbol{w}|\boldsymbol{s},a,r,\boldsymbol{s}')\right] \tag{2.32}$$

查看算法 2 的伪代码。

算法 2： 具有函数逼近和重放缓冲区的 Q 学习

初始化环境状态 $\boldsymbol{s}$，网络参数 $\boldsymbol{w}_0$，重放缓冲区 $\mathcal{D}=\emptyset$，折扣因子 γ，步长 η，策略 $\pi_0(a|s)=\epsilon$均匀分布 $(a)+(1-\epsilon)\delta(a=\text{argmax}_a\, Q_{\boldsymbol{w}_0}(s,a))$**对于** 迭代 $k=0,\ 1,\ 2,\ \ldots$ **执行** **对于** 环境步 $s=0,\ 1,\ \ldots,\ S-1$ **执行** 采样动作：$a\sim\pi_k(a|s)$ 与环境交互：$(s',r)=\text{env.step}(a)$ 更新缓冲区：$\mathcal{D}\leftarrow\mathcal{D}\cup\{(s,a,s',r)\}$ $\boldsymbol{w}_{k,0}\leftarrow\boldsymbol{w}_k$**对于** 梯度步 $g=0,\ 1,\ \ldots,\ G-1$ **执行** 采样批量：$B\subset\mathcal{D}$ 计算误差：$\mathcal{L}(B,\boldsymbol{w}_{k,g})=\frac{1}{|B|}\sum_{(s,a,r,s')\in B}\left[Q_{\boldsymbol{w}_{k,g}}(s,a)-(r+\gamma\max_{a'}Q_{\boldsymbol{w}_k}(s',a'))\right]^2$ 更新参数：$\boldsymbol{w}_{k,g}\leftarrow\boldsymbol{w}_{k,g}-\eta\nabla_{\boldsymbol{w}_{k,g}}\mathcal{L}(B,\boldsymbol{w}_{k,g})$ $\boldsymbol{w}_{k+1}\leftarrow\boldsymbol{w}_{k,G}$

2.5.2.1 神经拟合 Q

这种方法的第一次应用被称为**神经拟合 Q 迭代** [Rie05], , 这对应于在每次迭代中完全优化 $\mathcal{L}$（$\boldsymbol{w}$）（相当于使用 $G=\infty$ 梯度步数）。

2.5.2.2 DQN 深度 Q 网络

有影响力的深度 Q 网络或 DQN 论文 [Mni+15] 也使用了神经网络来表示 Q 函数，但每次迭代的梯度更新次数较少。此外，他们提出了在拟合 Q 函数时修改目标值，以避免训练过程中的不稳定性（详见第 2.5.2.4 节，获取详细信息）。

DQN 方法因其能够训练出在玩各种 Atari 游戏时能超越人类的智能体而闻名。这里输入的是一个小彩色图像，动作空间对应于左右上下移动，以及可选的射击动作。[1]

自 2015 年以来，针对 DQN 提出了许多扩展，旨在通过各种方式提高性能，无论是峰值奖励还是样本效率（例如，仅在获得少量样本后获得的奖励）

[1]关于ALE的更多讨论，请参阅[Mac+18a], , 以及关于连续动作（表示摇杆控制）的最近扩展，请参阅[FC24]的CALE基准。请注意，DQN 并不是第一个从像素输入训练代理的深度强化学习方法；这个荣誉属于 [LR10], , 他训练了一个自动编码器将图像嵌入到低维潜在空间中，然后使用神经拟合 Q 学习（第 2.5.2.1 节）来拟合 Q 函数。

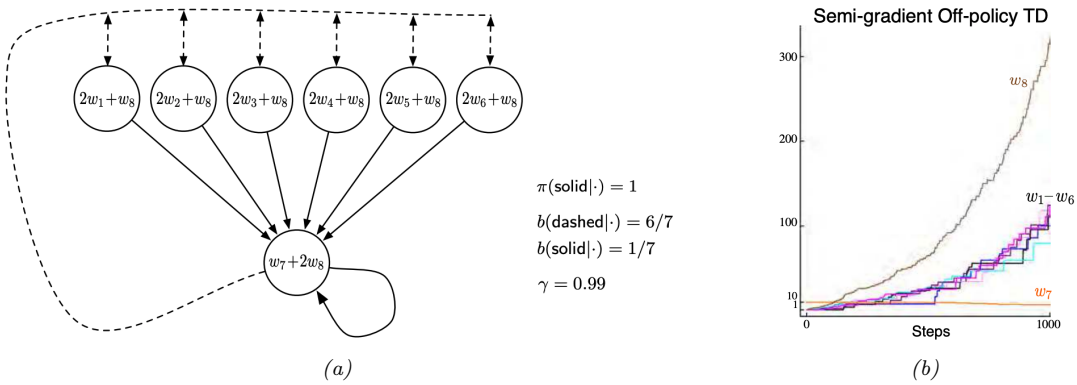


Figure 2.6: (a) A simple MDP. (b) Parameters of the policy diverge over time. From Figures 11.1 and 11.2 of [SB18]. Used with kind permission of Richard Sutton.

100k steps in the environment, as proposed in the **Atari-100k** benchmark [Kai+19]), or training stability, or all of the above. We discuss some of these extensions in Section 2.5.4.

2.5.2.3 Experience replay

Since Q learning is an off-policy method, we can update the Q function using any data source. This is particularly important when we use nonlinear function approximation (see Section 2.5.2), which often needs a lot of data for model fitting. A natural source of data is data collected earlier in the trajectori of the agent; this is called an **experience replay** buffer, which stores (s, a, r, s') transition tuples into a buffer. This can improve the stability and sample efficiency of learning, and was originally proposed in [Lin92].

This modification has two advantages. First, it improves data efficiency as every transition can be used multiple times. Second, it improves stability in training, by reducing the correlation of the data samples that the network is trained on, since the training tuples do not have to come from adjacent moments in time. (Note that experience replay requires the use of off-policy learning methods, such as Q learning, since the training data is sampled from older versions of the policy, not the current policy.)

It is possible to replace the uniform sampling from the buffer with one that favors more important transition tuples that may be more informative about Q. This idea is formalized in [Sch+16a], who develop a technique known as **prioritized experience replay**.

2.5.2.4 The deadly triad

The problem with the naive Q learning objective in Equation (2.31) is that it can lead to instability, since the target we are regressing towards uses the same parameters $\boldsymbol{w}$ as the function we are updating. So the network is "chasing its own tail". Although this is fine for tabular models, it can fail for nonlinear models, as we discuss below.

In general, an RL algorithm can become unstable when it has these three components: function approximation (such as neural networks), bootstrapped value function estimation (i.e., using TD-like methods instead of MC), and off-policy learning (where the actions are sampled from some distribution other than the policy that is being optimized). This combination is known as **the deadly triad** [Sut15; van+18]).

A classic example of this is the simple MDP depicted in Figure 2.6a, due to [Bai95]. (This is known as **Baird's counter example**.) It has 7 states and 2 actions. Taking the dashed action takes the environment to the 6 upper states uniformly at random, while the solid action takes it to the bottom state. The reward is 0 in all transitions, and $\gamma = 0.99$. The value function $V_{\boldsymbol{w}}$ uses a linear parameterization indicated by the expressions shown inside the states, with $\boldsymbol{w} \in \mathbb{R}^8$. The target policies π always chooses the solid action in every state. Clearly, the true value function, $V_\pi(s) = 0$, can be exactly represented by setting $\boldsymbol{w} = \mathbf{0}$.

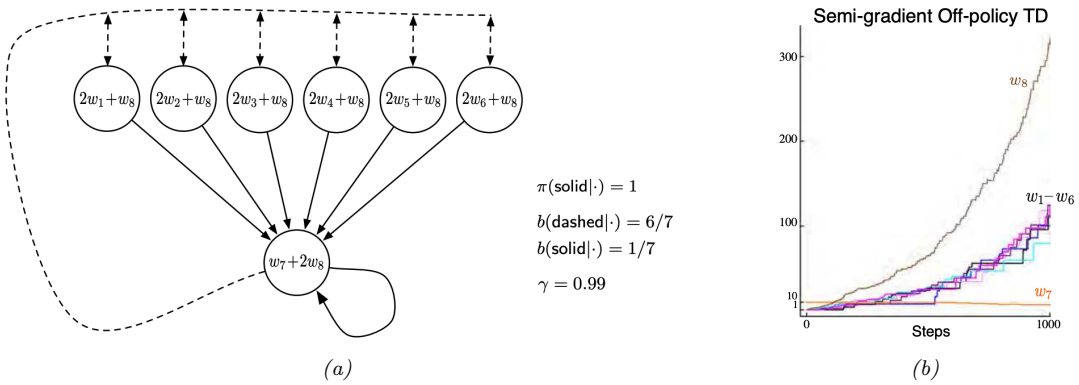


图 2.6：（a）一个简单的 MDP。（b）策略参数随时间变化。来自 *[SB18]* 的第 11.1 和 11.2 图。经理查德·萨顿许可使用。

在环境中进行 100k 步，如 **Atari-100k** 基准测试 [Kai+19]）中提出的那样，或者训练稳定性，或者所有上述内容。我们在第 2.5.4 节中讨论了这些扩展之一。

2.5.2.3 经验回放

由于 Q 学习是一种离线策略方法，我们可以使用任何数据源来更新 Q 函数。这在使用非线性函数逼近（见第 2.5.2 节）时尤为重要，这通常需要大量数据来拟合模型。数据的一个自然来源是代理在轨迹中早期收集的数据；这被称为经验重放缓冲区，它将 (s, a, r, s') 转换元组存储到缓冲区中。这可以提高学习的稳定性和样本效率，最初由 [Lin92] 提出。

这种修改有两个优点。首先，它提高了数据效率，因为每个转换都可以多次使用。其次，它通过减少网络训练所使用的数据样本之间的相关性来提高训练的稳定性，因为训练元组不必来自相邻的时间点。（请注意，经验重放需要使用离线学习方法，如 Q 学习，因为训练数据是从策略的旧版本中采样的，而不是当前策略。）

可能用更倾向于重要转换元组的采样来替换缓冲区中的均匀采样，这些转换元组可能对 Q 提供更多信息。这一想法在 [Sch+16a]，的研究中得到了形式化，他们开发了一种称为 **优先经验回放** 的技术。

2.5.2.4 致命三联征

方程（2.31）中朴素 Q 学习目标的问题在于，由于我们回归的目标函数使用与我们要更新的函数相同的参数 $\boldsymbol{w}$，这可能导致不稳定性。因此，网络 “ 在追逐自己的尾巴 ”。尽管这对表格模型来说是可以的，但正如我们下面所讨论的，它可能对非线性模型失败。

通常情况下，当 RL 算法包含以下三个组件时，可能会变得不稳定：函数逼近（如神经网络）、自举值函数估计（即使用 TD-like 方法而不是 MC）以及离策略学习（动作是从除正在优化的策略之外的分布中采样的）。这种组合被称为**致命三联症** [Sut15； van+18]）。

这是一个经典的例子，如图 2.6a 所示，由于 [Bai95]。（这被称为 **Baird 的逆例**。）它有 7 个状态和 2 个动作。执行虚线动作将环境随机均匀地转移到 6 个上状态，而实线动作将其转移到底部状态。所有转换的奖励都是 0，并且 $\gamma = 0.99$。值函数 $V_{\boldsymbol{w}}$ 使用由状态内所示表达式指示的线性参数化，$\boldsymbol{w} \in \mathbb{R}^8$。目标策略 π 在所有状态下始终选择实线动作。显然，真实值函数 $V_\pi(s) = 0$ 可以通过设置 $\boldsymbol{w} = \mathbf{0}$。来精确表示。

Suppose we use a behavior policy b to generate a trajectory, which chooses the dashed and solid actions with probabilities 6/7 and 1/7, respectively, in every state. If we apply TD(0) on this trajectory, the parameters diverge to ∞ (Figure 2.6b), even though the problem appears simple. In contrast, with on-policy data (that is, when b is the same as π), TD(0) with linear approximation can be guaranteed to converge to a good value function approximate [TR97]. The difference is that with on-policy learning, as we improve the value function, we also improve the policy, so the two become self-consistent, whereas with off-policy learning, the behavior policy may not match the optimal value function that is being learned, leading to inconsistencies.

The divergence behavior is demonstrated in many value-based bootstrapping methods, including TD, Q-learning, and related approximate dynamic programming algorithms, where the value function is represented either linearly (like the example above) or nonlinearly [Gor95; TVR97; OCD21]. The root cause of these divergence phenomena is that bootstrapping methods typically are not minimizing a fixed objective function. Rather, they create a learning target using their own estimates, thus potentially creating a self-reinforcing loop to push the estimates to infinity. More formally, the problem is that the contraction property in the tabular case (Equation (2.10)) may no longer hold when V is approximated by $V_{\boldsymbol{w}}$.

We discuss some solutions to the deadly triad problem below.

2.5.2.5 Target networks

One heuristic solution to the deadly triad, proposed in the DQN paper, is to use a "frozen" **target network** computed at an earlier iteration to define the target value for the DQN updates, rather than trying to chase a constantly moving target. Specifically, we maintain an extra copy the Q-network, $Q_{\boldsymbol{w}^-}$, with the same structure as $Q_{\boldsymbol{w}}$. This new Q-network is used to compute bootstrapping targets

$$q(r, \boldsymbol{s}'; \boldsymbol{w}^-) = r + \gamma \max_{a'} Q_{\boldsymbol{w}^-}(\boldsymbol{s}', a') \tag{2.33}$$

for training $Q_{\boldsymbol{w}}$. We can periodically set $\boldsymbol{w}^- \leftarrow \text{sg}(\boldsymbol{w})$, usually after a few episodes, where the stop gradient operator is used to prevent autodiff propagating gradients back to $\boldsymbol{w}$. Alternatively, we can use an exponential moving average (EMA) of the weights, i.e., we use $\overline{\boldsymbol{w}} = \rho\overline{\boldsymbol{w}} + (1-\rho)\text{sg}(\boldsymbol{w})$, where $\rho \ll 1$ ensures that $Q_{\overline{\boldsymbol{w}}}$ slowly catches up with $Q_{\boldsymbol{w}}$. (If $\rho = 0$, we say that this is a **detached target**, since it is just a frozen copy of the current weights.) The final loss has the form

$$\mathcal{L}(\boldsymbol{w}) = \mathbb{E}_{(\boldsymbol{s},a,r,\boldsymbol{s}') \sim U(\mathcal{D})} \left[\mathcal{L}(\boldsymbol{w}|\boldsymbol{s}, a, r, \boldsymbol{s}')\right] \tag{2.34}$$

$$\mathcal{L}(\boldsymbol{w}|\boldsymbol{s}, a, r, \boldsymbol{s}') = (q(r, \boldsymbol{s}'; \overline{\boldsymbol{w}}) - Q_{\boldsymbol{w}}(\boldsymbol{s}, a))^2 \tag{2.35}$$

Theoretical work justifying this technique is given in [FSW23; Che+24a].

2.5.2.6 Two time-scale methods

A general way to ensure convergence in off-policy learning is to construct an objective function, the minimization of which leads to a good value function approximation. This is the basis of the **gradient TD method** of [SSM08; Mae+09; Ghi+20]. In practice, this can be achieved by updating the target value in the TD update more quickly than the value function itself; this is known as a **two timescale optimization** (see e.g., [Yu17; Zha+19; Hon+23]). It is also possible to use a standard single timescale method provided the target value is computed using a frozen target network, as discussed in Section 2.5.2.5. See [FSW23; Che+24a] for details.

2.5.2.7 Layer norm

More recently, [Gal+24] proved that just adding LayerNorm [BKH16] to the penultimate layer of the critic network, just before the linear head, is sufficient to provably yield convergence of TD learning even in the off-policy setting. In particular, suppose the network has the form $Q(s, a|\boldsymbol{w}, \boldsymbol{\theta}) = \boldsymbol{w}^T \text{ReLU}(\text{LayerNorm}(f(s, a; \boldsymbol{\theta})))$. Since $||\text{LayerNorm}(f(s, a; \boldsymbol{\theta}))|| \leq 1$, we have $||Q(s, a|\boldsymbol{w}, \boldsymbol{\theta}) \leq ||\boldsymbol{w}||$, which means the magnitude of the output is always bounded, as shown in Figure 2.7. In [Gal+24], they prove this (plus ℓ_2 regularization on $\boldsymbol{w}$, and a sufficiently wide penultimate layer) is sufficient to ensure convergence of the value function estimate.

假设我们使用行为策略 b 来生成轨迹，该策略在每个状态下以概率 $6/7$ 和 $1/7$ 分别选择虚线和实线动作。如果我们对这条轨迹应用 TD(0)，参数会发散到 ∞ （图 2.6b ），尽管问题看起来很简单。相比之下，在有策略数据（即，当 b 与 π 相同时），带有线性近似的 TD(0) 可以保证收敛到一个好的值函数近似 [TR97]。区别在于，在有策略学习中，随着我们改进值函数，我们也改进策略，因此两者变得自洽，而离策略学习中，行为策略可能不匹配正在学习的最优值函数，导致不一致。

这些发散现象的根本原因在于，自举方法通常不是最小化一个固定的目标函数。相反，它们使用自己的估计来创建一个学习目标，从而可能形成一个自我强化的循环，将估计推向无穷大。更正式地说，问题在于，当 V 被 $V_{\boldsymbol{w}}$ 近似时，表格情况下的收缩性质（方程（ 2.10 ））可能不再成立。

我们下面讨论一些针对致命三联征问题的解决方案。

2.5.2.5 目标网络

一种针对致命三联征的启发式解决方案，在 DQN 论文中提出，是使用一个 “ 冻结 ” 的**目标网络**，在早期迭代中计算，以定义 DQN 更新的目标值，而不是试图追逐一个不断移动的目标。具体来说，我们维护一个额外的 Q- 网络副本 $Q_{\boldsymbol{w}^-}$，其结构与 $Q_{\boldsymbol{w}}$ 相同。这个新的 Q- 网络用于计算引导目标

$$q(r, \boldsymbol{s}'; \boldsymbol{w}^-) = r + \gamma \max_{a'} Q_{\boldsymbol{w}^-}(\boldsymbol{s}', a') \tag{2.33}$$

for training $Q_{\boldsymbol{w}}$。我们可以定期设置 $\boldsymbol{w}^- \leftarrow \text{sg}(\boldsymbol{w})$，通常在几轮之后，使用停止梯度算子防止自动微分将梯度传播回 $\boldsymbol{w}$。或者，我们可以使用权重的指数移动平均（ EMA ），即我们使用 $\boldsymbol{w} = \rho \boldsymbol{w} + (1-\rho)\text{sg}(\boldsymbol{w})$，其中 $\rho \ll 1$ 确保 $Q_{\boldsymbol{w}}$ 慢慢赶上 $Q_{\boldsymbol{w}}$。（如果 $\rho = 0$，我们称这为 **解耦目标**，因为它只是当前权重的冻结副本。）最终的损失形式为

$$\mathcal{L}(\boldsymbol{w}) = \mathbb{E}_{(\boldsymbol{s},a,r,\boldsymbol{s}') \sim U(\mathcal{D})} \left[\mathcal{L}(\boldsymbol{w} | \boldsymbol{s}, a, r, \boldsymbol{s}') \right] \tag{2.34}$$

$$\mathcal{L}(\boldsymbol{w} | \boldsymbol{s}, a, r, \boldsymbol{s}') = (q(r, \boldsymbol{s}'; \boldsymbol{w}) - Q_{\boldsymbol{w}}(\boldsymbol{s}, a))^2 \tag{2.35}$$

理论工作中对这种技术的论证见 [FSW23； Che+24a].

2.5.2.6 双时间尺度方法

确保离策略学习中收敛的一般方法是通过构建一个目标函数，其最小化会导致良好的值函数近似。这是 SSM08； Mae； Ghi 梯度 TD 方法的基础； Mae+09； Ghi+20]。在实践中，这可以通过比值函数本身更快地更新 TD 更新中的目标值来实现；这被称为**双时间尺度优化**（例如，参见 Yu17[； Zha+19； Hon+23] ）。如果目标值是使用冻结的目标网络计算的，也可以使用标准的单时间尺度方法，如第 2.5.2.5 节所述。有关详细信息，请参阅 FSW23[； Che+24]。

2.5.2.7 层归一化

最近，[Gal+24] 证明了只需在评论家网络的倒数第二层，即在线性头之前，添加 LayerNorm [BKH16] ，就可以在离线策略设置中保证 TD 学习的收敛。特别是，假设网络具有形式 $Q(s, a | \boldsymbol{w}, \boldsymbol{\theta}) = \boldsymbol{w}^T \text{ReLU}(\text{LayerNorm}(f(s, a; \boldsymbol{\theta})))$。由于 $||\text{LayerNorm}(f(s, a; \boldsymbol{\theta}))|| \leq 1$，我们有 $||Q(s, a | \boldsymbol{w}, \boldsymbol{\theta}) \leq ||\boldsymbol{w}||$，这意味着输出的幅度总是有界的，如图 2.7 所示。在 [Gal+24] 中，他们证明了这一点（加上 ℓ_2 对 $\boldsymbol{w}$ 的正则化，以及足够宽的倒数第二层）足以保证值函数估计的收敛。

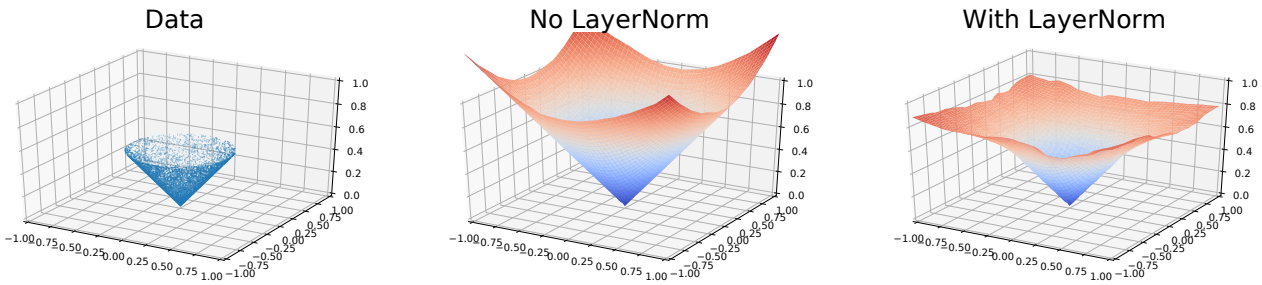


Figure 2.7: We generate a dataset (left) with inputs $\boldsymbol{x}$ distributed in a circle with radius 0.5 and labels $y = ||\boldsymbol{x}||$. We then fit a two-layer MLP without LayerNorm (center) and with LayerNorm (right). LayerNorm bounds the values and prevents catastrophic overestimation when extrapolating. From Figure 3 of [Bal+23]. Used with kind permission of Philip Ball.

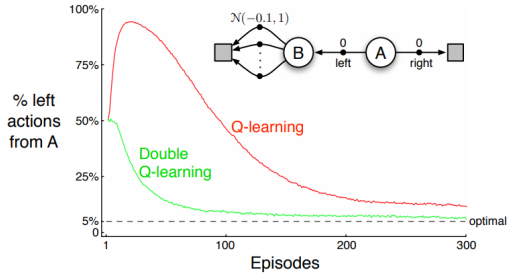


Figure 2.8: Comparison of Q-learning and double Q-learning on a simple episodic MDP using ϵ-greedy action selection with $\epsilon = 0.1$. The initial state is A, and squares denote absorbing states. The data are averaged over 10,000 runs. From Figure 6.5 of [SB18]. Used with kind permission of Richard Sutton.

2.5.3 Maximization bias

Standard Q-learning suffers from a problem known as the **optimizer's curse** [SW06], or the **maximization bias**. The problem refers to the simple statistical inequality: $\mathbb{E}\left[\max_a X_a\right] \geq \max_a \mathbb{E}\left[X_a\right]$, for a set of random variables $\{X_a\}$. Thus, if we pick actions greedily according to their random scores $\{X_a\}$, we might pick a wrong action just because random noise makes it appealing.

Figure 2.8 gives a simple example of how this can happen in an MDP. The start state is A. The right action gives a reward 0 and terminates the episode. The left action also gives a reward of 0, but then enters state B, from which there are many possible actions, with rewards drawn from $\mathcal{N}(-0.1, 1.0)$. Thus the expected return for any trajectory starting with the left action is -0.1, making it suboptimal. Nevertheless, the RL algorithm may pick the left action due to the maximization bias making B appear to have a positive value.

2.5.3.1 Double Q-learning

One solution to avoid the maximization bias is to use two separate Q-functions, Q_1 and Q_2, one for selecting the greedy action, and the other for estimating the corresponding Q-value. In particular, upon seeing a transition (s, a, r, s'), we perform the following update for $i = 1 : 2$:

$$Q_i(s,a) \leftarrow Q_i(s,a) + \eta(q_i(s,a) - Q_i(s,a)) \tag{2.36}$$

$$q_i(s,a) = r + \gamma Q_i(s', \operatorname*{argmax}_{a'} Q_{-i}(s', a')) \tag{2.37}$$

So we see that Q_1 uses Q_2 to choose the best action but uses Q_1 to evaluate it, and vice versa. This technique is called **double Q-learning** [Has10]. Figure 2.8 shows the benefits of the algorithm over standard Q-learning

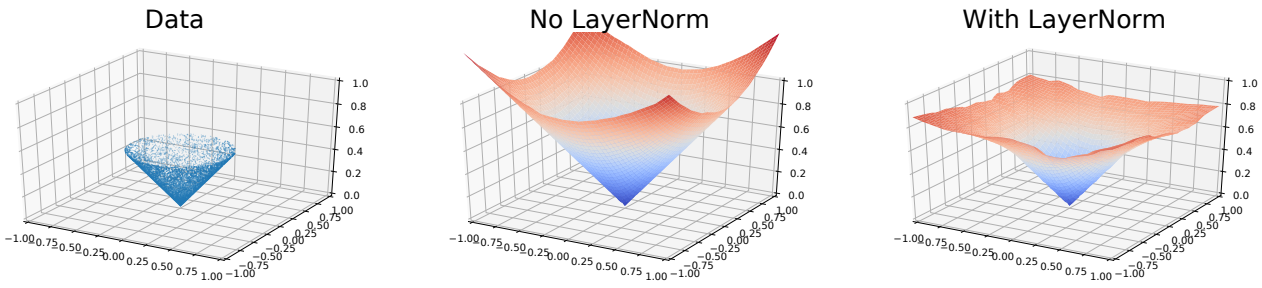


图 2.7：我们生成一个数据集（左侧），其中输入 $\boldsymbol{x}$ 分布在半径为 0.5 的圆中，标签为 $y = ||\boldsymbol{x}||$。然后我们拟合了一个没有 LayerNorm 层（中间）和带有 LayerNorm 层（右侧）的两层 MLP。LayerNorm 限制了值，并在外推时防止灾难性过估计。来自 *[Bal+23]* 的第 3 图。经 Philip Ball 许可使用。

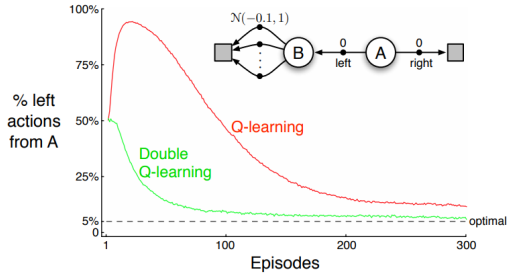


图 2.8：使用 ϵ- 贪心动作选择比较 Q 学习与双 Q 学习在简单的事件驱动的 MDP 上的表现。初始状态为 A，正方形表示吸收状态。数据是在 10000 次运行中平均得到的。来自 *[SB18]* 的第 6.5 图。经 Richard Sutton 许可使用。

2.5.3 最大化偏差

标准 Q 学习存在一个被称为**优化者诅咒** [SW06]，或**最大化偏差**的问题。这个问题指的是一个简单的统计不等式：$\mathbb{E}\left[\max_a X_a\right] \geq \max_a \mathbb{E}\left[X_a\right]$，对于一组随机变量 $\{X_a\}$。因此，如果我们根据它们的随机得分贪婪地选择动作 $\{X_a\}$，我们可能会因为随机噪声使其具有吸引力而选择错误的动作。

图 2.8 提供了一个简单示例，说明了这种情况如何在 MDP 中发生。起始状态是 A。向右的动作获得奖励 0 并结束游戏。向左的动作也获得奖励 0，但随后进入状态 B，从 B 状态有多个可能动作，奖励从 $\mathcal{N}(-0.1, 1.0)$ 中抽取。因此，以左动作开始的任何轨迹的期望回报是 -0.1，使其次优。尽管如此，由于最大化偏差，RL 算法可能会选择左动作，使 B 看起来具有正值。

2.5.3.1 双 Q 学习

一种避免最大化偏差的解决方案是使用两个独立的 Q- 函数，Q_1 和 Q_2，一个用于选择贪婪动作，另一个用于估计相应的 Q- 值。特别是，当我们看到转换（s，a，r，s'）时，我们对 $i = 1$ 执行以下更新：2：

$$Q_i(s,a) \leftarrow Q_i(s,a) + \eta(q_i(s,a) - Q_i(s,a)) \tag{2.36}$$

$$q_i(s,a) = r + \gamma Q_i(s', \operatorname*{argmax}_{a'} Q_{-i}(s',a')) \tag{2.37}$$

因此，我们看到 Q_1 使用 Q_2 来选择最佳动作，但使用 Q_1 来评估它，反之亦然。这种技术被称为**双 Q 学习** [具有 10]。图 2.8 显示了该算法相对于标准 Q 学习的优势

in a toy problem.

2.5.3.2 Double DQN

In [HGS16], they combine double Q learning with deep Q networks (Section 2.5.2.2) to get **double DQN**. This modifies Equation (2.37) to its gradient form, and then the current network for action proposals, but the target network for action evaluation. Thus the training target becomes

$$q(r, \boldsymbol{s}'; \boldsymbol{w}, \overline{\boldsymbol{w}}) = r + \gamma Q_{\overline{\boldsymbol{w}}}(\boldsymbol{s}', \operatorname*{argmax}_{a'} Q_{\boldsymbol{w}}(\boldsymbol{s}', a')) \tag{2.38}$$

In Section 3.6.2 we discuss an extension called **clipped double DQN** which uses two Q networks and their frozen copies to define the following target:

$$q(r, \boldsymbol{s}'; \boldsymbol{w}_{1:2}, \overline{\boldsymbol{w}}_{1:2}) = r + \gamma \min_{i=1,2} Q_{\overline{\boldsymbol{w}}_i}(\boldsymbol{s}', \operatorname*{argmax}_{a'} Q_{\boldsymbol{w}_i}(\boldsymbol{s}', a')) \tag{2.39}$$

where $Q_{\overline{\boldsymbol{w}}_i}$ is the target network for $Q_{\boldsymbol{w}_i}$.

2.5.3.3 Randomized ensemble DQN

The double DQN method is extended in the **REDQ** (randomized ensembled double Q learning) method of [Che+20], which uses an ensemble of $N > 2$ Q-networks. Furthermore, at each step, it draws a random sample of $M \leq N$ networks, and takes the minimum over them when computing the target value. That is, it uses the following update (see Algorithm 2 in appendix of [Che+20]):

$$q(r, \boldsymbol{s}'; \boldsymbol{w}_{1:N}, \overline{\boldsymbol{w}}_{1:N}) = r + \gamma \max_{a'} \min_{i \in \mathcal{M}} Q_{\overline{\boldsymbol{w}}_i}(\boldsymbol{s}', a') \tag{2.40}$$

where $\mathcal{M}$ is a random subset from the N value functions. The ensemble reduces the variance, and the minimum reduces the overestimation bias.[2] If we set $N = M = 2$, we get a method similar to clipped double Q learning. (Note that REDQ is very similiar to the **Random Ensemble Mixture** method of [ASN20], which was designed for offline RL.)

2.5.4 DQN extensions

In this section, we discuss various extensions of DQN.

2.5.4.1 Q learning for continuous actions

Q learning is not directly applicable to continuous actions due to the need to compute the argmax over actions. An early solution to this problem, based on neural fitted Q learning (see Section 2.5.2.1), is proposed in [HR11]. This became the basis of the DDPG algorithm of Section 3.6.1, which learns a policy to predict the argmax.

An alternative approach is to use gradient-free optimizers such as the cross-entropy method to approximate the argmax. The **QT-Opt** method of [Kal+18] treats the action vector $\boldsymbol{a}$ as a sequence of actions, and optimizes one dimension at a time [Met+17]. The **CAQL** (continuous action Q-learning) method of [Ryu+20]) uses mixed integer programming to solve the argmax problem, leveraging the ReLU structure of the Q-network. The method of [Sey+22] quantizes each action dimension separately, and then solves the argmax problem using methods inspired by multi-agent RL.

[2] In addition, REDQ performs $G \gg 1$ updates of the value functions for each environment step; this high **Update-To-Data** (UTD) ratio (also called **Replay Ratio**) is critical for sample efficiency, and is commonly used in model-based RL.

在一个玩具问题中。

2.5.3.2 双重 DQN

在 [HGS16]，中，他们结合了双 Q 学习和深度 Q 网络（第 2.5.2.2 节）以获得**双 DQN**。这修改了方程（ 2.37 ）为梯度形式，然后是用于动作建议的当前网络，但用于动作评估的目标网络。因此，训练目标变为

$$q(r, \boldsymbol{s}'; \boldsymbol{w}, \boldsymbol{w}) = r + \gamma Q_{\boldsymbol{w}}(\boldsymbol{s}', \operatorname*{argmax}_{a'} Q_{\boldsymbol{w}}(\boldsymbol{s}', a')) \tag{2.38}$$

在第 3.6.2 节中，我们讨论了一种名为 **clipped double DQN** 的扩展，它使用两个 Q 网络及其冻结副本来定义以下目标:

$$q(r, \boldsymbol{s}'; \boldsymbol{w}_{1:2}, \boldsymbol{w}_{1:2}) = r + \gamma \min_{i=1,2} Q_{\boldsymbol{w}_i}(\boldsymbol{s}', \operatorname*{argmax}_{a'} Q_{\boldsymbol{w}_i}(\boldsymbol{s}', a')) \tag{2.39}$$

$Q_{\boldsymbol{w}_i}$ 是 $Q_{\boldsymbol{w}_i}$ 的目标网络。

2.5.3.3 随机集成 DQN

双 DQN 方法在 **REDQ** （随机集成双 Q 学习）方法中得到扩展，该方法由 [Che+20] 提出，使用 Q 网络的集成。此外，在每一步中，它从网络中抽取一个随机样本，并在计算目标值时取它们的平均值。也就是说，它使用以下更新（参见附录中的算法 2 ）：

$$q(r, \boldsymbol{s}'; \boldsymbol{w}_{1:N}, \boldsymbol{w}_{1:N}) = r + \gamma \max_{a'} \min_{i \in \mathcal{M}} Q_{\boldsymbol{w}_i}(\boldsymbol{s}', a') \tag{2.40}$$

其中 $\mathcal{M}$ 是从 N 值函数中随机选取的子集。集成方法可以减少方差，最小值可以减少高估偏差。[2] 如果我们设置 $N = M = 2$，我们得到一种类似于剪裁双 Q 学习的方法。（注意，REDQ 与**随机集成混合**方法非常相似，后者是为离线强化学习设计的。[ASN20], ）

2.5.4 DQN 扩展

本节中，我们讨论了 DQN 的各种扩展。

2.5.4.1 连续动作的 Q 学习

Q 学习由于需要计算动作的 argmax，因此不能直接应用于连续动作。针对此问题，在 HR 中提出了一种早期解决方案，基于神经拟合 Q 学习（参见第 2.5.2.1 节）。这成为了第 3.6.1 节中 DDPG 算法的基础，该算法学习了一种策略来预测 argmax。

一种替代方法是使用无梯度优化器，如交叉熵方法来近似 argmax。**QT-Opt** 方法将 [Kal+18] 的行动向量视为一系列动作，并逐维优化 $\boldsymbol{a}$ 。[Met+17]。**CAQL** （连续动作 Q- 学习）方法 [Ryu+20] ）使用混合整数规划来解决 argmax 问题，利用 ReLU 结构的 Q- 网络。[Sey+22] 将每个动作维度分别量化，然后使用受多智能体强化学习启发的方法解决 argmax 问题。

[2]此外，REDQ在每个环境步骤中更新值函数的$G \gg 1$；这种高**更新到数据**（UTD）比率（也称为**重放比率**）对于样本效率至关重要，并且在基于模型的强化学习中常用。

2.5.4.2 Dueling DQN

The **dueling DQN** method of [Wan+16], learns a value function and an advantage function, and derives the Q function, rather than learning it directly. This is helpful when there are many actions with similar Q-values, since the advantage $A(s,a) = Q(s,a) - V(s)$ focuses on the differences in value relative to a shared baseline.

In more detail, we define a network with $|A|+1$ output heads, which computes $A_{\boldsymbol{w}}(\boldsymbol{s}, a)$ for $a = 1 : A$ and $V_{\boldsymbol{w}}(\boldsymbol{s})$. We can then derive

$$Q_{\boldsymbol{w}}(\boldsymbol{s}, a) = V_{\boldsymbol{w}}(\boldsymbol{s}) + A_{\boldsymbol{w}}(\boldsymbol{s}, a) \tag{2.41}$$

However, this naive approach ignores the following constraint that holds for any policy π:

$$\mathbb{E}_{\pi(a|s)}\left[A^\pi(s,a)\right] = \mathbb{E}_{\pi(a|s)}\left[Q^\pi(s,a) - V^\pi(s)\right] \tag{2.42}$$

$$= V^\pi(s) - V^\pi(s) = 0 \tag{2.43}$$

Fortunately, for the optimal policy $\pi^*(s) = \text{argmax}_{a'} Q^*(s,a')$ we have

$$0 = \mathbb{E}_{\pi^*(a|s)}\left[Q^*(s,a)\right] - V^*(s) \tag{2.44}$$

$$= Q^*(s, \operatorname*{argmax}_{a'} Q^*(s,a')) - V^*(s) \tag{2.45}$$

$$= \max_{a'} Q^*(s,a') - V^*(s) \tag{2.46}$$

$$= \max_{a'} A^*(s,a') \tag{2.47}$$

Thus we can satisfy the constraint for the optimal policy by subtracting off $\max_a A(s,a)$ from the advantage head. Equivalently we can compute the Q function using

$$Q_{\boldsymbol{w}}(\boldsymbol{s}, a) = V_{\boldsymbol{w}}(\boldsymbol{s}) + A_{\boldsymbol{w}}(\boldsymbol{s}, a) - \max_{a'} A_{\boldsymbol{w}}(\boldsymbol{s}, a') \tag{2.48}$$

In practice, the max is replaced by an average, which seems to work better empirically.

2.5.4.3 Noisy nets and exploration

Standard DQN relies on the epsilon-greedy strategy to perform exploration. However, this will explore equally in all states, whereas we would like to the amount of exploration to be state dependent, to reflect the amount of uncertainty in the outcomes of trying each action in that state due to lack of knowledge (i.e., **epistemic uncertainty** rather than aleatoric or irreducile uncertainty). An early approach to this, known as **noisy nets** [For+18], added random noise to the network weights to encourage exploration which is temporally consistent within episodes. More recent methods for exploration are discussed in Section 1.4.

2.5.4.4 Multi-step DQN

As we discussed in Section 2.3.3, we can reduce the bias introduced by bootstrapping by replacing TD(1) updates with TD(n) updates, where we unroll the value computation for n MC steps, and then plug in the value function at the end. We can apply this to the DQN context by defining the target

$$q(s_0, a_0) = \sum_{t=1}^{n} \gamma^{t-1} r_t + \gamma^n \max_{a_n} Q_{\boldsymbol{w}}(s_n, a_n) \tag{2.49}$$

This can be implemented for episodic environments by storing experience tuples of the form

$$\tau = (s, a, \sum_{k=1}^{n} \gamma^{k-1} r_k, s_n, \text{done}) \tag{2.50}$$

where done $= 1$ if the trajectory ended at any point during the n-step rollout.

2.5.4.2 对抗式 DQN

Wan 的**对抗性 DQN** 方法学习一个价值函数和一个优势函数，并推导出 Q 函数，而不是直接学习它。当存在许多具有相似 Q- 值的动作时，这很有帮助，因为优势 $A(s,a) = Q(s,a) - V(s)$ 相对于一个共享基线关注价值差异。

更详细地说，我们定义了一个具有 $|A|+1$ 输出头的网络，该网络计算 $A_{\boldsymbol{w}}(\boldsymbol{s},a)$ 对于 $a = 1:\ A$ 和 $V_{\boldsymbol{w}}(\boldsymbol{s})$。然后我们可以推导出

$$Q_{\boldsymbol{w}}(\boldsymbol{s},a) = V_{\boldsymbol{w}}(\boldsymbol{s}) + A_{\boldsymbol{w}}(\boldsymbol{s},a) \tag{2.41}$$

然而，这种朴素的方法忽略了以下对于任何策略 π 都成立的约束：

$$\mathbb{E}_{\pi(a|s)}\left[A^{\pi}(s,a)\right] = \mathbb{E}_{\pi(a|s)}\left[Q^{\pi}(s,a) - V^{\pi}(s)\right] \tag{2.42}$$

$$= V^{\pi}(s) - V^{\pi}(s) = 0 \tag{2.43}$$

幸运的是，对于最优策略 $\pi^*(s) = \text{argmax}_{a'} Q^*(s,a')$，我们有

$$0 = \mathbb{E}_{\pi^*(a|s)}\left[Q^*(s,a)\right] - V^*(s) \tag{2.44}$$

$$= Q^*(s, \underset{a'}{\text{argmax}}\, Q^*(s,a')) - V^*(s) \tag{2.45}$$

$$= \max_{a'} Q^*(s,a') - V^*(s) \tag{2.46}$$

$$= \max_{a'} A^*(s,a') \tag{2.47}$$

因此，我们可以通过从优势头中减去 $\max_a A(s,a)$ 来满足最优策略的约束。等价地，我们可以使用 Q 函数来计算。

$$Q_{\boldsymbol{w}}(\boldsymbol{s},a) = V_{\boldsymbol{w}}(\boldsymbol{s}) + A_{\boldsymbol{w}}(\boldsymbol{s},a) - \max_{a'} A_{\boldsymbol{w}}(\boldsymbol{s},a') \tag{2.48}$$

在实践中，最大值被平均数所取代，这在经验上似乎效果更好。

2.5.4.3 噪声网络和探索

标准 DQN 依赖于 ε- 贪婪策略进行探索。然而，这将在所有状态下进行同等探索，而我们所希望的是探索量与状态相关，以反映由于缺乏知识而尝试该状态下每个动作的结果的不确定性（即，**认识论不确定性** 而不是随机或不可减少的不确定性）。对此的一个早期方法，称为**噪声网络** [For+18]，向网络权重添加随机噪声以鼓励探索，这种探索在剧集内是时间上一致的。更近期的探索方法在第 1.4 节中讨论。

2.5.4.4 多步 DQN

如我们在第 2.3.3 节中讨论的那样，我们可以通过用 TD(n) 更新替换 TD(1) 更新来减少由自举引入的偏差，其中我们展开 n MC 步的价值计算，然后在最后插入价值函数。我们可以通过定义目标将此应用于 DQN 环境。

$$q(s_0,a_0) = \sum_{t=1}^{n} \gamma^{t-1} r_t + \gamma^n \max_{a_n} Q_{\boldsymbol{w}}(s_n,a_n) \tag{2.49}$$

这可以通过存储形式为的经验元组来实现，用于情境式环境

$$\tau = (s, a, \sum_{k=1}^{n} \gamma^{k-1} r_k, s_n, \text{done}) \tag{2.50}$$

where done $= 1$ 如果在 n- 步的任何时刻轨迹结束 rollout.

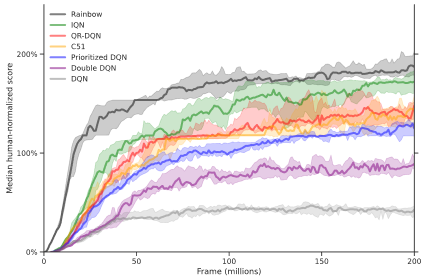


Figure 2.9: Plot of median human-normalized score over all 57 Atari games for various DQN agents. The yellow, red and green curves are distributional RL methods (Section 5.1), namely categorical DQN (C51) (Section 5.1.2) Quantile Regression DQN (Section 5.1.1), and Implicit Quantile Networks [Dab+18]. Figure from https://github.com/google-deepmind/dqn_zoo.

Theoretically this method is only valid if all the intermediate actions, $a_{2:n-1}$, are sampled from the current optimal policy derived from $Q_{\boldsymbol{w}}$, as opposed to some behavior policy, such as epsilon greedy or some samples from the replay buffer from an old policy. In practice, we can just restrict sampling to recent samples from the replay buffer, making the resulting method approximately on-policy.

Instead of using a fixed n, it is possible to use a weighted combination of returns; this is known as the $Q(\lambda)$ algorithm [PW94; Koz+21].

2.5.4.5 Rainbow

The **Rainbow** method of [Hes+18] combined 6 improvements to the vanilla DQN method, as listed below. (The paper is called "Rainbow" due to the color coding of their results plot, a modified version of which is shown in Figure 2.9.) At the time it was published (2018), this produced SOTA results on the Atari-200M benchmark. The 6 improvements are as follows:

- Use double DQN, as in Section 2.5.3.2.
- Use prioritized experience replay, as in Section 2.5.2.3.
- Use the categorical DQN (C51) (Section 5.1.2) distributional RL method.
- Use n-step returns (with $n = 3$), as in Section 2.5.4.4.
- Use dueling DQN, as in Section 2.5.4.2.
- Use noisy nets, as in Section 2.5.4.3.

Each improvement gives diminishing returns, as can be see in Figure 2.9.

Recently the "Beyond the Rainbow" paper [Unk24] proposed several more extensions:

- Use a larger CNN with residual connections, namely the Impala network from [Esp+18] with the modifications (including the use of spectral normalization) proposed in [SS21].
- Replace C51 with Implicit Quantile Networks [Dab+18].
- Use **Munchausen RL** [VPG20], which modifies the Q learning update rule by adding an entropy-like penalty.
- Collect 1 environment step from 64 parallel workers for each minibatch update (rather than taking many steps from a smaller number of workers).

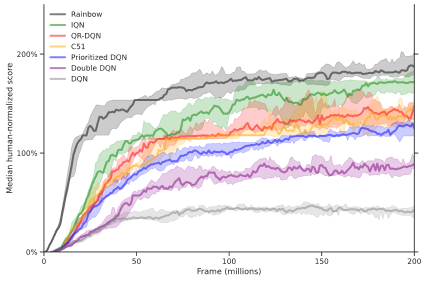


图 2.9：各种 DQN 代理在所有 57 款 Atari 游戏上的中值人类归一化分数图。黄色、红色和绿色曲线是分布式强化学习（第 *5.1* 节），即分类 DQN （ C51 ）（第 *5.1.2* 节）、分位数回归 DQN （第 *5.1.1* 节）和隐式分位数网络 *[Dab+18]*。图来自 *https://github.com/google-deepmind/dqn_zoo*。

理论上，这种方法仅在所有中间动作 $a_{2:n-1}$ 都从基于 $Q_{\boldsymbol{w}}$ 的当前最优策略中采样，而不是从某些行为策略（如 ε 贪婪或从旧策略的重放缓冲区中的一些样本）中采样时有效。在实践中，我们只需将采样限制为重放缓冲区中的最近样本，从而使该方法近似于在线策略。

而不是使用固定的 n，可以使用回报的加权组合；这被称为 $Q(\lambda)$ 算法 [PW94 ； Koz+21].

2.5.4.5 彩虹

彩虹方法对原始 DQN 方法进行了 6 项改进，如下所示。（该论文被称为 “ 彩虹 ”，因为其结果图的彩色编码，如图 2.9 所示。）在 2018 年发表时，它在 Atari-200M 基准测试上产生了 SOTA 结果。6 项改进如下：

- 使用双 DQN，如第 2.5.3.2 节所述。• 使用优先级经验回放，如第 2.5.2.3 节所述。

使用分类 DQN （ C51 ）（第 5.1.2 节）分布式强化学习方法。

- 使用 n 步返回（带有 $n = 3$ ），如第 2.5.4.4. 节所述。
- 使用对抗性 DQN，如第 2.5.4.2 节所述。
- 使用有噪声的神经网络，如第 2.5.4.3 节所述。

每次改进都带来递减的回报，如图 2.9 所示。

最近，“ 超越彩虹 ” 论文 [未知 24] 提出了几个更多扩展：

- 使用具有残差连接的大 CNN，即来自 [Esp+18] 的 Impala 网络，并采用 [SS21] 中提出的修改（包括使用谱归一化）。• 将 C51 替换为隐式分位数网络 [Dab+18]。• 使用 **Munchausen RL**[VPG20], , 该网络通过添加熵惩罚来修改 Q 学习更新规则。• 对于每个小批量更新，从 64 个并行工作者中收集 1 个环境步骤（而不是从较少的工作者那里采取许多步骤）。

2.5.4.6 Bigger, Better, Faster

At the time of writing this document (2024), the SOTA on the 100k sample-efficient Atari benchmark [Kai+19] is obtained by the **BBF** algorithm of [Sch+23b]. (BBF stands for "Bigger, Better, Faster".) It uses the following tricks, in order of decreasing importance:

- Use a larger CNN with residual connections, namely a modified version of the Impala network from [Esp+18].
- Increase the **update-to-data** (UTD) ratio (number of times we update the Q function for every observation that is observed), in order to increase sample efficiency [HHA19].
- Use a periodic soft reset of (some of) the network weights to avoid loss of elasticity due to increased network updates, following the **SR-SPR** method of [D'O+22].
- Use n-step returns, as in Section 2.5.4.4, and then gradually decrease (anneal) the n-step return from $n = 10$ to $n = 3$, to reduce the bias over time.
- Add weight decay.
- Add a self-predictive representation loss (Section 4.3.2.2) to increase sample efficiency.
- Gradually increase the discount factor from $\gamma = 0.97$ to $\gamma = 0.997$, to encourage longer term planning once the model starts to be trained.[3]
- Drop noisy nets (which requires multiple network copies and thus slows down training due to increased memory use), since it does not help.
- Use dueling DQN (see Section 2.5.4.2).
- Use distributional DQN (see Section 5.1).

2.5.4.7 Other methods

Many other methods have been proposed to reduce the sample complexity of value-based RL while maintaining performance, see e.g., the **MEME** paper of [Kap+22].

[3] The **Agent 57** method of [Bad+20] automatically learns the exploration rate and discount factor using a multi-armed bandit stratey, which lets it be more exploratory or more exploitative, depending on the game. This resulted in super human performance on all 57 Atari games in ALE. However, it required 80 billion frames (environment steps)! This was subsequently reduced to the "standard" 200M frames in the **MEME** method of [Kap+22].

2.5.4.6 更大、更好、更快

截至撰写本文（2024 年），在 100k 样本高效的 Atari 基准测试中，SOTA （最先进的技术）是通过 [Kai+19] 获得的 **BBF** 算法 [Sch+23b]。（BBF 代表 “ 更大、更好、更快 ”。）它使用以下技巧，按重要性递减的顺序：

增加**更新到数据**（UTD）[illegible] • 使用具有残差连接的 CNN [illegible] Impala [illegible] 样本效率 [HHA19]。• 使用（某些）网络权重的周期性软重置，以避免由于网络更新增加而导致的弹性损失，遵循 **SR-SPR** 方法 [D'O+22]。

到 $n = 3$，以减少时间差分偏差 [illegible] • 使用 [illegible] 2.5.4.4 [illegible] 4.3.2.2 [illegible] 以提高样本效率。• 逐渐增加折扣因子从 $\gamma = 0.97$ 到 $\gamma = 0.997$，以鼓励模型开始训练后进行长期规划。[3] • 丢弃噪声网络（这需要多个网络副本，因此由于内存使用增加而减慢训练速度），因为它没有帮助。• 使用对抗 DQN （见第 2.5.4.2 节）。• 使用分布式 DQN （见第 5.1 节）。

2.5.4.7 其他方法

许多其他方法已被提出，以在保持性能的同时降低基于价值的强化学习（ RL ）的样本复杂度，例如，参见 **MEME** 论文 [Kap+22]。

[3] The **代理57**方法通过使用多臂老虎机策略自动学习探索率和折扣因子，使其在游戏中更具探索性或更具剥削性，这导致了在ALE 中所有 57 款 Atari 游戏上的超人类表现。然而，这需要 800 亿帧（环境步骤）！这随后被减少到 **MEME** 方法中的 “ 标准 ”2 亿帧 [Kap+22]。

Chapter 3

Policy-based RL

In the previous section, we considered methods that estimate the action-value function, $Q(s, a)$, from which we derive a policy. However, these methods have several disadvantages: (1) they can be difficult to apply to continuous action spaces; (2) they may diverge if function approximation is used (see Section 2.5.2.4); (3) the training of Q, often based on TD-style updates, is not directly related to the expected return garnered by the learned policy; (4) they learn deterministic policies, whereas in stochastic and partially observed environments, stochastic policies are provably better [JSJ94].

In this section, we discuss **policy search** methods, which directly optimize the parameters of the policy so as to maximize its expected return. We mostly focus on **policy gradient** methods, that use the gradient of the loss to guide the search. As we will see, these policy methods often benefit from estimating a value or advantage function to reduce the variance in the policy search process, so we will also use techniques from Chapter 2. The parametric policy will be denoted by $\pi_{\boldsymbol{\theta}}(a|s)$. For discrete actions, this can be a DNN with a softmax output. For continuous actions, we can use a Gaussian output layer, or a diffusion policy [Ren+24].

For more details on policy gradient methods, see [Wen18b; Leh24].

3.1 The policy gradient theorem

We start by defining the objective function for policy learning, and then derive its gradient. The objective, which we aim to maximize, is defined as

$$J(\pi) \triangleq \mathbb{E}_\pi \left[\sum_{t=0}^{\infty} \gamma^t R_{t+1} \right] \tag{3.1}$$

$$= \sum_{t=0}^{\infty} \gamma^t \sum_s \left(\sum_{s_0} p_0(s_0) p^\pi(s_0 \to s, t) \right) \sum_a \pi(a|s) R(s, a) \tag{3.2}$$

$$= \sum_s \left(\sum_{s_0} \sum_{t=0}^{\infty} \gamma^t p_0(s_0) p^\pi(s_0 \to s, t) \right) \sum_a \pi(a|s) R(s, a) \tag{3.3}$$

$$= \sum_s \rho^\pi(s) \sum_a \pi(a|s) R(s, a) \tag{3.4}$$

where we have defined the discounted state visitation measure

$$\rho_\pi^\gamma(s) \triangleq \sum_{t=0}^{\infty} \gamma^t \underbrace{\sum_{s_0} p_0(s_0) p^\pi(s_0 \to s, t)}_{p_t^\pi(s)} \tag{3.5}$$

第三章

基于策略的强化学习

在前一节中，我们考虑了估计动作值函数的方法，$Q(s,a)$，从中我们推导出一个策略。然而，这些方法存在几个缺点：(1) 它们可能难以应用于连续动作空间； (2) 如果使用函数逼近，它们可能会发散（参见第 2.5.2.4 节）； (3)Q 的训练，通常基于 TD 风格的更新，与通过学习策略获得的预期回报没有直接关系； (4) 它们学习确定性策略，而在随机和部分观察环境中，随机策略已被证明更好 [JSJ94]。

在本节中，我们讨论**策略搜索**方法，这些方法直接优化策略的参数以最大化其期望回报。我们主要关注**策略梯度**方法，这些方法使用损失函数的梯度来引导搜索。正如我们将看到的，这些策略方法通常从估计值函数或优势函数中受益，以减少策略搜索过程中的方差，因此我们还将使用第 2 章的技术。参数化策略将表示为 $\pi_{\boldsymbol{\theta}}(a|s)$。对于离散动作，这可以是一个具有 softmax 输出的 DNN。对于连续动作，我们可以使用高斯输出层，或者使用 [Ren+24] 策略。

关于策略梯度方法的更多细节，请参阅 [Wen18b； Leh24]。

3.1 政策梯度定理

我们首先定义策略学习的目标函数，然后推导其梯度。我们旨在最大化的目标定义为

$$J(\pi) \triangleq \mathbb{E}_\pi \left[\sum_{t=0}^{\infty} \gamma^t R_{t+1} \right] \tag{3.1}$$

$$= \sum_{t=0}^{\infty} \gamma^t \sum_s \left(\sum_{s_0} p_0(s_0) p^\pi(s_0 \to s, t) \right) \sum_a \pi(a|s) R(s,a) \tag{3.2}$$

$$= \sum_s \left(\sum_{s_0} \sum_{t=0}^{\infty} \gamma^t p_0(s_0) p^\pi(s_0 \to s, t) \right) \sum_a \pi(a|s) R(s,a) \tag{3.3}$$

$$= \sum_s \rho^\pi(s) \sum_a \pi(a|s) R(s,a) \tag{3.4}$$

我们在其中定义了折现状态访问度量

$$\rho_\pi^\gamma(s) \triangleq \sum_{t=0}^{\infty} \gamma^t \underbrace{\sum_{s_0} p_0(s_0) p^\pi(s_0 \to s, t)}_{p_t^\pi(s)} \tag{3.5}$$

where $p^\pi(s_0 \to s, t)$ is the probability of going from s_0 to s in t steps, and $p_t^\pi(s)$ is the marginal probability of being in state s at time t (after each episodic reset). Note that ρ_π^γ is a measure of time spent in non-terminal states, but it is not a probability measure, since it is not normalized, i.e., $\sum_s \rho_\pi^\gamma(s) \neq 1$. However, we may abuse notation and still treat it like a probability, so we can write things like

$$\mathbb{E}_{\rho_\pi^\gamma(s)}\left[f(s)\right] = \sum_s \rho_\pi^\gamma(s) f(s) \tag{3.6}$$

Using this notation, we can define the objective as

$$J(\pi) = \mathbb{E}_{\rho_\pi^\gamma(s), \pi(a|s)}\left[R(s,a)\right] \tag{3.7}$$

We can also define a normalized version of the measure ρ by noting that $\sum_{t=0}^\infty \gamma^t = \frac{1}{1-\gamma}$ for $\gamma < 1$. Hence the normalized discounted state visitation distribution is given by

$$p_\pi^\gamma(s) = (1-\gamma)\rho_\pi^\gamma(s) = (1-\gamma)\sum_{t=0}^\infty \gamma^t p_t(s) \tag{3.8}$$

(Note the change from ρ to p.)

Note that in [SB18, Sec 13.2], they use slightly different notation. In particular, they assume $\gamma = 1$, and define the non-discounted state visitation measure as $\eta(s)$ and the corresponding normalized version by $\mu(s)$. This is equivalent to ignoring the discount factor γ^t when defining $\rho_\pi(s)$. This is standard practice in many implementations, since we can just average over (unweighted) trajectories when estimating the objective and its gradient, even though it results in a biased estimate [NT20; CVRM23].

It can be shown that the gradient of the above objective is given by

$$\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) = \sum_s \rho_\pi^\gamma(s) \sum_a Q^\pi(s,a) \nabla_{\boldsymbol{\theta}} \pi_{\boldsymbol{\theta}}(a|s) \tag{3.9}$$

$$= \sum_s \rho_\pi^\gamma(s) \sum_a Q^{\pi_{\boldsymbol{\theta}}}(s,a) \pi_{\boldsymbol{\theta}}(a|s) \nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(a|s) \tag{3.10}$$

$$= \mathbb{E}_{\rho_\pi^\gamma(s)\pi_{\boldsymbol{\theta}}(a|s)}\left[Q^{\pi_{\boldsymbol{\theta}}}(s,a) \nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(a|s)\right] \tag{3.11}$$

This is known as the **policy gradient theorem** [Sut+99]. In statistics, the term $\nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(\boldsymbol{a}|\boldsymbol{s})$ is called the (Fisher) **score function**[1], so sometimes Equation (3.11) is called the **score function estimator** or **SFE** [Fu15; Moh+20].

3.2 REINFORCE

One way to apply the policy gradient theorem to optimize a policy is to use stochastic gradient ascent. Theoretical results concerning the convergence and sample complexity of such methods can be found in [Aga+21a].

To implement such a method, let $\boldsymbol{\tau} = (s_0, a_0, r_0, s_1, \ldots, s_T)$ be a trajectory created by sampling from $s_0 \sim p_0$ and then following $\pi_{\boldsymbol{\theta}}$. Then we have

$$\nabla_{\boldsymbol{\theta}} J(\pi_{\boldsymbol{\theta}}) = \sum_{t=0}^\infty \gamma^t \mathbb{E}_{p_t(s)\pi_{\boldsymbol{\theta}}(a_t|s_t)}\left[\nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(a_t|s_t) Q_{\pi_{\boldsymbol{\theta}}}(s_t, a_t)\right] \tag{3.12}$$

$$\approx \sum_{t=0}^{T-1} \gamma^t G_t \nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(a_t|s_t) \tag{3.13}$$

[1] This is distinct from the Stein score, which is the gradient wrt the argument of the log probability, $\nabla_{\boldsymbol{a}} \log \pi_{\boldsymbol{\theta}}(\boldsymbol{a}|\boldsymbol{s})$, as used in diffusion.

$p^\pi(s_0 \to s,t)$ 是在 t 步内从 s_0 到 s 的概率，而 $p_t^\pi(s)$ 是在时间 t （每次情景重置后）处于状态 s 的边缘概率。请注意，ρ_π^γ 是在非终端状态花费时间的度量，但它不是一个概率度量，因为它没有归一化，即 $\sum_s \rho_\pi^\gamma(s) \neq 1$。然而，我们可以滥用符号，仍然将其视为概率，因此我们可以写出类似的事物。

$$\mathbb{E}_{\rho_\pi^\gamma(s)}\left[f(s)\right] = \sum_s \rho_\pi^\gamma(s) f(s) \tag{3.6}$$

使用这种表示法，我们可以定义目标为

$$J(\pi) = \mathbb{E}_{\rho_\pi^\gamma(s),\pi(a|s)}\left[R(s,a)\right] \tag{3.7}$$

我们还可以定义一个测度 ρ 的标准化版本，即注意到对于 $\gamma < 1$，有 $\sum_{t=0}^\infty \gamma^t = {}^1_{1-\gamma}$。因此，标准化折现状态访问分布由以下给出:

$$p_\pi^\gamma(s) = (1-\gamma)\rho_\pi^\gamma(s) = (1-\gamma)\sum_{t=0}^\infty \gamma^t p_t(s) \tag{3.8}$$

（注意从 ρ 到 p 的变化。）

请注意，在 [SB18 的第 13 节中，他们使用了略微不同的符号。特别是，他们假设 $\gamma = 1$，并将非折扣状态访问度量定义为 $\eta(s)$ ，以及相应的归一化版本为 $\mu(s)$。这相当于在定义 () 时忽略了折扣因子 γ^t 。这在许多实现中是标准做法，因为我们可以在估计目标及其梯度时对（未加权的）轨迹进行平均，尽管这会导致有偏估计 [NT20； CVRM23]。它可以

e 表明上述目标函数的梯度是给定的 通过

$$\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) = \sum_s \rho_\pi^\gamma(s) \sum_a Q^\pi(s,a) \nabla_{\boldsymbol{\theta}} \pi_{\boldsymbol{\theta}}(a|s) \tag{3.9}$$

$$= \sum_s \rho_\pi^\gamma(s) \sum_a Q^{\pi_{\boldsymbol{\theta}}}(s,a) \pi_{\boldsymbol{\theta}}(a|s) \nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(a|s) \tag{3.10}$$

$$= \mathbb{E}_{\rho_\pi^\gamma(s)\pi_{\boldsymbol{\theta}}(a|s)}\left[Q^{\pi_{\boldsymbol{\theta}}}(s,a) \nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(a|s)\right] \tag{3.11}$$

这被称为**策略梯度定理** [Sut+99]。在统计学中，术语 $\nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(\boldsymbol{a}|\boldsymbol{s})$ 被称为（ Fisher ）**得分函数** [1]，因此有时方程（ 3.11 ）被称为**得分函数估计量**或 **SFE**[Fu15； Moh+20]。

3.2 强化

一种将策略梯度定理应用于策略优化的方法是通过使用随机梯度上升。关于此类方法收敛性和样本复杂度的理论结果可以在 [Aga+21a] 中找到。

为了实现这种方法，设 $\boldsymbol{\tau} = (s_0, a_0, r_0, s_1, \ldots, s_T)$ 为由从 $s_0 \sim p_0$ 采样并遵循 $\pi_{\boldsymbol{\theta}}$ 生成的轨迹。然后我们有

$$\nabla_{\boldsymbol{\theta}} J(\pi_{\boldsymbol{\theta}}) = \sum_{t=0}^\infty \gamma^t \mathbb{E}_{p_t(s)\pi_{\boldsymbol{\theta}}(a_t|s_t)}\left[\nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(a_t|s_t) Q_{\pi_{\boldsymbol{\theta}}}(s_t,a_t)\right] \tag{3.12}$$

$$\approx \sum_{t=0}^{T-1} \gamma^t G_t \nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(a_t|s_t) \tag{3.13}$$

[1]这与Stein分数不同，Stein分数是关于对数概率的参数的梯度，$\nabla_{\boldsymbol{a}} \log \pi_{\boldsymbol{\theta}}(\boldsymbol{a}|\boldsymbol{s})$，在扩散中的应用。

where the return is defined as follows

$$G_t \triangleq r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \cdots + \gamma^{T-t-1} r_{T-1} = \sum_{k=0}^{T-t-1} \gamma^k r_{t+k} = \sum_{j=t}^{T-1} \gamma^{j-t} r_j \tag{3.14}$$

See Algorithm 3 for the pseudocode.

Algorithm 3: REINFORCE (episodic version)

```
Initialize policy parameters θ
repeat
    Sample an episode τ = (s_0, a_0, r_0, s_1, ..., s_T) using π_θ
    for t = 0, 1, ..., T − 1 do
        G_t = Σ_{k=t+1}^{T} γ^{k−t−1} R_k
        θ ← θ + η_θ γ^t G_t ∇_θ log π_θ(a_t|s_t)
until converged
```

In practice, estimating the policy gradient using Equation (3.11) can have a high variance. A **baseline** function $b(s)$ can be used for variance reduction to get

$$\nabla_{\boldsymbol{\theta}} J(\pi_{\boldsymbol{\theta}}) = \mathbb{E}_{\rho_{\boldsymbol{\theta}}(s)\pi_{\boldsymbol{\theta}}(a|s)} \left[\nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(a|s)(Q_{\pi_{\boldsymbol{\theta}}}(s,a) - b(s))\right] \tag{3.15}$$

Any function that satisfies $\mathbb{E}\left[\nabla_{\boldsymbol{\theta}} b(s)\right] = 0$ is a valid baseline. This follows since

$$\sum_a \nabla_{\boldsymbol{\theta}} \pi_{\boldsymbol{\theta}}(a|s)(Q(s,a) - b(s)) = \nabla_{\boldsymbol{\theta}} \sum_a \pi_{\boldsymbol{\theta}}(a|s) Q(s,a) - \nabla_{\boldsymbol{\theta}} [\sum_a \pi_{\boldsymbol{\theta}}(a|s)] b(s) = \nabla_{\boldsymbol{\theta}} \sum_a \pi_{\boldsymbol{\theta}}(a|s) Q(s,a) - 0 \tag{3.16}$$

A common choice for the baseline is $b(s) = V_{\pi_{\boldsymbol{\theta}}}(s)$. This is a good choice since $V_{\pi_{\boldsymbol{\theta}}}(s)$ and $Q(s,a)$ are correlated and have similar magnitudes, so the scaling factor in front of the gradient term will be small.

Using this we get an update of the following form

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \eta \sum_{t=0}^{T-1} \gamma^t (G_t - b(s_t)) \nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(a_t|s_t) \tag{3.17}$$

This is is called the **REINFORCE** estimator [Wil92].[2] The update equation can be interpreted as follows: we compute the sum of discounted future rewards induced by a trajectory, compared to a baseline, and if this is positive, we increase $\boldsymbol{\theta}$ so as to make this trajectory more likely, otherwise we decrease $\boldsymbol{\theta}$. Thus, we reinforce good behaviors, and reduce the chances of generating bad ones.

3.3 Actor-critic methods

An **actor-critic** method [BSA83] uses the policy gradient method, but where the expected return G_t is estimated using temporal difference learning of a value function instead of MC rollouts. (The term "actor" refers to the policy, and the term "critic" refers to the value function.) The use of bootstrapping in TD updates allows more efficient learning of the value function compared to MC, and further reduces the variance. In addition, it allows us to develop a fully online, incremental algorithm, that does not need to wait until the end of the trajectory before updating the parameters.

[2] The term "REINFORCE" is an acronym for "REward Increment = nonnegative Factor x Offset Reinforcement x Characteristic Eligibility". The phrase "characteristic eligibility" refers to the $\nabla \log \pi_{\boldsymbol{\theta}}(a_t|s_t)$ term; the phrase "offset reinforcement" refers to the $G_t - b(s_t)$ term; and the phrase "nonnegative factor" refers to the learning rate η of SGD.

在哪里返回被定义为如下

$$G_t \triangleq r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \cdots + \gamma^{T-t-1} r_{T-1} = \sum_{k=0}^{T-t-1} \gamma^k r_{t+k} = \sum_{j=t}^{T-1} \gamma^{j-t} r_j \tag{3.14}$$

查看算法 3 的伪代码。

算法 3： REINFORCE （时序版本）

初始化策略参数 $\boldsymbol{\theta}$ **重复** 采样一个片段 $\boldsymbol{\tau} = (s_0, a_0, r_0, s_1, \ldots, s_T)$ 使用 $\pi_{\boldsymbol{\theta}}$ **对于** $t = 0, 1, \ldots, T-1$ **做** $G_t = \sum_{k=t+1}^{T} \gamma^{k-t-1} R_k$ $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \eta_{\boldsymbol{\theta}} \gamma^t G_t \nabla_{\boldsymbol{\theta}}$ 记录 $\pi_{\boldsymbol{\theta}}(a_t|s_t)$ **直到**收敛

在实践中，使用公式（ 3.11 ）估计策略梯度可能会有很高的方差。可以使用一个**基线**函数 b （ s ）来降低方差，以获得

$$\nabla_{\boldsymbol{\theta}} J(\pi_{\boldsymbol{\theta}}) = \mathbb{E}_{\rho_{\boldsymbol{\theta}}(s)\pi_{\boldsymbol{\theta}}(a|s)} \left[\nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(a|s)(Q_{\pi_{\boldsymbol{\theta}}}(s,a) - b(s))\right] \tag{3.15}$$

任何满足 $\mathbb{E}\left[\nabla_{\boldsymbol{\theta}} b(s)\right] = 0$ 的函数都是一个有效基线。这符合以下情况：

$$\sum_a \nabla_{\boldsymbol{\theta}} \pi_{\boldsymbol{\theta}}(a|s)(Q(s,a) - b(s)) = \nabla_{\boldsymbol{\theta}} \sum_a \pi_{\boldsymbol{\theta}}(a|s) Q(s,a) - \nabla_{\boldsymbol{\theta}} [\sum_a \pi_{\boldsymbol{\theta}}(a|s)] b(s) = \nabla_{\boldsymbol{\theta}} \sum_a \pi_{\boldsymbol{\theta}}(a|s) Q(s,a) - 0 \tag{3.16}$$

基线的一个常见选择是 $b(s) = V_{\pi_{\boldsymbol{\theta}}}(s)$。这是一个不错的选择，因为 $V_{\pi_{\boldsymbol{\theta}}}(s)$ 和 $Q(s,a)$ 相关且具有相似的大小，因此梯度项前面的缩放因子将很小。

使用此方法，我们得到以下形式的更新

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \eta \sum_{t=0}^{T-1} \gamma^t (G_t - b(s_t)) \nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(a_t|s_t) \tag{3.17}$$

这被称为 **REINFORCE** 估计器 [Wil92]。[2] 更新方程可以解释如下：我们计算由轨迹引起的折扣未来奖励的总和，与基线相比，如果这是正的，我们就增加 $\boldsymbol{\theta}$ ，以便使这条轨迹更有可能，否则我们减少 $\boldsymbol{\theta}$。因此，我们强化良好行为，并减少产生不良行为的可能性。

3.3 演员 - 评论家方法

一个 **actor-critic** 方法 [BSA83] 使用策略梯度方法，但预期回报 G_t 是通过值函数的时间差学习来估计，而不是通过 MC 回滚。（术语 “actor” 指的是策略，“critic” 指的是值函数。）在 TD 更新中使用自举允许比 MC 更有效地学习值函数，并进一步减少方差。此外，它还允许我们开发一个完全在线的增量算法，不需要等待直到轨迹的末尾才更新参数。

[2] “REINFORCE”这个术语是“REward Increment nonnegative Factor x Offset Reinforcement x Characteristic Eligibility”的缩写。短语 “ 特征资格 ” 指的是 $\nabla \log \pi_{\boldsymbol{\theta}}(a_t|s_t)$ 项；短语 “ 偏置强化 ” 指的是 $G_t - b(s_t)$ 项；而短语 “ 非负因子 ” 指的是 SGD 的学习率 η 。

3.3.1 Advantage actor critic (A2C)

Concretely, consider the use of the one-step TD method to estimate the return in the episodic case, i.e., we replace G_t with $G_{t:t+1} = r_t + \gamma V_{\boldsymbol{w}}(s_{t+1})$. If we use $V_{\boldsymbol{w}}(s_t)$ as a baseline, the REINFORCE update in Equation (3.17) becomes

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \eta \sum_{t=0}^{T-1} \gamma^t \left(G_{t:t+1} - V_{\boldsymbol{w}}(s_t)\right) \nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(a_t|s_t) \tag{3.18}$$

$$= \boldsymbol{\theta} + \eta \sum_{t=0}^{T-1} \gamma^t \big(r_t + \gamma V_{\boldsymbol{w}}(s_{t+1}) - V_{\boldsymbol{w}}(s_t)\big) \nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(a_t|s_t) \tag{3.19}$$

Note that $\delta_t = r_{t+1} + \gamma V_{\boldsymbol{w}}(s_{t+1}) - V_{\boldsymbol{w}}(s_t)$ is a single sample approximation to the advantage function $A(s_t, a_t) = Q(s_t, a_t) - V(s_t)$. This method is therefore called **advantage actor critic** or **A2C**. See Algorithm 4 for the pseudo-code.[3] (Note that $V_{\boldsymbol{w}}(s_{t+1}) = 0$ if s_t is a done state, representing the end of an episode.) Note that this is an on-policy algorithm, where we update the value function $V_{\boldsymbol{w}}^{\pi}$ to reflect the value of the current policy π. See Section 3.3.3 for further discussion of this point.

Algorithm 4: Advantage actor critic (A2C) algorithm (episodic)

Initialize actor parameters $\boldsymbol{\theta}$, critic parameters $\boldsymbol{w}$
repeat
  Sample starting state s_0 of a new episode
  for $t = 0, 1, 2, \ldots$ **do**
    Sample action $a_t \sim \pi_{\boldsymbol{\theta}}(\cdot|s_t)$
    $(s_{t+1}, r_t, \text{done}_t) = \text{env.step}(s_t, a_t)$
    $q_t = r_t + \gamma(1 - \text{done}_t) V_{\boldsymbol{w}}(s_{t+1})$ // Target
    $\delta_t = q_t - V_{\boldsymbol{w}}(s_t)$ // Advantage
    $\boldsymbol{w} \leftarrow \boldsymbol{w} + \eta_{\boldsymbol{w}} \delta_t \nabla_{\boldsymbol{w}} V_{\boldsymbol{w}}(s_t)$ // Critic
    $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \eta_{\boldsymbol{\theta}} \gamma^t \delta_t \nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(a_t|s_t)$ // Actor
    if $done_t = 1$ **then**
      break
until *converged*

In practice, we should use a stop-gradient operator on the target value for the TD update, for reasons explained in Section 2.5.2.4. Furthermore, it is common to add an entropy term to the policy, to act as a regularizer (to ensure the policy remains stochastic, which smoothens the loss function — see Section 3.5.4). If we use a shared network with separate value and policy heads, we need to use a single loss function for training all the parameters $\boldsymbol{\phi}$. Thus we get the following loss, for each trajectory, where we want to minimize TD loss, maximize the policy gradient (expected reward) term, and maximize the entropy term.

$$\mathcal{L}(\boldsymbol{\phi}; \tau) = \frac{1}{T} \sum_{t=1}^{T} \left[\lambda_{TD} \mathcal{L}_{TD}(s_t, a_t, r_t, s_{t+1}) - \lambda_{PG} J_{PG}(s_t, a_t, r_t, s_{t+1}) - \lambda_{ent} J_{ent}(s_t)\right] \tag{3.20}$$

$$q_t = r_t + \gamma(1 - \text{done}(s_t)) V_{\boldsymbol{\phi}}(s_{t+1}) \tag{3.21}$$

$$\mathcal{L}_{TD}(s_t, a_t, r_t, s_{t+1}) = (\text{sg}(q_t) - V_{\phi}(s))^2 \tag{3.22}$$

$$J_{PG}(s_t, a_t, r_t, s_{t+1}) = (\text{sg}(q_t - V_{\phi}(s_t)) \log \pi_{\boldsymbol{\phi}}(a_t|s_t) \tag{3.23}$$

$$J_{ent}(s_t) = -\sum_a \pi_{\boldsymbol{\phi}}(a|s_t) \log \pi_{\boldsymbol{\phi}}(a|s_t) \tag{3.24}$$

[3] In [Mni+16], they proposed a distributed version of A2C known as **A3C** which stands for "asynchrononous advantage actor critic".

3.3.1 优势演员评论员（A2C）

具体而言，考虑在连续案例中使用一步 TD 方法来估计回报，即我们将 G_t 替换为 $G_{t:t+1} = r_t + \gamma V_{\boldsymbol{w}}(s_{t+1})$。如果我们使用 $V_{\boldsymbol{w}}(s_t)$ 作为基线，那么方程（3.17）中的 REINFORCE 更新变为

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \eta \sum_{t=0}^{T-1} \gamma^t \left(G_{t:t+1} - V_{\boldsymbol{w}}(s_t)\right) \nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(a_t|s_t) \tag{3.18}$$

$$= \boldsymbol{\theta} + \eta \sum_{t=0}^{T-1} \gamma^t \big(r_t + \gamma V_{\boldsymbol{w}}(s_{t+1}) - V_{\boldsymbol{w}}(s_t)\big) \nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(a_t|s_t) \tag{3.19}$$

请注意，$\delta_t = r_{t+1} + \gamma V_{\boldsymbol{w}}(s_{t+1}) - V_{\boldsymbol{w}}(s_t)$ 是优势函数 $A(s_t, a_t) = Q(s_t, a_t) - V(s_t)$ 的单样本近似。因此，这种方法被称为**优势演员评论家**或 **A2C**。请参阅算法 4 以获取伪代码。[3]（请注意，$V_{\boldsymbol{w}}(s_{t+1}) = 0$ 如果 s_t 是完成状态，表示一局的结束。）请注意，这是一个基于策略的算法，其中我们更新值函数 $V_{\boldsymbol{w}}^{\pi}$ 以反映当前策略 π 的价值。请参阅第 3.3.3 节以进一步讨论此点。

算法 4： 优势演员评论员（A2C）算法（分集）

初始化演员参数 $\boldsymbol{\theta}$，评论家参数 $\boldsymbol{w}$
重复
采样新剧集的起始状态 s_0
对于 $t = 0, 1, 2, \ldots$ **做**
采样动作 $a_t \sim \pi_{\boldsymbol{\theta}}(\cdot|s_t)$
$(s_{t+1}, r_t, 完成_t) = env.step(s_t, a_t)$
$q_t = r_t + \gamma(1 - 完成_t) V_{\boldsymbol{w}}(s_{t+1})$ // 目标
$\delta_t = q_t - V_{\boldsymbol{w}}(s_t)$ // 势
$\boldsymbol{w} \leftarrow \boldsymbol{w} + \eta_{\boldsymbol{w}} \delta_t \nabla_{\boldsymbol{w}} V_{\boldsymbol{w}}(s_t)$ // 评论家
$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \eta_{\boldsymbol{\theta}} \gamma^t \delta_t \nabla_{\boldsymbol{\theta}}$ 记录 $\pi_{\boldsymbol{\theta}}(a_t|s_t)$ // 演员
如果 完成$_t = 1$ **那么**
跳出
直到收敛

在实践中，我们应该在 TD 更新中对目标值使用停止梯度算子，原因在第 2.5.2.4 节中解释。此外，通常会在策略中添加熵项，作为正则化器（以确保策略保持随机性，从而平滑损失函数 —— 参见第 3.5.4 节）。如果我们使用具有单独的价值和策略头的共享网络，我们需要使用单个损失函数来训练所有参数 $\boldsymbol{\phi}$。因此，对于每个轨迹，我们得到以下损失，我们希望最小化 TD 损失，最大化策略梯度（期望奖励）项，并最大化熵项。

$$\mathcal{L}(\boldsymbol{\phi}; \tau) = \frac{1}{T} \sum_{t=1}^{T} \left[\lambda_{TD} \mathcal{L}_{TD}(s_t, a_t, r_t, s_{t+1}) - \lambda_{PG} J_{PG}(s_t, a_t, r_t, s_{t+1}) - \lambda_{ent} J_{ent}(s_t)\right] \tag{3.20}$$

$$q_t = r_t + \gamma(1 - \text{done}(s_t)) V_{\boldsymbol{\phi}}(s_{t+1}) \tag{3.21}$$

$$\mathcal{L}_{TD}(s_t, a_t, r_t, s_{t+1}) = (\text{sg}(q_t) - V_{\phi}(s))^2 \tag{3.22}$$

$$J_{PG}(s_t, a_t, r_t, s_{t+1}) = (\text{sg}(q_t - V_{\phi}(s_t)) \log \pi_{\boldsymbol{\phi}}(a_t|s_t) \tag{3.23}$$

$$J_{ent}(s_t) = -\sum_a \pi_{\boldsymbol{\phi}}(a|s_t) \log \pi_{\boldsymbol{\phi}}(a|s_t) \tag{3.24}$$

[3] 在 [Mni+16] 中，他们提出了一种名为 **A3C** 的A2C分布式版本，代表“异步优势演员评论家”。

To handle the dynamically varying scales of the different loss functions, we can use the **PopArt** method of [Has+16; Hes+19] to allow for a fixed set of hyper-parameter values for λ_i. (PopArt stands for "Preserving Outputs Precisely, while Adaptively Rescaling Targets".)

3.3.2 Generalized advantage estimation (GAE)

In A2C, we replaced the high variance, but unbiased, MC return G_t with the low variance, but biased, one-step bootstrap return $G_{t:t+1} = r_t + \gamma V_{\boldsymbol{w}}(s_{t+1})$. More generally, we can compute the n-step estimate

$$G_{t:t+n} = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \cdots + \gamma^{n-1} r_{t+n-1} + \gamma^n V_{\boldsymbol{w}}(s_{t+n}) \tag{3.25}$$

and thus obtain the (truncated) n-step advantage estimate as follows:

$$A^{(n)}_{\boldsymbol{w}}(s_t, a_t) = G_{t:t+n} - V_{\boldsymbol{w}}(s_t) \tag{3.26}$$

Unrolling to infinity, we get

$$A^{(1)}_t = r_t + \gamma v_{t+1} - v_t \tag{3.27}$$

$$A^{(2)}_t = r_t + \gamma r_{t+1} + \gamma^2 v_{t+2} - v_t \tag{3.28}$$

$$\vdots \tag{3.29}$$

$$A^{(\infty)}_t = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \cdots - v_t \tag{3.30}$$

$A^{(1)}_t$ is high bias but low variance, and $A^{(\infty)}_t$ is unbiased but high variance.

Instead of using a single value of n, we can take a weighted average. That is, we define

$$A_t = \frac{\sum_{n=1}^{T} w_n A^{(n)}_t}{\sum_{n=1}^{T} w_n} \tag{3.31}$$

If we set $w_n = \lambda^{n-1}$ we get the following simple recursive calculation:

$$\delta_t = r_t + \gamma v_{t+1} - v_t \tag{3.32}$$

$$A_t = \delta_t + \gamma\lambda\delta_{t+1} + \cdots + (\gamma\lambda)^{T-t+1}\delta_{T-1} = \delta_t + \gamma\lambda A_{t+1} \tag{3.33}$$

Here $\lambda \in [0, 1]$ is a parameter that controls the bias-variance tradeoff: larger values decrease the bias but increase the variance. This is called **generalized advantage estimation** (**GAE**) [Sch+16b]. See Algorithm 5 for some pseudocode. Using this, we can define a general actor-critic method, as shown in Algorithm 6.

Algorithm 5: Generalized Advantage Estimation

```
def GAE(r_{1:T}, v_{1:T}, γ, λ)
A' = 0
for t = T : 1 do
    δ_t = r_t + γ v_{t+1} − v_t
    A' = δ_t + γλA'
    A_t = A' // advantage
    q_t = A_t + v_t // TD target
Return (A_{1:T}), q_{1:T})
```

We can generalize this approach even further, by using gradient estimators of the form

$$\nabla J(\boldsymbol{\theta}) = \mathbb{E}\left[\sum_{t=0}^{\infty} \Psi_t \nabla \log \pi_{\boldsymbol{\theta}}(a_t|s_t)\right] \tag{3.34}$$

处理不同损失函数动态变化的尺度，我们可以使用 **PopArt** 方法 [Has+16； Hes+19] 允许为 λ_i 设置一组固定的超参数值。（ PopArt 代表 “ 精确保留输出，自适应缩放目标 ”。）

3.3.2 广义优势估计（ GAE ）

在 A2C 中，我们用低方差但偏置的单一步长自举回报 $G_{t:t+1} = r_t + \gamma V_{\boldsymbol{w}}(s_{t+1})$ 替换了高方差但无偏的 MC 回报 G_t 。更一般地，我们可以计算 n 步估计

$$G_{t:t+n} = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \cdots + \gamma^{n-1} r_{t+n-1} + \gamma^n V_{\boldsymbol{w}}(s_{t+n}) \tag{3.25}$$

因此，得到（截断的） n- 步优势估计如下：

$$A_{\boldsymbol{w}}^{(n)}(s_t, a_t) = G_{t:t+n} - V_{\boldsymbol{w}}(s_t) \tag{3.26}$$

展开到无限，我们得到

$$A_t^{(1)} = r_t + \gamma v_{t+1} - v_t \tag{3.27}$$
$$A_t^{(2)} = r_t + \gamma r_{t+1} + \gamma^2 v_{t+2} - v_t \tag{3.28}$$
$$\vdots \tag{3.29}$$
$$A_t^{(\infty)} = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \cdots - v_t \tag{3.30}$$

$A_t^{(1)}$ 具有高偏差但低方差，而 $A_t^{(\infty)}$ 是无偏但高方差。

而不是使用单个值 n，我们可以取加权平均值。也就是说，我们定义

$$A_t = \frac{\sum_{n=1}^{T} w_n A_t^{(n)}}{\sum_{n=1}^{T} w_n} \tag{3.31}$$

如果我们设置 $w_n = \lambda^{n-1}$ ，我们得到以下简单的递归计算：

$$\delta_t = r_t + \gamma v_{t+1} - v_t \tag{3.32}$$
$$A_t = \delta_t + \gamma\lambda\delta_{t+1} + \cdots + (\gamma\lambda)^{T-t+1}\delta_{T-1} = \delta_t + \gamma\lambda A_{t+1} \tag{3.33}$$

这里 $\lambda \in [0, 1]$ 是一个控制偏差 - 方差权衡的参数：较大的值会降低偏差但增加方差。这被称为 **广义优势估计** （ **GAE** ）[Sch+16b]。请参阅算法 5 以获取一些伪代码。使用此方法，我们可以定义一个通用的 actor-critic 方法，如算法 6 所示。

算法 5： 广义优势估计

1 定义 GAE 函数 $(\delta_{t:T}, \gamma\lambda, A', \gamma, \lambda)$

3 **for** $t = T : 1$ **do**

4 $\quad \delta_t = r_t + \gamma v_{t+1} - v_t$ // TD 误差

5 $\quad A_t = \delta_t + \gamma\lambda A'$ // 优势

6 $\quad A' = A_t$

7 返回 $(A_{1:T}) \; q_{1:T}$

$t = A_t + v_t$

我们可以进一步推广这种方法，通过使用形式为 $\Psi_t \nabla \log \pi_{\boldsymbol{\theta}}(a_t|s_t)$ 的梯度估计器

$$\nabla J(\boldsymbol{\theta}) = \mathbb{E}\left[\sum_{t=0}^{\infty} \Psi_t \nabla \log \pi_{\boldsymbol{\theta}}(a_t|s_t)\right] \tag{3.34}$$

Algorithm 6: Actor critic with GAE

Initialize parameters $\boldsymbol{\phi}$, environment state s
repeat
  $(s_1, a_1, r_1, \ldots, s_T) = \text{rollout}(s, \pi_{\boldsymbol{\phi}})$
  $v_{1:T} = V_{\boldsymbol{\phi}}(s_{1:T})$
  $(A_{1:T}, q_{1:T}) = \text{sg}(\text{GAE}(r_{1:T}, v_{1:T}, \gamma, \lambda))$
  $\mathcal{L}(\boldsymbol{\phi}) = \frac{1}{T}\sum_{t=1}^{T}\left[\lambda_{TD}(V_{\boldsymbol{\phi}}(s_t) - q_t)^2 - \lambda_{PG} A_t \log \pi_{\boldsymbol{\phi}}(a_t|s_t) - \lambda_{ent}\,\mathbb{H}(\pi_{\boldsymbol{\phi}}(\cdot|s_t))\right]$
  $\boldsymbol{\phi} := \boldsymbol{\phi} - \eta\nabla\mathcal{L}(\boldsymbol{\phi})$
until *converged*

where Ψ_t may be any of the following:

$$\Psi_t = \sum_{i=t}^{\infty}\gamma^i r_i \quad \text{Monte Carlo target} \tag{3.35}$$

$$\Psi_t = \sum_{i=t}^{\infty}\gamma^i r_i - V_{\boldsymbol{w}}(s_t) \quad \text{MC with baseline} \tag{3.36}$$

$$\Psi_t = A_{\boldsymbol{w}}(s_t, a_t) \quad \text{advantage function} \tag{3.37}$$

$$\Psi_t = Q_{\boldsymbol{w}}(s_t, a_t) \quad \text{Q function} \tag{3.38}$$

$$\Psi_t = r_t + V_{\boldsymbol{w}}(s_{t+1}) - V_{\boldsymbol{w}}(s_t) \quad \text{TD residual} \tag{3.39}$$

See [Sch+16b] for details.

3.3.3 Two-time scale actor critic algorithms

In standard AC, we update the actor and critic in parallel. However, it is better to let critic $V_{\boldsymbol{w}}$ learn using a faster learning rate (or more updates), so that it reflects the value of the current policy $\pi_{\boldsymbol{\theta}}$ more accurately, in order to get better gradient estimates for the policy update. This is known as two timescale learning or **bilevel optimization** [Yu17; Zha+19; Hon+23; Zhe+22; Lor24]. (See also Section 4.2.1, where we discuss RL from a game theoretic perspective.)

3.3.4 Natural policy gradient methods

In this section, we discuss an improvement to policy gradient methods that uses preconditioning to speedup convergence. In particular, we replace gradient descent with **natural gradient descent** (**NGD**) [Ama98; Mar20], which we explain below. We then show how to combine it with actor-critic.

3.3.4.1 Natural gradient descent

NGD is a second order method for optimizing the parameters of (conditional) probability distributions, such as policies, $\pi_{\boldsymbol{\theta}}(\boldsymbol{a}|\boldsymbol{s})$. It typically converges faster and more robustly than SGD, but is computationally more expensive.

Before we explain NGD, let us review standard SGD, which is an update of the following form

$$\boldsymbol{\theta}_{k+1} = \boldsymbol{\theta}_k - \eta_k \boldsymbol{g}_k \tag{3.40}$$

where $\boldsymbol{g}_k = \nabla_{\boldsymbol{\theta}}\mathcal{L}(\boldsymbol{\theta}_k)$ is the gradient of the loss at the previous parameter values, and η_k is the learning rate. It can be shown that the above update is equivalent to minimizing a locally linear approximation to the loss, $\hat{\mathcal{L}}_k$, subject to the constraint that the new parameters do not move too far (in Euclidean distance) from the

算法 6： 基于 GAE 的演员 - 评论家

Initialize parameters $\boldsymbol{\phi}$, environment state s
repeat
  $(s_1, a_1, r_1, \ldots, s_T) = \text{rollout}(s, \pi_{\boldsymbol{\phi}})$
  $v_{1:T} = V_{\boldsymbol{\phi}}(s_{1:T})$
  $(A_{1:T}, q_{1:T}) = \text{sg}(\text{GAE}(r_{1:T}, v_{1:T}, \gamma, \lambda))$
  $\mathcal{L}(\boldsymbol{\phi}) = \frac{1}{T}\sum_{t=1}^{T}\left[\lambda_{TD}(V_{\boldsymbol{\phi}}(s_t) - q_t)^2 - \lambda_{PG} A_t \log \pi_{\boldsymbol{\phi}}(a_t|s_t) - \lambda_{ent}\,\mathbb{H}(\pi_{\boldsymbol{\phi}}(\cdot|s_t))\right]$
  $\boldsymbol{\phi} := \boldsymbol{\phi} - \eta \nabla \mathcal{L}(\boldsymbol{\phi})$
until *converged*

Ψ_t 可以是以下任何一个:

$$\Psi_t = \sum_{i=t}^{\infty} \gamma^i r_i \qquad \text{Monte Carlo target} \tag{3.35}$$

$$\Psi_t = \sum_{i=t}^{\infty} \gamma^i r_i - V_{\boldsymbol{w}}(s_t) \qquad \text{MC with baseline} \tag{3.36}$$

$$\Psi_t = A_{\boldsymbol{w}}(s_t, a_t) \qquad \text{advantage function} \tag{3.37}$$

$$\Psi_t = Q_{\boldsymbol{w}}(s_t, a_t) \qquad \text{Q function} \tag{3.38}$$

$$\Psi_t = r_t + V_{\boldsymbol{w}}(s_{t+1}) - V_{\boldsymbol{w}}(s_t) \qquad \text{TD residual} \tag{3.39}$$

请参阅 [Sch+16b] 以获取详细信息。

3.3.3 双时间尺度演员评论家算法 ms

在标准 AC 中，我们并行更新演员和评论家。然而，让评论家 $V_{\boldsymbol{w}}$ 使用更快的学习率（或更多更新）来学习会更好，这样它可以更准确地反映当前策略 $\pi_{\boldsymbol{\theta}}$ 的价值，以便获得更好的策略更新的梯度估计。这被称为双时间尺度学习或**双层优化** [Yu17； Zha+19； Hon+23； Zhe+22； Lor24]。（另见第 4.2.1 节，其中我们讨论了从博弈论角度的 RL。）

3.3.4 自然策略梯度方法

本节中，我们讨论了一种使用预条件加速收敛的改进策略梯度方法。特别是，我们用**自然梯度下降（ NGD ）** [Ama98； Mar20]，来替换梯度下降，以下我们将对其进行解释。然后，我们展示了如何将其与 actor-critic 结合。

3.3.4.1 自然梯度下降

NGD 是一种用于优化（条件）概率分布参数的二阶方法，例如策略、 $\pi_{\boldsymbol{\theta}}(\boldsymbol{a}|\boldsymbol{s})$。它通常比 SGD 收敛更快、更稳健，但计算成本更高。

在我们解释 NGD 之前，让我们回顾一下标准 SGD，它是一种以下形式的更新

$$\boldsymbol{\theta}_{k+1} = \boldsymbol{\theta}_k - \eta_k \boldsymbol{g}_k \tag{3.40}$$

$\boldsymbol{g}_k = \nabla_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{\theta}_k)$ 是在先前参数值处的损失梯度，而 η_k 是学习率。可以证明，上述更新等价于在约束新参数（在欧几里得距离上）不要移动得太远的情况下，最小化损失 $\hat{\mathcal{L}}_k$ 的局部线性近似。

(a) *(b)*

Figure 3.1: Changing the mean of a Gaussian by a fixed amount (from solid to dotted curve) can have more impact when the (shared) variance is small (as in a) compared to when the variance is large (as in b). Hence the impact (in terms of prediction accuracy) of a change to μ depends on where the optimizer is in (μ, σ) space. From Figure 3 of [Hon+10], reproduced from [Val00]. Used with kind permission of Antti Honkela.

previous parameters:

$$\boldsymbol{\theta}_{k+1} = \underset{\boldsymbol{\theta}}{\operatorname{argmin}}\, \hat{\mathcal{L}}_k(\boldsymbol{\theta}) \;\; \text{s.t.} \;\; ||\boldsymbol{\theta} - \boldsymbol{\theta}_k||_2^2 \leq \epsilon \tag{3.41}$$

$$\hat{\mathcal{L}}_k(\boldsymbol{\theta}) = \mathcal{L}(\boldsymbol{\theta}_k) + \boldsymbol{g}_k^{\mathsf{T}}(\boldsymbol{\theta} - \boldsymbol{\theta}_k) \tag{3.42}$$

where the step size η_k is proportional to ϵ. This is called a **proximal update** [PB+14].

One problem with the SGD update is that Euclidean distance in parameter space does not make sense for probabilistic models. For example, consider comparing two Gaussians, $p_{\boldsymbol{\theta}} = p(y|\mu, \sigma)$ and $p_{\boldsymbol{\theta}'} = p(y|\mu', \sigma')$. The (squared) Euclidean distance between the parameter vectors decomposes as $||\boldsymbol{\theta} - \boldsymbol{\theta}'||_2^2 = (\mu - \mu')^2 + (\sigma - \sigma')^2$. However, the predictive distribution has the form $\exp(-\frac{1}{2\sigma^2}(y - \mu)^2)$, so changes in μ need to be measured relative to σ. This is illustrated in Figure 3.1(a-b), which shows two univariate Gaussian distributions (dotted and solid lines) whose means differ by ϵ. In Figure 3.1(a), they share the same small variance σ^2, whereas in Figure 3.1(b), they share the same large variance. It is clear that the difference in μ matters much more (in terms of the effect on the distribution) when the variance is small. Thus we see that the two parameters interact with each other, which the Euclidean distance cannot capture.

The key to NGD is to measure the notion of distance between two probability distributions in terms of the KL divergence. This can be approximated in terms of the **Fisher information matrix** (FIM). In particular, for any given input $\boldsymbol{x}$, we have

$$D_{\mathbb{KL}}\left(p_{\boldsymbol{\theta}}(\boldsymbol{y}|\boldsymbol{x}) \parallel p_{\boldsymbol{\theta}+\boldsymbol{\delta}}(\boldsymbol{y}|\boldsymbol{x})\right) \approx \frac{1}{2}\boldsymbol{\delta}^{\mathsf{T}}\mathbf{F}_{\boldsymbol{x}}\boldsymbol{\delta} \tag{3.43}$$

where $\mathbf{F}_{\boldsymbol{x}}$ is the FIM

$$\mathbf{F}_{\boldsymbol{x}}(\boldsymbol{\theta}) = -\mathbb{E}_{p_{\boldsymbol{\theta}}(\boldsymbol{y}|\boldsymbol{x})}\left[\nabla^2 \log p_{\boldsymbol{\theta}}(\boldsymbol{y}|\boldsymbol{x})\right] = \mathbb{E}_{p_{\boldsymbol{\theta}}(\boldsymbol{y}|\boldsymbol{x})}\left[(\nabla \log p_{\boldsymbol{\theta}}(\boldsymbol{y}|\boldsymbol{x}))(\nabla \log p_{\boldsymbol{\theta}}(\boldsymbol{y}|\boldsymbol{x}))^{\mathsf{T}}\right] \tag{3.44}$$

We now replace the Euclidean distance between the parameters, $d(\boldsymbol{\theta}_k, \boldsymbol{\theta}_{k+1}) = ||\boldsymbol{\delta}||_2^2$, with

$$d(\boldsymbol{\theta}_k, \boldsymbol{\theta}_{k+1}) = \boldsymbol{\delta}^{\mathsf{T}}\mathbf{F}_k\boldsymbol{\delta}_k \tag{3.45}$$

where $\boldsymbol{\delta} = \boldsymbol{\theta}_{k+1} - \boldsymbol{\theta}_k$ and $\mathbf{F}_k = \mathbf{F}_{\boldsymbol{x}}(\boldsymbol{\theta}_k)$ for a randomly chosen input $\boldsymbol{x}$. This gives rise to the following constrained optimization problem:

$$\boldsymbol{\delta}_k = \underset{\boldsymbol{\delta}}{\operatorname{argmin}}\, \hat{\mathcal{L}}_k(\boldsymbol{\theta}_k + \boldsymbol{\delta}) \;\; \text{s.t.} \;\; \boldsymbol{\delta}^{\mathsf{T}}\mathbf{F}_k\boldsymbol{\delta} \leq \epsilon \tag{3.46}$$

If we replace the constraint with a Lagrange multiplier, we get the unconstrained objective:

$$J_k(\boldsymbol{\delta}) = \mathcal{L}(\boldsymbol{\theta}_k) + \boldsymbol{g}_k^{\mathsf{T}}\boldsymbol{\delta} + \eta_k\boldsymbol{\delta}^{\mathsf{T}}\mathbf{F}_k\boldsymbol{\delta} \tag{3.47}$$

(a) (b)

图 3.1：通过固定量改变高斯分布的均值（从实线到虚线曲线）对（共享）方差较小的情况（如图 a 所示）的影响可能更大，而方差较大时（如图 b 所示）的影响较小。因此，对 μ 的改变（在预测精度方面）的影响取决于优化器在 (μ,σ) 空间中的位置。本图来自 [Hon+10] 的第 3 图，由 [Val00] 复制。经 Antti Honkela 许可使用。

先前参数：

$$\boldsymbol{\theta}_{k+1} = \underset{\boldsymbol{\theta}}{\operatorname{argmin}}\, \hat{\mathcal{L}}_k(\boldsymbol{\theta}) \;\; \text{s.t.} \;\; ||\boldsymbol{\theta} - \boldsymbol{\theta}_k||_2^2 \leq \epsilon \tag{3.41}$$

$$\hat{\mathcal{L}}_k(\boldsymbol{\theta}) = \mathcal{L}(\boldsymbol{\theta}_k) + \boldsymbol{g}_k^{\mathsf{T}}(\boldsymbol{\theta} - \boldsymbol{\theta}_k) \tag{3.42}$$

步长 η_k 与 ϵ 成比例。这被称为**近端更新** [PB+14]。

SGD 更新的问题之一是，在参数空间中，欧几里得距离对于概率模型没有意义。例如，考虑比较两个高斯分布，$p_{\boldsymbol{\theta}} = p(y|\mu,\sigma)$ 和 $p_{\boldsymbol{\theta}'} = p(y|\mu',\sigma')$。参数向量之间的（平方）欧几里得距离可以分解为 $||\boldsymbol{\theta}-\boldsymbol{\theta}'||_2^2 = (\mu-\mu')^2+(\sigma-\sigma')^2$。然而，预测分布的形式为 $\exp(-\ {}^{1}_{2\sigma^2}(y-\mu)^2)$，因此需要相对于 μ 来衡量变化。这如图 3.1(a-b) 所示，该图显示了两个一元高斯分布（虚线和实线），其均值相差 ϵ。在图 3.1(a) 中，它们具有相同的小方差 σ^2，而在图 3.1(b) 中，它们具有相同的大方差。很明显，当方差较小时，μ 的差异（在影响分布方面）更为重要。因此，我们可以看到这两个参数相互作用，这是欧几里得距离无法捕捉到的。

NGD 的关键在于衡量两个概率分布之间的距离概念，以 KL 散度来衡量。这可以通过 **Fisher 信息矩阵**（FIM）来近似。特别是，对于任何给定的输入 $\boldsymbol{x}$，我们有

$$D_{\mathbb{KL}}\left(p_{\boldsymbol{\theta}}(\boldsymbol{y}|\boldsymbol{x}) \parallel p_{\boldsymbol{\theta}+\boldsymbol{\delta}}(\boldsymbol{y}|\boldsymbol{x})\right) \approx \frac{1}{2}\boldsymbol{\delta}^{\mathsf{T}}\mathbf{F}_{\boldsymbol{x}}\boldsymbol{\delta} \tag{3.43}$$

$\mathbf{F}_{\boldsymbol{x}}$ 是 FIM

$$\mathbf{F}_{\boldsymbol{x}}(\boldsymbol{\theta}) = -\mathbb{E}_{p_{\boldsymbol{\theta}}(\boldsymbol{y}|\boldsymbol{x})}\left[\nabla^2 \log p_{\boldsymbol{\theta}}(\boldsymbol{y}|\boldsymbol{x})\right] = \mathbb{E}_{p_{\boldsymbol{\theta}}(\boldsymbol{y}|\boldsymbol{x})}\left[(\nabla \log p_{\boldsymbol{\theta}}(\boldsymbol{y}|\boldsymbol{x}))(\nabla \log p_{\boldsymbol{\theta}}(\boldsymbol{y}|\boldsymbol{x}))^{\mathsf{T}}\right] \tag{3.44}$$

我们现在将参数之间的欧几里得距离，$d(\boldsymbol{\theta}_k,\boldsymbol{\theta}_{k+1}) = ||\boldsymbol{\delta}||_2^2$，替换为

$$d(\boldsymbol{\theta}_k, \boldsymbol{\theta}_{k+1}) = \boldsymbol{\delta}^{\mathsf{T}}\mathbf{F}_k\boldsymbol{\delta}_k \tag{3.45}$$

$\boldsymbol{\delta} = \boldsymbol{\theta}_{k+1} - \boldsymbol{\theta}_k$ 和 $\mathbf{F}_k = \mathbf{F}_{\boldsymbol{x}}$（$\boldsymbol{\theta}_k$）对于随机选择的输入 $\boldsymbol{x}$。这导致以下约束优化问题：

$$\boldsymbol{\delta}_k = \underset{\boldsymbol{\delta}}{\operatorname{argmin}}\, \hat{\mathcal{L}}_k(\boldsymbol{\theta}_k + \boldsymbol{\delta}) \;\; \text{s.t.} \;\; \boldsymbol{\delta}^{\mathsf{T}}\mathbf{F}_k\boldsymbol{\delta} \leq \epsilon \tag{3.46}$$

如果我们用拉格朗日乘子替换约束条件，我们得到无约束的目标函数：

$$J_k(\boldsymbol{\delta}) = \mathcal{L}(\boldsymbol{\theta}_k) + \boldsymbol{g}_k^{\mathsf{T}}\boldsymbol{\delta} + \eta_k\boldsymbol{\delta}^{\mathsf{T}}\mathbf{F}_k\boldsymbol{\delta} \tag{3.47}$$

Solving $J_k(\boldsymbol{\delta}) = 0$ gives the update

$$\boldsymbol{\delta} = -\eta_k \mathbf{F}_k^{-1} \boldsymbol{g}_k \tag{3.48}$$

The term $\mathbf{F}^{-1}\boldsymbol{g}$ is called the **natural gradient**. This is equivalent to a preconditioned gradient update, where we use the inverse FIM as a preconditioning matrix. We can compute the (adaptive) learning rate using

$$\eta_k = \sqrt{\frac{\epsilon}{\boldsymbol{g}_k^\mathsf{T} \mathbf{F}_k^{-1} \boldsymbol{g}_k}} \tag{3.49}$$

Computing the FIM can be hard. A simple approximation is to replace the model's distribution with the empirical distribution. In particular, define $p_\mathcal{D}(\boldsymbol{x}, \boldsymbol{y}) = \frac{1}{N}\sum_{n=1}^N \delta_{\boldsymbol{x}_n}(\boldsymbol{x})\delta_{\boldsymbol{y}_n}(\boldsymbol{y})$, $p_\mathcal{D}(\boldsymbol{x}) = \frac{1}{N}\sum_{n=1}^N \delta_{\boldsymbol{x}_n}(\boldsymbol{x})$ and $p_{\boldsymbol{\theta}}(\boldsymbol{x}, \boldsymbol{y}) = p_\mathcal{D}(\boldsymbol{x})p(\boldsymbol{y}|\boldsymbol{x}, \boldsymbol{\theta})$. Then we can compute the **empirical Fisher** [Mar16] as follows:

$$\mathbf{F}(\boldsymbol{\theta}) = \mathbb{E}_{p_{\boldsymbol{\theta}}(\boldsymbol{x},\boldsymbol{y})}\left[\nabla \log p(\boldsymbol{y}|\boldsymbol{x}, \boldsymbol{\theta}) \nabla \log p(\boldsymbol{y}|\boldsymbol{x}, \boldsymbol{\theta})^\mathsf{T}\right] \tag{3.50}$$

$$\approx \mathbb{E}_{p_\mathcal{D}(\boldsymbol{x},\boldsymbol{y})}\left[\nabla \log p(\boldsymbol{y}|\boldsymbol{x}, \boldsymbol{\theta}) \nabla \log p(\boldsymbol{y}|\boldsymbol{x}, \boldsymbol{\theta})^\mathsf{T}\right] \tag{3.51}$$

$$= \frac{1}{|\mathcal{D}|} \sum_{(\boldsymbol{x},\boldsymbol{y})\in\mathcal{D}} \nabla \log p(\boldsymbol{y}|\boldsymbol{x}, \boldsymbol{\theta}) \nabla \log p(\boldsymbol{y}|\boldsymbol{x}, \boldsymbol{\theta})^\mathsf{T} \tag{3.52}$$

3.3.4.2 Natural actor critic

To apply NGD to RL, we can adapt the A2C algorithm in Algorithm 6. In particular, define

$$\boldsymbol{g}_{kt} = \nabla_{\boldsymbol{\theta}_k} A_t \log \pi_{\boldsymbol{\theta}}(\boldsymbol{a}_t|\boldsymbol{s}_t) \tag{3.53}$$

where A_t is the advantage function at step t of the random trajectory generated by the policy at iteration k. Now we compute

$$\boldsymbol{g}_k = \frac{1}{T}\sum_{t=1}^T \boldsymbol{g}_{kt}, \; \mathbf{F}_k = \frac{1}{T}\sum_{t=1}^T \boldsymbol{g}_{kt}\boldsymbol{g}_{kt}^\mathsf{T} \tag{3.54}$$

and compute $\boldsymbol{\delta}_{k+1} = -\eta_k \mathbf{F}_k^{-1}\boldsymbol{g}_k$. This approach is called **natural policy gradient** [Kak01; Raj+17].

We can compute $\mathbf{F}_k^{-1}\boldsymbol{g}_k$ without having to invert $\mathbf{F}_k$ by using the conjugate gradient method, where each CG step uses efficient methods for Hessian-vector products [Pea94]. This is called **Hessian free optimization** [Mar10]. Similarly, we can efficiently compute $\boldsymbol{g}_k^\mathsf{T}(\mathbf{F}_k^{-1}\boldsymbol{g}_k)$.

As a more accurate alternative to the empirical Fisher, [MG15] propose the **KFAC** method, which stands for "Kronecker factored approximate curvature"; this approximates the FIM of a DNN as a block diagonal matrix, where each block is a Kronecker product of two small matrices. This was applied to policy gradient learning in [Wu+17].

3.4 Policy improvement methods

In this section, we discuss methods that try to monotonically improve performance of the policy at each step, rather than just following the gradient, which can result in a high variance estimate where performance can increase or decrease at each step. These are called **policy improvement** methods. Our presentation is based on [QPC24].

3.4.1 Policy improvement lower bound

We start by stating a useful result from [Ach+17]. Let π_k be the current policy at step k, and let π be any other policy (e.g., a candidate new one). Let $p^\gamma_{\pi_k}$ be the normalized discounted state visitation distribution for π_k, defined in Equation (3.8). Let $A^{\pi_k}(s,a) = Q^{\pi_k}(s,a) - V^{\pi_k}(s)$ be the advantage function. Finally, let the total variation distance between two distributions be given by

$$\mathrm{TV}(p,q) \triangleq \frac{1}{2}||\boldsymbol{p} - \boldsymbol{q}||_1 = \frac{1}{2}\sum_s |p(s) - q(s)| \tag{3.55}$$

解决 $J_k(\boldsymbol{\delta})=0$ 提供更新

$$\boldsymbol{\delta} = -\eta_k \mathbf{F}_k^{-1} \boldsymbol{g}_k \tag{3.48}$$

该术语 $\mathbf{F}^{-1}\boldsymbol{g}$ 被称为**自然梯度**。这相当于一个预处理的梯度更新，其中我们使用 FIM 的逆作为预处理矩阵。我们可以使用 计算（自适应）学习率。

$$\eta_k = \frac{\epsilon}{\sqrt{\boldsymbol{g}_k^{\mathsf{T}} \mathbf{F}_k^{-1} \boldsymbol{g}_k}} \tag{3.49}$$

计算 FIM 可能很困难。一个简单的近似是将模型的分布替换为经验分布。特别是，定义 $p_{\mathcal{D}}(\boldsymbol{x},\boldsymbol{y}) = \frac{1}{N}\sum_{n=1}^{N} \delta_{\boldsymbol{x}_n}(\boldsymbol{x})\delta_{\boldsymbol{y}_n}(\boldsymbol{y})$, $p_{\mathcal{D}}(\boldsymbol{x}) = \frac{1}{N}\sum_{n=1}^{N} \delta_{\boldsymbol{x}_n}(\boldsymbol{x})$ 和 $p_{\boldsymbol{\theta}}(\boldsymbol{x},\boldsymbol{y}) = p_{\mathcal{D}}(\boldsymbol{x})p(\boldsymbol{y}|\boldsymbol{x},\boldsymbol{\theta})$。然后我们可以按照以下方式计算经验 Fisher**Mar**[:

$$\mathbf{F}(\boldsymbol{\theta}) = \mathbb{E}_{p_{\boldsymbol{\theta}}(\boldsymbol{x},\boldsymbol{y})}\left[\nabla \log p(\boldsymbol{y}|\boldsymbol{x},\boldsymbol{\theta}) \nabla \log p(\boldsymbol{y}|\boldsymbol{x},\boldsymbol{\theta})^{\mathsf{T}}\right] \tag{3.50}$$

$$\approx \mathbb{E}_{p_{\mathcal{D}}(\boldsymbol{x},\boldsymbol{y})}\left[\nabla \log p(\boldsymbol{y}|\boldsymbol{x},\boldsymbol{\theta}) \nabla \log p(\boldsymbol{y}|\boldsymbol{x},\boldsymbol{\theta})^{\mathsf{T}}\right] \tag{3.51}$$

$$= \frac{1}{|\mathcal{D}|} \sum_{(\boldsymbol{x},\boldsymbol{y})\in\mathcal{D}} \nabla \log p(\boldsymbol{y}|\boldsymbol{x},\boldsymbol{\theta}) \nabla \log p(\boldsymbol{y}|\boldsymbol{x},\boldsymbol{\theta})^{\mathsf{T}} \tag{3.52}$$

3.3.4.2 自然演员 - 评论家

将 NGD 应用于 RL，我们可以对算法 6 中的 A2C 算法进行适配。特别是，定义

$$\boldsymbol{g}_{kt} = \nabla_{\boldsymbol{\theta}_k} A_t \log \pi_{\boldsymbol{\theta}}(\boldsymbol{a}_t|\boldsymbol{s}_t) \tag{3.53}$$

A_t 是随机轨迹在迭代 k 时由策略生成的优势函数在第 t 步的值。现在我们计算

$$\boldsymbol{g}_k = \frac{1}{T}\sum_{t=1}^{T} \boldsymbol{g}_{kt}, \ \mathbf{F}_k = \frac{1}{T}\sum_{t=1}^{T} \boldsymbol{g}_{kt}\boldsymbol{g}_{kt}^{\mathsf{T}} \tag{3.54}$$

并计算 $\boldsymbol{\delta}_{k+1} = -\eta_k \mathbf{F}_k^{-1}\boldsymbol{g}_k$。这种方法被称为**自然策略梯度** [Kak01； Raj+17]。

我们可以计算 $\mathbf{F}_k$，通过使用共轭梯度法，其中每个 CG 步骤使用高效的 Hessian 向量积方法 [Pea94]。这被称为 **Hessian free 优化** [Mar10]。同样，我们可以有效地计算 $\boldsymbol{g}_k^{\mathsf{T}}(\mathbf{F}_k^{-1}\boldsymbol{g}_k)$。

作为经验 Fisher 的一个更准确的替代方案，[MG15] 提出了 KFAC 方法，代表 “ 克罗内克分解近似曲率 ” ；该方法将 DNN 的 FIM 近似为一个分块对角矩阵，其中每个块是两个小矩阵的克罗内克积。这被应用于 Wu[的政策梯度学习。+17]

3.4 政策改进方法

在这个部分，我们讨论了尝试单调提高每一步策略性能的方法，而不是仅仅跟随梯度，这可能导致高方差估计，其中性能可以在每一步增加或减少。这些被称为**策略改进**方法。我们的展示基于 [QPC24]。

3.4.1 政策改进下限

我们首先从 [Ach+17] 中陈述一个有用的结果。设 π_k 为第 k 步的当前策略，设 π 为任何其他策略（例如，一个候选的新策略）。设 $p_{\pi_k}^{\gamma}$ 为 π_k 的归一化折扣状态访问分布，定义在方程（ 3.8 ）。设 $A^{\pi_k}(s,a) = Q^{\pi_k}(s,a) - V^{\pi_k}(s)$ 为优势函数。最后，设两个分布之间的总变分距离由 给出

$$\mathrm{TV}(p,q) \triangleq \frac{1}{2}||\boldsymbol{p}-\boldsymbol{q}||_1 = \frac{1}{2}\sum_{s}|p(s)-q(s)| \tag{3.55}$$

Then one can show [Ach+17] that

$$J(\pi) - J(\pi_k) \geq \frac{1}{1-\gamma} \underbrace{\mathbb{E}_{p^\gamma_{\pi_k}(s)\pi_k(a|s)}\left[\frac{\pi(a|s)}{\pi_k(a|s)} A^{\pi_k}(s,a)\right]}_{L(\pi,\pi_k)} - \frac{2\gamma C^{\pi,\pi_k}}{(1-\gamma)^2} \mathbb{E}_{p^\gamma_{\pi_k}(s)}\left[\mathrm{TV}(\pi(\cdot|s), \pi_k(\cdot|s))\right] \tag{3.56}$$

where $C^{\pi,\pi_k} = \max_s \left|\mathbb{E}_{\pi(a|s)}\left[A^{\pi_k}(s,a)\right]\right|$. In the above, $L(\pi, \pi_k)$ is a surrogate objective, and the second term is a penalty term.

If we can optimize this lower bound (or a stochastic approximation, based on samples from the current policy π_k), we can guarantee monotonic policy improvement (in expectation) at each step. We will replace this objective with a trust-region update that is easier to optimize:

$$\pi_{k+1} = \operatorname*{argmax}_\pi L(\pi, \pi_k) \quad \text{s.t.} \quad \mathbb{E}_{p^\gamma_{\pi_k}(s)}\left[\mathrm{TV}(\pi, \pi_k)(s)\right] \leq \epsilon \tag{3.57}$$

The constraint bounds the worst-case performance decline at each update. The overall procedure becomes an approximate policy improvement method. There are various ways of implementing the above method in practice, some of which we discuss below. (See also [GDWF22], who propose a framework called **mirror learning**, that justifies these "approximations" as in fact being the optimal thing to do for a different kind of objective.)

3.4.2 Trust region policy optimization (TRPO)

In this section, we describe the **trust region policy optimization** (**TRPO**) method of [Sch+15b]. This implements an approximation to Equation (3.57). First, it leverages the fact that if

$$\mathbb{E}_{p^\gamma_{\pi_k}(s)}\left[D_{\mathbb{KL}}\left(\pi_k \parallel \pi\right)(s)\right] \leq \delta \tag{3.58}$$

then π also satisfies the TV constraint with $\delta = \frac{\epsilon^2}{2}$. Next it considers a first-order expansion of the surrogate objective to get

$$L(\pi, \pi_k) = \mathbb{E}_{p^\gamma_{\pi_k}(s)\pi_k(a|s)}\left[\frac{\pi(a|s)}{\pi_k(a|s)} A^{\pi_k}(s,a)\right] \approx \boldsymbol{g}_k^\mathsf{T}(\boldsymbol{\theta} - \boldsymbol{\theta}_k) \tag{3.59}$$

where $\boldsymbol{g}_k = \nabla_{\boldsymbol{\theta}} L(\pi_{\boldsymbol{\theta}}, \pi_k)|_{\boldsymbol{\theta}_k}$. Finally it considers a second-order expansion of the KL term to get the approximate constraint

$$\mathbb{E}_{p^\gamma_{\pi_k}(s)}\left[D_{\mathbb{KL}}\left(\pi_k \parallel \pi\right)(s)\right] \approx \frac{1}{2}(\boldsymbol{\theta} - \boldsymbol{\theta}_k)^\mathsf{T} \mathbf{F}_k (\boldsymbol{\theta} - \boldsymbol{\theta}_k) \tag{3.60}$$

where $\mathbf{F}_k = \boldsymbol{g}_k \boldsymbol{g}_k^\mathsf{T}$ is an approximation to the Fisher information matrix (see Equation (3.54)). We then use the update

$$\boldsymbol{\theta}_{k+1} = \boldsymbol{\theta}_k + \eta_k \boldsymbol{v}_k \tag{3.61}$$

where $\boldsymbol{v}_k = \mathbf{F}_k^{-1} \boldsymbol{g}_k$ is the natural gradient, and the step size is initialized to $\eta_k = \sqrt{\frac{2\delta}{\boldsymbol{v}_k^\mathsf{T} \mathbf{F}_k \boldsymbol{v}_k}}$. (In practice we compute $\boldsymbol{v}_k$ by approximately solving the linear system $\mathbf{F}_k \boldsymbol{v} = \boldsymbol{g}_k$ using conjugate gradient methods, which just require matrix vector multiplies.) We then use a backtracking line search procedure to ensure the trust region is satisfied.

3.4.3 Proximal Policy Optimization (PPO)

In this section, we describe the the **proximal policy optimization** or **PPO** method of [Sch+17], which is a simplification of TRPO.

We start by noting the following result:

$$\mathbb{E}_{p^\gamma_{\pi_k}(s)}\left[\mathrm{TV}(\pi, \pi_k)(s)\right] = \frac{1}{2}\mathbb{E}_{(s,a)\sim p^\gamma_{\pi_k}}\left[\left|\frac{\pi(a|s)}{\pi_k(a|s)} - 1\right|\right] \tag{3.62}$$

然后可以展示 [Ach+17]that

$$J(\pi) - J(\pi_k) \geq \frac{1}{1-\gamma} \underbrace{\mathbb{E}_{p^{\gamma}_{\pi_k}(s)\pi_k(a|s)} \left[\frac{\pi(a|s)}{\pi_k(a|s)} A^{\pi_k}(s,a) \right]}_{L(\pi,\pi_k)} - \frac{2\gamma C^{\pi,\pi_k}}{(1-\gamma)^2} \mathbb{E}_{p^{\gamma}_{\pi_k}(s)} \left[\mathrm{TV}(\pi(\cdot|s), \pi_k(\cdot|s)) \right] \tag{3.56}$$

在上述表达式中，$C^{\pi,\ \pi_k} = \max_s \left| \mathbb{E}_{\pi(a|s)} \left[A^{\pi_k} \left(s,\ a \right) \right] \right|$。其中，$L\ (\pi,\ \pi_k)$ 是一个代理目标函数，第二项是惩罚项。

如果我们能优化这个下界（或基于当前策略 π_k 样本的随机近似），我们就可以保证每一步策略单调改进（在期望上）。我们将用更容易优化的信任域更新来替换这个目标。

$$\pi_{k+1} = \underset{\pi}{\operatorname{argmax}}\, L(\pi, \pi_k) \quad \text{s.t.} \quad \mathbb{E}_{p^{\gamma}_{\pi_k}(s)} \left[\mathrm{TV}(\pi, \pi_k)(s) \right] \leq \epsilon \tag{3.57}$$

约束条件限制了每次更新时的最坏情况性能下降。整体过程成为了一种近似策略改进方法。在实践中，有各种方法来实现上述方法，其中一些我们将在下面讨论。（另见 [GDWF22], ，他们提出了一种称为 **镜像学习** 的框架，该框架证明这些 “ 近似 ” 实际上是为了不同类型的目标而采取的最优做法。）

3.4.2 信任区域策略优化（ TRPO ）

本节中，我们描述了**信任域策略优化**（ **TRPO** ）方法 [Sch+15b]。这实现了对公式（ 3.57 ）的近似。首先，它利用了以下事实，即如果

$$\mathbb{E}_{p^{\gamma}_{\pi_k}(s)} \left[D_{\mathbb{KL}} \left(\pi_k \parallel \pi \right)(s) \right] \leq \delta \tag{3.58}$$

然后 π 也满足 TV 约束与 $\delta = \frac{\epsilon^2}{2}$。接下来，它考虑对代理目标函数进行一阶展开以获得

$$L(\pi, \pi_k) = \mathbb{E}_{p^{\gamma}_{\pi_k}(s)\pi_k(a|s)} \left[\frac{\pi(a|s)}{\pi_k(a|s)} A^{\pi_k}(s,a) \right] \approx \boldsymbol{g}_k^{\mathsf{T}} (\boldsymbol{\theta} - \boldsymbol{\theta}_k) \tag{3.59}$$

$\boldsymbol{g}_k = \nabla_{\boldsymbol{\theta}} L(\pi_{\boldsymbol{\theta}}, \pi_k)|_{\boldsymbol{\theta}_k}$。最后，它考虑 KL 项的二阶展开以获得近似的约束

$$\mathbb{E}_{p^{\gamma}_{\pi_k}(s)} \left[D_{\mathbb{KL}} \left(\pi_k \parallel \pi \right)(s) \right] \approx \frac{1}{2} (\boldsymbol{\theta} - \boldsymbol{\theta}_k)^{\mathsf{T}} \mathbf{F}_k (\boldsymbol{\theta} - \boldsymbol{\theta}_k) \tag{3.60}$$

其中 $\mathbf{F}_k = \boldsymbol{g}_k \boldsymbol{g}_k^{\mathsf{T}}$ 是 Fisher 信息矩阵的近似（见方程（ 3.54 ））。然后我们使用更新

$$\boldsymbol{\theta}_{k+1} = \boldsymbol{\theta}_k + \eta_k \boldsymbol{v}_k \tag{3.61}$$

自然梯度为 $\boldsymbol{v}_k = \mathbf{F}_k^{-1} \boldsymbol{g}_k$ ，步长初始化为 $\eta_k = \sqrt{\frac{2\delta}{\boldsymbol{v}_k^{\mathsf{T}} \mathbf{F}_k \boldsymbol{v}_k}}$。在实际操作中，我们通过使用共轭梯度法近似求解线性系统 $\boldsymbol{v}_k\ \mathrm{F}_k \boldsymbol{v} = \boldsymbol{g}_k$ ，这只需要矩阵向量乘法。然后我们使用回溯线搜索过程以确保满足信任域。

3.4.3 近端策略优化（ PPO ）

在本节中，我们描述了 Sch+17] 的近端策略优化（ PPO ）方法，这是 TRPO 的简化版。

我们首先注意到以下结果：

$$\mathbb{E}_{p^{\gamma}_{\pi_k}(s)} \left[\mathrm{TV}(\pi, \pi_k)(s) \right] = \frac{1}{2} \mathbb{E}_{(s,a) \sim p^{\gamma}_{\pi_k}} \left[\left| \frac{\pi(a|s)}{\pi_k(a|s)} - 1 \right| \right] \tag{3.62}$$

This holds provided the support of π is contained in the support of π_k at every state. We then use the following update:

$$\pi_{k+1} = \underset{\pi}{\operatorname{argmax}} \mathbb{E}_{(s,a)\sim p^{\gamma}_{\pi_k}} \left[\min\left(\rho_k(s,a)A^{\pi_k}(s,a), \tilde{\rho}_k(s,a)A^{\pi_k}(s,a)\right)\right] \tag{3.63}$$

where $\rho_k(s,a) = \frac{\pi(a|s)}{\pi_k(a|s)}$ is the likelihood ratio, and $\tilde{\rho}_k(s,a) = \text{clip}(\rho_k(s,a), 1-\epsilon, 1+\epsilon)$, where $\text{clip}(x,l,u) = \min(\max(x,l),u)$. See [GDWF22] for a theoretical justification for these simplifications. Furthermore, this can be modified to ensure monotonic improvement as discussed in [WHT19], making it a true bound optimization method.

Some pseudocode for PPO (with GAE) is given in Algorithm 7. It is basically identical to the AC code in Algorithm 6, except the policy loss has the form $\min(\rho_t A_t, \tilde{\rho}_t A_t)$ instead of $A_t \log \pi_{\boldsymbol{\phi}}(a_t|s_t)$, and we perform multiple policy updates per rollout, for increased sample efficiency. For all the implementation details, see https://iclr-blog-track.github.io/2022/03/25/ppo-implementation-details/.

Algorithm 7: PPO with GAE

Initialize parameters $\boldsymbol{\phi}$, environment state s
for *iteration* $k = 1, 2, \ldots$ **do**
  $(\tau, s) = \text{rollout}(s, \pi_{\boldsymbol{\phi}})$
  $(s_1, a_1, r_1, \ldots, s_T) = \tau$
  $v_t = V_{\boldsymbol{\phi}}(s_t)$ for $t = 1:T$
  $(A_{1:T}, q_{1:T}) = \text{GAE}(r_{1:T}, v_{1:T}, \gamma, \lambda)$
  $\boldsymbol{\phi}_{\text{old}} \leftarrow \boldsymbol{\phi}$
  for $m = 1:M$ **do**
    $\rho_t = \frac{\pi_{\boldsymbol{\phi}}(a_t|s_t)}{\pi_{\boldsymbol{\phi}_{\text{old}}}(a_t|s_t)}$ for $t = 1:T$
    $\tilde{\rho}_t = \text{clip}(\rho_t)$ for $t = 1:T$
    $\mathcal{L}(\boldsymbol{\phi}) = \frac{1}{T}\sum_{t=1}^{T}\left[\lambda_{TD}(V_{\boldsymbol{\phi}}(s_t) - q_t)^2 - \lambda_{PG}\min(\rho_t A_t, \tilde{\rho}_t A_t) - \lambda_{ent}\,\mathbb{H}(\pi_{\boldsymbol{\phi}}(\cdot|s_t))\right]$
    $\boldsymbol{\phi} := \boldsymbol{\phi} - \eta\nabla_{\boldsymbol{\phi}}\mathcal{L}(\boldsymbol{\phi})$

3.4.4 VMPO

In this section, we discuss the **VMPO** algorithm of [FS+19], which is an on-policy extenson of the earlier on-policy **MPO** algorithm (MAP policy optimization) from [Abd+18]. It was originally explained in terms of "control as inference" (see Section 1.5), but we can also view it as a contrained policy improvement method, based on Equation (3.57). In particular, VMPO leverages the fact that if

$$\mathbb{E}_{p^{\gamma}_{\pi_k}(s)}\left[D_{\mathbb{KL}}\left(\pi \parallel \pi_k\right)(s)\right] \leq \delta \tag{3.64}$$

then π also satisfies the TV constraint with $\delta = \frac{\epsilon^2}{2}$.

Note that here the KL is reversed compared to TRPO in Section 3.4.2. This new version will encourage π to be mode-covering, so it will naturally have high entropy, which can result in improved robustness. Unfortunately, this kind of KL is harder to compute, since we are taking expectations wrt the unknown distribution π.

To solve this problem, VMPO adopts an EM-type approach. In the E step, we compute a non-parametric version of the state-action distribution given by the unknown new policy:

$$\psi(s,a) = \pi(a|s)p^{\gamma}_{\pi_k}(s) \tag{3.65}$$

The optimal new distribution is given by

$$\psi_{k+1} = \underset{\psi}{\operatorname{argmax}}\, \mathbb{E}_{\psi(s,a)}\left[A^{\pi_k}(s,a)\right] \quad \text{s.t.} \quad D_{\mathbb{KL}}\left(\psi \parallel \psi_k\right) \leq \delta \tag{3.66}$$

这成立，前提是 π 的支持包含在每个状态下的 π_k 的支持。然后我们使用以下更新：

$$\pi_{k+1} = \underset{\pi}{\operatorname{argmax}} \mathbb{E}_{(s,a)\sim p^{\gamma}_{\pi_k}} \left[\min\left(\rho_k(s,a)A^{\pi_k}(s,a), \tilde{\rho}_k(s,a)A^{\pi_k}(s,a)\right)\right] \tag{3.63}$$

其中 $\rho_k(s,a) = \frac{\pi(a|s)}{\pi_k(a|s)}$ 是似然比，且 $\tilde{\rho}_k(s,a) = \text{clip}(\rho_k(s,a)1-\epsilon, 1+\epsilon)$，其中 $\text{clip}(x,l,u) = \min(\max(x,l)u)$。参见 [GDWF22] 对这些简化的理论依据。此外，这可以修改为确保单调改进，如 [WHT19]，中讨论的那样，使其成为一种真正的边界优化方法。

以下为 PPO （带 GAE ）的伪代码，见算法 7。它基本上与算法 6 中的 AC 代码相同，除了策略损失的形式为 $\min(\rho_t A_t, \tilde{\rho}_t A_t)$，而不是 $A_t \log \pi_{\boldsymbol{\phi}}(a_t|s_t)$，并且我们在每次 rollout 中执行多次策略更新，以提高样本效率。所有实现细节，请见 https://iclr-blog-track.github.io/2022/03/25/ppo-implementation-details/。

算法 7： 使用 GAE 的 PPO

Initialize parameters $\boldsymbol{\phi}$, environment state s
for *iteration* $k = 1, 2, \ldots$ **do**
  $(\tau, s) = \text{rollout}(s, \pi_{\boldsymbol{\phi}})$
  $(s_1, a_1, r_1, \ldots, s_T) = \tau$
  $v_t = V_{\boldsymbol{\phi}}(s_t)$ for $t = 1:T$
  $(A_{1:T}, q_{1:T}) = \text{GAE}(r_{1:T}, v_{1:T}, \gamma, \lambda)$
  $\boldsymbol{\phi}_{\text{old}} \leftarrow \boldsymbol{\phi}$
  for $m = 1:M$ **do**
    $\rho_t = \frac{\pi_{\boldsymbol{\phi}}(a_t|s_t)}{\pi_{\boldsymbol{\phi}_{\text{old}}}(a_t|s_t)}$ for $t = 1:T$
    $\tilde{\rho}_t = \text{clip}(\rho_t)$ for $t = 1:T$
    $\mathcal{L}(\boldsymbol{\phi}) = \frac{1}{T}\sum_{t=1}^{T}\left[\lambda_{TD}(V_{\boldsymbol{\phi}}(s_t) - q_t)^2 - \lambda_{PG}\min(\rho_t A_t, \tilde{\rho}_t A_t) - \lambda_{ent}\mathbb{H}(\pi_{\boldsymbol{\phi}}(\cdot|s_t))\right]$
    $\boldsymbol{\phi} := \boldsymbol{\phi} - \eta\nabla_{\boldsymbol{\phi}}\mathcal{L}(\boldsymbol{\phi})$

3.4.4 虚拟机配置

在本节中，我们讨论了 **VMPO** 算法的 [FS+19]，它是对早期在线策略 **MPO** 算法（ MAP 策略优化）的在线策略扩展，来自 [Abd+18]。它最初是用 “ 控制作为推理 ” 来解释的（参见第 1.5 节），但我们也可以将其视为基于方程（ 3.57 ）的约束策略改进方法。特别是，VMPO 利用了以下事实，如果

$$\mathbb{E}_{p^{\gamma}_{\pi_k}(s)}\left[D_{\mathbb{KL}}\left(\pi \parallel \pi_k\right)(s)\right] \leq \delta \tag{3.64}$$

然后 π 也满足 TV 约束，与 $\delta = \frac{\epsilon^2}{2}$。

请注意，与 3.4.2 节中的 TRPO 相比，这里的 KL 是相反的。这个新版本将鼓励，因此它将自然具有高熵，这可以提高鲁棒性。不幸的是，这种 KL 更难计算，因为我们是在对未知的分布 π 取期望。π。

为了解决这个问题，VMPO 采用了一种 EM 型方法。在 E 步中，我们计算了由未知新策略给出的状态 - 动作分布的非参数版本：

$$\psi(s,a) = \pi(a|s)p^{\gamma}_{\pi_k}(s) \tag{3.65}$$

最优的新分布由以下给出

$$\psi_{k+1} = \underset{\psi}{\operatorname{argmax}} \mathbb{E}_{\psi(s,a)}\left[A^{\pi_k}(s,a)\right] \quad \text{s.t.} \quad D_{\mathbb{KL}}\left(\psi \parallel \psi_k\right) \leq \delta \tag{3.66}$$

where $\psi_k(s,a) = \pi_k(a|s)p^\gamma_{\pi_k}(s)$. The solution to this is

$$\psi_{k+1}(s,a) = p^\gamma_{\pi_k}(s)\pi_k(a|s)w(s,a) \tag{3.67}$$

$$w(s,a) = \frac{\exp(A^{\pi_k}(s,a)/\lambda^*)}{Z(\lambda^*)} \tag{3.68}$$

$$Z(\lambda) = \mathbb{E}_{(s,a)\sim p^\gamma_{\pi_k}}\left[\exp(A^{\pi_k}(s,a)/\lambda)\right] \tag{3.69}$$

$$\lambda^* = \operatorname*{argmin}_{\lambda\geq 0} \lambda\delta + \lambda \log Z(\lambda) \tag{3.70}$$

In the M step, we project this target distribution back onto the space of parametric policies, while satisfying the KL trust region constraint:

$$\pi_{k+1} = \operatorname*{argmax}_\pi \mathbb{E}_{(s,a)\sim p^\gamma_{\pi_k}}\left[w(s,a)\log\pi(a|s)\right] \quad \text{s.t.} \quad \mathbb{E}_{p^\gamma_{\pi_k}}\left[D_{\mathbb{KL}}\left(\psi_k \parallel \psi\right)(s)\right] \leq \delta \tag{3.71}$$

3.5 Off-policy methods

In many cases, it is useful to train a policy using data collected from a distinct **behavior policy** $\pi_b(a|s)$ that is not the same as the **target policy** $\pi(a|s)$ that is being learned. For example, this could be data collected from earlier trials or parallel workers (with different parameters $\boldsymbol{\theta}'$) and stored in a **replay buffer**, or it could be **demonstration data** from human experts. This is known as **off-policy RL**, and can be much more sample efficient than the on-policy methods we have discussed so far, since these methods can use data from multiple sources. However, off-policy methods are more complicated, as we will explain below.

The basic difficulty is that the target policy that we want to learn may want to try an action in a state that has not been experienced before in the existing data, so there is no way to predict the outcome of this new (s,a) pair. In this section, we tackle this problem by assuming that the target policy is not too different from the behavior policy, so that the ratio $\pi(a|s)/\pi_b(a|s)$ is bounded, which allows us to use methods based on importance sampling. In the online learning setting, we can ensure this property by using conservative incremental updates to the policy. Alternatively we can use policy gradient methods with various regularization methods, as we discuss below.

In Section 5.5, we discuss offline RL, which is an extreme instance of off-policy RL where we have a fixed behavioral dataset, possibly generated from an unknown behavior policy, and can never collect any new data.

3.5.1 Policy evaluation using importance sampling

Assume we have a dataset of the form $\mathcal{D} = \{\boldsymbol{\tau}^{(i)}\}_{1\leq i\leq n}$, where each trajectory is a sequence $\boldsymbol{\tau}^{(i)} = (s_0^{(i)}, a_0^{(i)}, r_0^{(i)}, s_1^{(i)} \ldots)$, where the actions are sampled according to a behavior policy π_b, and the reward and next states are sampled according to the reward and transition models. We want to use this offline dataset to evaluate the performance of some target policy π; this is called **off-policy policy evaluation** or **OPE**. If the trajectories $\boldsymbol{\tau}^{(i)}$ were sampled from π. we could use the standard Monte Carlo estimate:

$$\hat{J}(\pi) \triangleq \frac{1}{n}\sum_{i=1}^{n}\sum_{t=0}^{T-1}\gamma^t r_t^{(i)} \tag{3.72}$$

However, since the trajectories are sampled from π_b, we use **importance sampling** (IS) to correct for the distributional mismatch, as first proposed in [PSS00]. This gives

$$\hat{J}_{\mathrm{IS}}(\pi) \triangleq \frac{1}{n}\sum_{i=1}^{n}\frac{p(\boldsymbol{\tau}^{(i)}|\pi)}{p(\boldsymbol{\tau}^{(i)}|\pi_b)}\sum_{t=0}^{T-1}\gamma^t r_t^{(i)} \tag{3.73}$$

It can be verified that $\mathbb{E}_{\pi_b}\left[\hat{J}_{\mathrm{IS}}(\pi)\right] = J(\pi)$, that is, $\hat{J}_{\mathrm{IS}}(\pi)$ is **unbiased**, provided that $p(\boldsymbol{\tau}|\pi_b) > 0$ whenever $p(\boldsymbol{\tau}|\pi) > 0$. The **importance ratio**, $\frac{p(\boldsymbol{\tau}^{(i)}|\pi)}{p(\boldsymbol{\tau}^{(i)}|\pi_b)}$, is used to compensate for the fact that the data is sampled

此处 $\psi_k(s,a) = \pi_k(a|s) p^{\gamma}_{\pi_k}(s)$ 的解为。

$$\psi_{k+1}(s,a) = p^{\gamma}_{\pi_k}(s)\pi_k(a|s)w(s,a) \tag{3.67}$$

$$w(s,a) = \frac{\exp(A^{\pi_k}(s,a)/\lambda^*)}{Z(\lambda^*)} \tag{3.68}$$

$$Z(\lambda) = \mathbb{E}_{(s,a)\sim p^{\gamma}_{\pi_k}}\left[\exp(A^{\pi_k}(s,a)/\lambda)\right] \tag{3.69}$$

$$\lambda^* = \operatorname*{argmin}_{\lambda\geq 0} \lambda\delta + \lambda \log Z(\lambda) \tag{3.70}$$

在 M 步骤中，我们将目标分布投影回参数策略空间，同时满足 KL 信任域约束：

$$\pi_{k+1} = \operatorname*{argmax}_{\pi} \mathbb{E}_{(s,a)\sim p^{\gamma}_{\pi_k}}\left[w(s,a)\log\pi(a|s)\right] \quad \text{s.t.} \quad \mathbb{E}_{p^{\gamma}_{\pi_k}}\left[D_{\mathbb{KL}}\left(\psi_k \parallel \psi\right)(s)\right] \leq \delta \tag{3.71}$$

3.5 离策略方法

在许多情况下，使用从不同的行为策略 （ $\pi_b(a|s)$ ）收集的数据来训练策略是有用的，这个行为策略与正在学习的目标策略 （ $\pi(a|s)$ ）不同。例如，这可能是从早期试验或并行工作者（具有不同的参数 $\boldsymbol{\theta}'$ ）收集的数据，并存储在 **重放缓冲区**中，或者可能是来自人类专家的 **演示数据**。这被称为 **离策略强化学习**，与之前讨论的在线策略方法相比，可以更有效地利用样本，因为这些方法可以使用来自多个来源的数据。然而，离策略方法更复杂，我们将在下面解释。

基本困难在于，我们想要学习的目标策略可能想要在现有数据中未经历过的状态下尝试一个动作，因此无法预测这个新（ s,a ）对的结局。在本节中，我们通过假设目标策略与行为策略不太不同来解决这个问题，使得 $\pi(a|s)/\pi_b(a|s)$ 的比率是有界的，这允许我们使用基于重要性抽样的方法。在在线学习设置中，我们可以通过使用对策略的保守增量更新来确保这一属性。或者，我们可以使用具有各种正则化方法的策略梯度方法，如下所述。

在章节 5.5 中，我们讨论了离线强化学习，这是离线策略强化学习的一个极端例子，其中我们有一个固定的行为数据集，可能是由一个未知的行为策略生成的，并且永远无法收集任何新数据。

3.5.1 使用重要性采样进行策略评估

假设我们有一个形式为 $\mathcal{D} = \{\boldsymbol{\tau}^{(i)}\}_{1\leq i\leq n}$的数据集，其中每个轨迹是一个序列 $\boldsymbol{\tau}^{(i)} = (s^{(i)}{}_0, a^{(i)}{}_0, r^{(i)}{}_0, s^{(i)}{}_1 \ldots)$，动作是根据行为策略 π_b 采样的，奖励和下一个状态是根据奖励和转换模型采样的。我们希望使用这个离线数据集来评估某些目标策略 π 的性能；这被称为**离线策略评估**或 **OPE**。如果轨迹 $\boldsymbol{\tau}^{(i)}$ 是从 π 采样的，我们可以使用标准的蒙特卡洛估计：

$$\hat{J}(\pi) \triangleq \frac{1}{n}\sum_{i=1}^{n}\sum_{t=0}^{T-1}\gamma^t r_t^{(i)} \tag{3.72}$$

然而，由于轨迹是从 π_b 中采样的，我们使用 **重要性采样**（ IS ）来校正分布不匹配，如首次在 [PSS00] 中提出。这给出了

$$\hat{J}_{\mathrm{IS}}(\pi) \triangleq \frac{1}{n}\sum_{i=1}^{n}\frac{p(\boldsymbol{\tau}^{(i)}|\pi)}{p(\boldsymbol{\tau}^{(i)}|\pi_b)}\sum_{t=0}^{T-1}\gamma^t r_t^{(i)} \tag{3.73}$$

可以验证，$\mathbb{E}_{\pi_b}\left[\hat{J}_{\mathrm{IS}}(\pi)\right] = J(\pi)$，即 $\hat{J}_{\mathrm{IS}}(\pi)$ 是**无偏**，前提是 $p(\boldsymbol{\tau}|\pi_b) > 0$ 每当 $p(\boldsymbol{\tau}|\pi) > 0$。重要性比，$\frac{p(\boldsymbol{\tau}^{(i)}|\pi}{)_p(\boldsymbol{\tau}^{(i)}|\pi_b)}$，用于补偿数据采样的

from π_b and not π. It can be simplified as follows:

$$\frac{p(\boldsymbol{\tau}|\pi)}{p(\boldsymbol{\tau}|\pi_b)} = \frac{p(s_0)\prod_{t=0}^{T-1}\pi(a_t|s_t)p_S(s_{t+1}|s_t,a_t)p_R(r_t|s_t,a_t,s_{t+1})}{p(s_0)\prod_{t=0}^{T-1}\pi_b(a_t|s_t)p_S(s_{t+1}|s_t,a_t)p_R(r_t|s_t,a_t,s_{t+1})} = \prod_{t=0}^{T-1}\frac{\pi(a_t|s_t)}{\pi_b(a_t|s_t)} \tag{3.74}$$

This simplification makes it easy to apply IS, as long as the target and behavior policies are known. (If the behavior policy is unknown, we can estimate it from $\mathcal{D}$, and replace π_b by its estimate $\hat{\pi}_b$. For convenience, define the **per-step importance ratio** at time t by

$$\rho_t(\boldsymbol{\tau}) \triangleq \pi(a_t|s_t)/\pi_b(a_t|s_t) \tag{3.75}$$

We can reduce the variance of the estimator by noting that the reward r_t is independent of the trajectory beyond time t. This leads to a **per-decision importance sampling** variant:

$$\hat{J}_{\text{PDIS}}(\pi) \triangleq \frac{1}{n}\sum_{i=1}^{n}\sum_{t=0}^{T-1}\prod_{t'\leq t}\rho_{t'}(\boldsymbol{\tau}^{(i)})\gamma^t r_t^{(i)} \tag{3.76}$$

3.5.2 Off-policy actor critic methods

In this section, we discuss how to extend actor-critic methods to work with off-policy data.

3.5.2.1 Learning the critic using V-trace

In this section we build on Section 3.5.1 to develop a practical method, known as **V-trace** [Esp+18], to estimate the value function for a target policy using off-policy data. (This is an extension of the earlier **Retrace** algorithm [Mun+16], which estimates the Q function using off-policy data.)

First consider the n-step target value for $V(s_i)$ in the on-policy case:

$$V_i = V(s_i) + \sum_{t=i}^{i+n-1}\gamma^{t-i}r_t + \gamma^n V(s_{i+n}) \tag{3.77}$$

$$= V(s_i) + \sum_{t=i}^{i+n-1}\gamma^{t-i}\underbrace{(r_t + \gamma V(s_{t+1}) - V(s_t))}_{\delta_t} \tag{3.78}$$

where we define $\delta_t = (r_t + \gamma V(s_{t+1}) - V(s_t))$ as the TD error at time t. To extend this to the off-policy case, we use the per-step importance ratio trick. However, to bound the variance of the estimator, we truncate the IS weights. In particular, we define

$$c_t = \min\left(\overline{c}, \frac{\pi(a_t|s_t)}{\pi_b(a_t|s_t)}\right), \ \rho_t = \min\left(\overline{\rho}, \frac{\pi(a_t|s_t)}{\pi_b(a_t|s_t)}\right) \tag{3.79}$$

where $\overline{c}$ and $\overline{\rho}$ are hyperparameters. We then define the V-trace target value for $V(s_i)$ as

$$v_i = V(s_i) + \sum_{t=i}^{i+n-1}\gamma^{t-i}\left(\prod_{t'=i}^{t-1}c_{t'}\right)\rho_t\delta_t \tag{3.80}$$

Note that we can compute these targets recursively using

$$v_i = V(s_i) + \rho_i\delta_i + \gamma c_i(v_{i+1} - V(s_{i+1})) \tag{3.81}$$

The product of the weights $c_i \ldots c_{t-1}$ (known as the "trace") measures how much a temporal difference δ_t at time t impacts the update of the value function at earlier time i. If the policies are very different, the

从 π_b 而不是 π。它可以简化如下：

$$\frac{p(\boldsymbol{\tau}|\pi)}{p(\boldsymbol{\tau}|\pi_b)} = \frac{p(s_0)\prod_{t=0}^{T-1}\pi(a_t|s_t)p_S(s_{t+1}|s_t,a_t)p_R(r_t|s_t,a_t,s_{t+1})}{p(s_0)\prod_{t=0}^{T-1}\pi_b(a_t|s_t)p_S(s_{t+1}|s_t,a_t)p_R(r_t|s_t,a_t,s_{t+1})} = \prod_{t=0}^{T-1}\frac{\pi(a_t|s_t)}{\pi_b(a_t|s_t)} \tag{3.74}$$

这种简化使得只要知道目标和行为策略，就可以轻松应用 IS。如果行为策略未知，我们可以从 $\mathcal{D}$ 中估计它，并用其估计值 $\hat{\pi_b}$ 替换 π_b 。为了方便，定义时间 t 的每步重要性比率为。

$$\rho_t(\boldsymbol{\tau}) \triangleq \pi(a_t|s_t)/\pi_b(a_t|s_t) \tag{3.75}$$

我们可以通过注意到奖励 r_t 与时间 t 之后的轨迹无关来降低估计量的方差。这导致了一个 **每次决策的重要性采样** 变体：

$$\hat{J}_{\text{PDIS}}(\pi) \triangleq \frac{1}{n}\sum_{i=1}^{n}\sum_{t=0}^{T-1}\prod_{t'\le t}\rho_{t'}(\boldsymbol{\tau}^{(i)})\gamma^t r_t^{(i)} \tag{3.76}$$

3.5.2 离策演员评论方法

本节中，我们讨论如何将 actor-critic 方法扩展到离线策略数据。

3.5.2.1 使用 V-trace 学习 the critic

在本节中，我们基于第 3.5.1 节，开发了一种实用方法，称为 **V-trace** [Esp+18]，用于使用离线策略数据估计目标策略的价值函数。（这是对早期 **Retrace** 算法 [Mun+16] 的扩展，该算法使用离线策略数据估计函数。）

首先考虑在策略不变情况下，n- 步目标值对于 $V(s_i)$。

$$V_i = V(s_i) + \sum_{t=i}^{i+n-1}\gamma^{t-i}r_t + \gamma^n V(s_{i+n}) \tag{3.77}$$

$$= V(s_i) + \sum_{t=i}^{i+n-1}\gamma^{t-i}\underbrace{(r_t + \gamma V(s_{t+1}) - V(s_t))}_{\delta_t} \tag{3.78}$$

我们在时间 t 定义 $\delta_t = (r_t + \gamma V(s_{t+1}) - V(s_t))$ 为 TD 错误。为了扩展到离策略情况，我们使用每步重要性比率的技巧。然而，为了限制估计量的方差，我们截断 IS 权重。特别是，我们定义

$$c_t = \min\left(c, \frac{\pi(a_t|s_t)}{\pi_b(a_t|s_t)}\right), \quad \rho_t = \min\left(\rho, \frac{\pi(a_t|s_t)}{\pi_b(a_t|s_t)}\right) \tag{3.79}$$

c 和 ρ 是超参数。然后我们定义 $V(s_i)$ 的 V-trace 目标值为

$$v_i = V(s_i) + \sum_{t=i}^{i+n-1}\gamma^{t-i}\left(\prod_{t'=i}^{t-1}c_{t'}\right)\rho_t\delta_t \tag{3.80}$$

请注意，我们可以递归地计算这些目标

$$v_i = V(s_i) + \rho_i\delta_i + \gamma c_i(v_{i+1} - V(s_{i+1})) \tag{3.81}$$

权重乘积（称为 “ 迹 ” ）衡量时间差 δ_t 在时间 t 对早期时间 i 的价值函数更新的影响程度。如果策略非常不同，那么

variance of this product will be large. So the truncation parameter $\overline{c}$ is used to reduce the variance. In [Esp+18], they find $\overline{c} = 1$ works best.

The use of the target $\rho_t \delta_t$ rather than δ_t means we are evaluating the value function for a policy that is somewhere between π_b and π. For $\overline{\rho} = \infty$ (i.e., no truncation), we converge to the value function V^π, and for $\overline{\rho} \to 0$, we converge to the value function V^{π_b}. In [Esp+18], they find $\overline{\rho} = 1$ works best.

Note that if $\overline{c} = \overline{\rho}$, then $c_i = \rho_i$. This gives rise to the simplified form

$$v_t = V(s_t) + \sum_{j=0}^{n-1} \gamma^j \left(\prod_{m=0}^{j} c_{t+m} \right) \delta_{t+j} \tag{3.82}$$

We can use the above V-trace targets to learn an approximate value function by minimizing the usual ℓ_2 loss:

$$\mathcal{L}(\boldsymbol{w}) = \mathbb{E}_{t \sim \mathcal{D}} \left[(v_t - V_{\boldsymbol{w}}(s_t))^2 \right] \tag{3.83}$$

the gradient of which has the form

$$\nabla \mathcal{L}(\boldsymbol{w}) = 2\mathbb{E}_{t \sim \mathcal{D}} \left[(v_t - V_{\boldsymbol{w}}(s_t)) \nabla_{\boldsymbol{w}} V_{\boldsymbol{w}}(s_t) \right] \tag{3.84}$$

3.5.2.2 Learning the actor

We now discuss how to update the actor using an off-policy estimate of the policy gradient. We start by defining the objective to be the expected value of the new policy, where the states are drawn from the behavior policy's state distribution, but the actions are drawn from the target policy:

$$J_{\pi_b}(\pi_{\boldsymbol{\theta}}) = \sum_s p^\gamma_{\pi_b}(s) V_\pi(s) = \sum_s p^\gamma_{\pi_b}(s) \sum_a \pi_{\boldsymbol{\theta}}(a|s) Q_\pi(s,a) \tag{3.85}$$

Differentiating this and ignoring the term $\nabla_{\boldsymbol{\theta}} Q_\pi(s,a)$, as suggested by [DWS12], gives a way to (approximately) estimate the **off-policy policy-gradient** using a one-step IS correction ratio:

$$\nabla_{\boldsymbol{\theta}} J_{\pi_b}(\pi_{\boldsymbol{\theta}}) \approx \sum_s \sum_a p^\gamma_{\pi_b}(s) \nabla_{\boldsymbol{\theta}} \pi_{\boldsymbol{\theta}}(a|s) Q_\pi(s,a) \tag{3.86}$$

$$= \mathbb{E}_{p^\gamma_{\pi_b}(s), \pi_b(a|s)} \left[\frac{\pi_{\boldsymbol{\theta}}(a|s)}{\pi_b(a|s)} \nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(a|s) Q_\pi(s,a) \right] \tag{3.87}$$

In practice, we can approximate $Q_\pi(s_t, a_t)$ by $q_t = r_t + \gamma v_{t+1}$, where v_{t+1} is the V-trace estimate for state s_{t+1}. If we use $V(s_t)$ as a baseline, to reduce the variance, we get the following gradient estimate for the policy:

$$\nabla J(\boldsymbol{\theta}) = \mathbb{E}_{t \sim \mathcal{D}} \left[\rho_t \nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(a_t|s_t) (r_t + \gamma v_t - V_{\boldsymbol{w}}(s_t)) \right] \tag{3.88}$$

We can also replace the 1-step IS-weighted TD error $\rho_t(r_t + \gamma v_t - V_{\boldsymbol{w}}(s_t))$ with an IS-weighted GAE value by modifying the generalized advantage estimation method in Section 3.3.2. In particular, we just need to define $\lambda_t = \lambda \min(1, \rho_t)$. We denote the IS-weighted GAE estimate as A^ρ_t.[4]

3.5.2.3 IMPALA

As an example of an off-policy AC method, we consider **IMPALA**, which stands for "Importance Weighted Actor-Learning Architecture". [Esp+18]. This uses shared parameters for the policy and value function (with different output heads), and adds an entropy bonus to ensure the policy remains stochastic. Thus we end up with the following objective, which is very similar to on-policy actor-critic shown in Algorithm 6:

$$\mathcal{L}(\boldsymbol{\phi}) = \mathbb{E}_{t \sim \mathcal{D}} \left[\lambda_{TD} (V_{\boldsymbol{\phi}}(s_t) - v_t)^2 - \lambda_{PG} A^\rho_t \log \pi_{\boldsymbol{\phi}}(a_t|s_t) - \lambda_{ent} \mathbb{H}(\pi_{\boldsymbol{\phi}}(\cdot|s_t)) \right] \tag{3.89}$$

[4] For an implementation, see https://github.com/google-deepmind/rlax/blob/master/rlax/_src/multistep.py#L39

该产品的方差将很大。因此，使用截断参数 c 来减少方差。在 [Esp+18] 中，他们发现 $c = 1$ 效果最佳。

使用目标 $\rho_t \delta_t$ 而不是 δ_t 意味着我们正在评估介于 π_b 和 π 之间的策略的价值函数。对于 $\rho = \infty$ （即无截断），我们收敛到价值函数 V^{π}，而对于 $\rho \to 0$，我们收敛到价值函数 V^{π_b}。在 [Esp+18] 中，他们发现 $\rho = 1$ 效果最佳。

请注意，如果 $c = \rho$，则 $c_i = \rho_i$。这导致了简化的形式

$$v_t = V(s_t) + \sum_{j=0}^{n-1} \gamma^j \left(\prod_{m=0}^{j} c_{t+m} \right) \delta_{t+j} \tag{3.82}$$

我们可以使用上述 V-trace 目标通过最小化通常的 ℓ_2 损失来学习一个近似值函数：

$$\mathcal{L}(\boldsymbol{w}) = \mathbb{E}_{t \sim \mathcal{D}} \left[(v_t - V_{\boldsymbol{w}}(s_t))^2 \right] \tag{3.83}$$

该梯度的形式

$$\nabla \mathcal{L}(\boldsymbol{w}) = 2\mathbb{E}_{t \sim \mathcal{D}} \left[(v_t - V_{\boldsymbol{w}}(s_t)) \nabla_{\boldsymbol{w}} V_{\boldsymbol{w}}(s_t) \right] \tag{3.84}$$

3.5.2.2 学习演员

我们现在讨论如何使用离策略估计的策略梯度来更新演员。我们首先定义目标为新策略的期望值，其中状态是从行为策略的状态分布中抽取的，但动作是从目标策略中抽取的：

$$J_{\pi_b}(\pi_{\boldsymbol{\theta}}) = \sum_s p^{\gamma}_{\pi_b}(s) V_{\pi}(s) = \sum_s p^{\gamma}_{\pi_b}(s) \sum_a \pi_{\boldsymbol{\theta}}(a|s) Q_{\pi}(s,a) \tag{3.85}$$

区分此项并忽略项 $\nabla_{\boldsymbol{\theta}} Q_{\pi}$ （s，a），如 [DWS12], 所建议，提供了一种使用一步 IS 校正比率的（近似）估计离策略策略梯度的方法：

$$\nabla_{\boldsymbol{\theta}} J_{\pi_b}(\pi_{\boldsymbol{\theta}}) \approx \sum_s \sum_a p^{\gamma}_{\pi_b}(s) \nabla_{\boldsymbol{\theta}} \pi_{\boldsymbol{\theta}}(a|s) Q_{\pi}(s,a) \tag{3.86}$$

$$= \mathbb{E}_{p^{\gamma}_{\pi_b}(s), \pi_b(a|s)} \left[\frac{\pi_{\boldsymbol{\theta}}(a|s)}{\pi_b(a|s)} \nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(a|s) Q_{\pi}(s,a) \right] \tag{3.87}$$

在实践中，我们可以将 $Q_{\pi}(s_t,a_t)$ 近似为 $q_t = r_t + \gamma v_{t+1}$，其中 v_{t+1} 是状态 s_{t+1} 的 V-trace 估计。如果我们使用 $V(s_t)$ 作为基线，以减少方差，我们得到以下策略的梯度估计：

$$\nabla J(\boldsymbol{\theta}) = \mathbb{E}_{t \sim \mathcal{D}} \left[\rho_t \nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(a_t|s_t) (r_t + \gamma v_t - V_{\boldsymbol{w}}(s_t)) \right] \tag{3.88}$$

我们还可以通过修改第 3.3.2 节中的广义优势估计方法，用 IS 权重的 GAE 值替换 1 步 IS 权重 TD 错误 $\rho_t(r_t + \gamma v_t - V_{\boldsymbol{w}}(s_t))$。特别是，我们只需要定义 $\lambda_t = \lambda \min(1, \rho_t)$。我们表示 IS 权重的 GAE 估计为 $A^{\rho}{}_t$。[4]

3.5.2.3 IMPALA（由于“IMPALA”是一个专有名词，因此保留原样）

作为一个离策略 AC 方法的例子，我们考虑 **IMPALA**，它代表 “ 重要性加权演员学习架构 ”。[Esp+18]。这种方法使用共享参数来表示策略和价值函数（具有不同的输出头），并添加熵奖励以确保策略保持随机性。因此，我们得到以下目标，这与算法 6 中展示的在线策略演员 - 评论家非常相似：

$$\mathcal{L}(\boldsymbol{\phi}) = \mathbb{E}_{t \sim \mathcal{D}} \left[\lambda_{TD} (V_{\boldsymbol{\phi}}(s_t) - v_t)^2 - \lambda_{PG} A^{\rho}_t \log \pi_{\boldsymbol{\phi}}(a_t|s_t) - \lambda_{ent} \mathbb{H}(\pi_{\boldsymbol{\phi}}(\cdot|s_t)) \right] \tag{3.89}$$

[4] For an implementation, see https://github.com/google-deepmind/rlax/blob/master/rlax/_src/multistep.py#L39

The only difference from standard A2C is that we need to store the probabilities of each action, $\pi_b(a_t|s_t)$, in addition to (s_t, a_t, r_t, s_{t+1}) in the dataset $\mathcal{D}$, which can be used to compute ρ_t. [Esp+18] was able to use this method to train a single agent (using a shared CNN and LSTM for both value and policy) to play all 57 games at a high level. Furthermore, they showed that their method — thanks to its off-policy corrections — outperformed the A3C method (a parallel version of A2C) in Section 3.3.1.

3.5.3 Off-policy policy improvement methods

So far we have focused on actor-critic methods. However, policy improvement methods, such as PPO, are often preferred to AC methods, since they monotonically improve the objective. In [QPC21] they propose one way to extend PPO to the off-policy case. This method was generalized in [QPC24] to cover a variety of policy improvement algorithms, including TRPO and VMPO. We give a brief summary below.

The key insight is to realize that we can generalize the lower bound in Equation (3.56) to any reference policy

$$J(\pi) - J(\pi_k) \geq \frac{1}{1-\gamma}\mathbb{E}_{p^\gamma_{\pi_{\text{ref}}}(s)\pi_k(a|s)}\left[\frac{\pi(a|s)}{\pi_{\text{ref}}(a|s)}A^{\pi_k}(s,a)\right] - \frac{2\gamma C^{\pi,\pi_k}}{(1-\gamma)^2}\mathbb{E}_{p^\gamma_{\pi_{\text{ref}}}(s)}\left[\text{TV}(\pi(\cdot|s), \pi_{\text{ref}}(\cdot|s))\right] \quad (3.90)$$

The reference policy can be any previous policy, or a convex combination of them. In particular, if π_k is the current policy, we can consider the reference policy to be $\pi_{\text{ref}} = \sum_{i=1}^M \nu_i \pi_{k-i}$, where $0 \leq \nu_i \leq 1$ and $\sum_i \nu_i = 1$ are mixture weights. We can approximate the expectation by sampling from the replay buffer, which contains samples from older policies. That is, $(s,a) \sim p^\gamma_{\pi_{\text{ref}}}$ can be implemented by $i \sim \nu$ and $(s,a) \sim p^\gamma_{\pi_{k-i}}$.

To compute the advantage function A^{π_k} from off policy data, we can adapt the V-trace method of Equation (3.82) to get

$$A^{\pi_k}_{\text{trace}}(s_t, a_t) = \delta_t + \sum_{j=0}^{n-1}\gamma^j\left(\prod_{m=1}^{j} c_{t+m}\right)\delta_{t+j} \quad (3.91)$$

where $\delta_t = r_t + \gamma V(s_{t+1}) - V(s_t)$, and $c_t = \min\left(\overline{c}, \frac{\pi_k(a_t|s_t)}{\pi_{k-i}(a_t|s_t)}\right)$ is the truncated importance sampling ratio.

To compute the TV penalty term from off policy data, we need to choose between the PPO (Section 3.4.3), VMPO (Section 3.4.4) and TRPO (Section 3.4.2) approach. We discuss each of these cases below.

3.5.3.1 Off-policy PPO

The simplest is to use off-policy PPO, which gives an update of the following form (known as **Generalized PPO**):

$$\pi_{k+1} = \underset{\pi}{\text{argmax}}\, \mathbb{E}_{i\sim\nu}\left[\mathbb{E}_{(s,a)\sim p^\gamma_{\pi_{k-i}}}\left[\min(\rho_{k-i}(s,a)A^{\pi_k}(s,a), \tilde{\rho}_{k-i}(s,a)A^{\pi_k}(s,a))\right]\right] \quad (3.92)$$

where $\rho_{k-i}(s,a) = \frac{\pi(a|s)}{\pi_{k-i}(a|s)}$ and $\tilde{\rho}_{k-i}(s,a) = \text{clip}(\frac{\pi(a|s)}{\pi_{k-i}(a|s)}, l, u)$, where $l = \frac{\pi_k(a|s)}{\pi_{k-i}(a|s)} - \epsilon$ and $u = \frac{\pi_k(a|s)}{\pi_{k-i}(a|s)} + \epsilon$. (For other off-policy variants of PPO, see e.g., [Men+23; LMW24].)

3.5.3.2 Off-policy VMPO

For an off-policy version of VMPO, see the original **MPO** method of [Abd+18]; this is derived using an EM framework, but EM is just another bound optimization algorithm [HL04], and the result is equivalent to the version presented in [QPC24].

3.5.3.3 Off-policy TRPO

For details on the off-policy version of TRPO, see [QPC24].

与标准 A2C 的唯一区别是我们需要存储每个动作的概率，$\pi_b(a_t|s_t)$，除了在数据集 (s_t,a_t,r_t,s_{t+1}) 中，还可以用来计算 $\mathcal{D}$。ρ_t。[Esp+18] 能够使用这种方法训练单个智能体（使用共享的 CNN 和 LSTM 来处理价值和策略）以高水平玩所有 57 场比赛。此外，他们还表明，他们的方法 —— 得益于其离线策略校正 —— 在章节 3.3.1 中优于 A3C 方法（ A2C 的并行版本）。

3.5.3 离策略策略改进方法

到目前为止，我们一直关注演员 - 评论家方法。然而，由于它们单调地提高目标，策略改进方法，如 PPO，通常比 AC 方法更受欢迎。在 [QPC21] 中，他们提出了一种将 PPO 扩展到离策略情况的方法。这种方法在 [QPC24] 中被推广，以涵盖各种策略改进算法，包括 TRPO 和 VMPO。以下是我们对此的简要总结。

关键洞见在于意识到我们可以将方程（ 3.56 ）的下界推广到任何参考策略

$$J(\pi) - J(\pi_k) \geq \frac{1}{1-\gamma} \mathbb{E}_{p^\gamma_{\pi_{\text{ref}}}(s)\pi_k(a|s)} \left[\frac{\pi(a|s)}{\pi_{\text{ref}}(a|s)} A^{\pi_k}(s,a) \right] - \frac{2\gamma C^{\pi,\pi_k}}{(1-\gamma)^2} \mathbb{E}_{p^\gamma_{\pi_{\text{ref}}}(s)} \left[\text{TV}(\pi(\cdot|s), \pi_{\text{ref}}(\cdot|s)) \right] \quad (3.90)$$

参考策略可以是任何之前的策略，或者它们的凸组合。特别是，如果 π_k 是当前策略，我们可以考虑参考策略为 $\pi_{\text{ref}} = \sum_{i=1}^M \nu_i \pi_{k-i}$，其中 $0 \leq \nu_i \leq 1$ 和 $\sum_i \nu_i = 1$ 是混合权重。我们可以通过从包含旧策略样本的重放缓冲区中采样来近似期望。也就是说，$(s,a) \sim p^\gamma_{\pi_{\text{ref}}}$ 可以通过 $i \sim \nu$ 实现，而 $(s,a) \sim p^\gamma_{\pi_{k-i}}$。

从离策略数据中计算优势函数 A^{π_k} ，我们可以将方程（ 3.82 ）的 V-trace 方法进行适配以获得

$$A^{\pi_k}_{\text{trace}}(s_t,a_t) = \delta_t + \sum_{j=0}^{n-1} \gamma^j \left(\prod_{m=1}^{j} c_{t+m} \right) \delta_{t+j} \quad (3.91)$$

$\delta_t = r_t + \gamma V(s_{t+1}) - V(s_t)$，并且 $c_t = \min\left(c, \frac{\pi_k(a_t|s_t)}{\pi_{k-i}(a_t|s_t)}\right)$ 是截断的重要性采样比

.

从离策略数据中计算 TV 惩罚项，我们需要在 PPO （第 3.4.3 节）、 VMPO （第 3.4.4 节）和 TRPO （第 3.4.2 节）方法之间进行选择。以下我们将讨论这些情况。

3.5.3.1 离策略 PPO

最简单的是使用离策略 PPO，它给出以下形式的更新（称为**广义 PPO** ）：

$$\pi_{k+1} = \underset{\pi}{\text{argmax}}\, \mathbb{E}_{i\sim\nu} \left[\mathbb{E}_{(s,a)\sim p^\gamma_{\pi_{k-i}}} \left[\min(\rho_{k-i}(s,a) A^{\pi_k}(s,a), \tilde{\rho}_{k-i}(s,a) A^{\pi_k}(s,a)) \right] \right] \quad (3.92)$$

在 $\rho_{k-i}(s,a) = \frac{\pi(a|s)}{\pi_{k-i}(a|s)}$ 和 $\tilde{\rho}_{k-i}(s,a) = \text{clip}(\frac{\pi(a|s)}{\pi_k(a|s)}, l, u)$，其中 $l = \frac{\pi_k(a|s)}{\pi_k(a|s)} - \epsilon$ 和 $u = \frac{\pi_k(a|s)}{\pi_k(a|s)} + \epsilon$。（有关 PPO 的其他离策略变体，例如，参见 [Men+23;LMW24]。）

3.5.3.2 非策略 VMPO

关于 VMPO 的离策略版本，请参阅原文 **MPO** 方法 [Abd+18]；这是使用 EM 框架推导出来的，但 EM 只是另一种有界优化算法 [HL04], ，并且结果与在 [QPC24] 中提出的版本等效。

3.5.3.3 偏策略 TRPO

关于 TRPO 离策略版本的详细信息，请参阅 [QPC 24].

3.5.4 Soft actor-critic (SAC)

The **soft actor-critic** (**SAC**) algorithm [Haa+18a; Haa+18b] is an off-policy actor-critic method based on a framework known as maximum entropy RL, which we introduced in Section 1.5.3. Crucially, even though SAC is off-policy and utilizes a replay buffer to sample past experiences, the policy update is done using the actor's own probability distribution, eliminating the need to use importance sampling to correct for discrepancies between the behavior policy (used to collect data) and the target policy (used for updating), as we will see below.

We start by slightly rewriting the maxent RL objective from Equation (1.67) using modified notation:

$$J^{\text{SAC}}(\boldsymbol{\theta}) \triangleq \mathbb{E}_{p^{\gamma}_{\pi_{\boldsymbol{\theta}}}(s)\pi_{\boldsymbol{\theta}}(a|s)}\left[R(s,a) + \alpha\, \mathbb{H}(\pi_{\boldsymbol{\theta}}(\cdot|s))\right] \tag{3.93}$$

Note that the entropy term makes the objective easier to optimize, and encourages exploration.

To optimize this, we can perform a soft policy evaluation step, and then a soft policy improvement step. In the policy evaluation step, we can repeatedly apply a modified Bellman backup operator $\mathcal{T}^{\pi}$ defined as

$$\mathcal{T}^{\pi} Q(\boldsymbol{s}_t, \boldsymbol{a}_t) = r(\boldsymbol{s}_t, \boldsymbol{a}_t) + \gamma \mathbb{E}_{\boldsymbol{s}_{t+1} \sim p}\left[V(\boldsymbol{s}_{t+1})\right] \tag{3.94}$$

where

$$V(\boldsymbol{s}_t) = \mathbb{E}_{\boldsymbol{a}_t \sim \pi}\left[Q(\boldsymbol{s}_t, \boldsymbol{a}_t) - \alpha \log \pi(\boldsymbol{a}_t|\boldsymbol{s}_t)\right] \tag{3.95}$$

is the **soft value function**. If we iterate $Q^{k+1} = \mathcal{T}^{\pi} Q^k$,, this will converge to the soft Q function for π.

In the policy improvement step, we derive the new policy based on the soft Q function by softmaxing over the possible actions for each state. We then project the update back on to the policy class Π:

$$\pi_{\text{new}} = \arg\min_{\pi' \in \Pi} D_{\mathbb{KL}}\left(\pi'(\cdot|\boldsymbol{s}_t) \parallel \frac{\exp(\frac{1}{\alpha} Q^{\pi_{\text{old}}}(\boldsymbol{s}_t, \cdot))}{Z^{\pi_{\text{old}}}(\boldsymbol{s}_t)}\right) \tag{3.96}$$

(The partition function $Z^{\pi_{\text{old}}}(\boldsymbol{s}_t)$ may be intractable to compute for a continuous action space, but it cancels out when we take the derivative of the objective, so this is not a problem, as we show below.) After solving the above optimization problem, we are guaranteed to satisfy the soft policy improvement theorem, i.e., $Q^{\pi_{\text{new}}}(\boldsymbol{s}_t, \boldsymbol{a}_t) \geq Q^{\pi_{\text{old}}}(\boldsymbol{s}_t, \boldsymbol{a}_t)$ for all $\boldsymbol{s}_t$ and $\boldsymbol{a}_t$.

The above equations are intractable in the non-tabular case, so we now extend to the setting where we use function approximation.

3.5.4.1 Policy evaluation

For policy evaluation, we hold the policy parameters π fixed and optimize the parameters $\boldsymbol{w}$ of the Q function by minimizing the soft Bellman residual

$$J_Q(\boldsymbol{w}) = \mathbb{E}_{(\boldsymbol{s}_t, \boldsymbol{a}_t, r_{t+1}, \boldsymbol{s}_{t+1}) \sim \mathcal{D}}\left[\frac{1}{2}\left(Q_{\boldsymbol{w}}(\boldsymbol{s}_t, \boldsymbol{a}_t) - q(r_{t+1}, \boldsymbol{s}_{t+1})\right)^2\right] \tag{3.97}$$

where $\mathcal{D}$ is a replay buffer,

$$q(r_{t+1}, \boldsymbol{s}_{t+1}) = r_{t+1} + \gamma V_{\overline{\boldsymbol{w}}}(\boldsymbol{s}_{t+1}) \tag{3.98}$$

is the frozen target value, and and $V_{\overline{\boldsymbol{w}}}(\boldsymbol{s})$ is a frozen version of the soft value function from Equation (3.95):

$$V_{\overline{\boldsymbol{w}}}(\boldsymbol{s}_t) = \mathbb{E}_{\pi(\boldsymbol{a}_t|\boldsymbol{s}_t)}\left[Q_{\overline{\boldsymbol{w}}}(\boldsymbol{s}_t, \boldsymbol{a}_t) - \alpha \log \pi(\boldsymbol{a}_t|\boldsymbol{s}_t)\right] \tag{3.99}$$

where $\overline{\boldsymbol{w}}$ is the EMA version of $\boldsymbol{w}$. (The use of a frozen target is to avoid bootstrapping instablilities discussed in Section 2.5.2.4.)

To avoid the positive overestimation bias that can occur with actor-critic methods, [Haa+18a], suggest fitting two soft Q functions, by optimizing $J_Q(\boldsymbol{w}_i)$, for $i = 1, 2$, independently. Inspired by clipped double Q learning, used in TD3 (Section 3.6.2), the targets are defined as

$$q(r_{t+1}, \boldsymbol{s}_{t+1}; \overline{\boldsymbol{w}}_{1:2}, \boldsymbol{\theta}) = r_{t+1} + \gamma\left[\min_{i=1,2} Q_{\overline{\boldsymbol{w}}_i}(\boldsymbol{s}_{t+1}, \tilde{\boldsymbol{a}}_{t+1}) - \alpha \log \pi_{\boldsymbol{\theta}}(\tilde{\boldsymbol{a}}_{t+1}|\boldsymbol{s}_{t+1})\right] \tag{3.100}$$

3.5.4 软演员 - 评论家（ SAC ）

软演员 - 评论家（ **SAC** ）算法 **Haa** ； [Haa+18b 是一个基于称为最大熵强化学习（ RL ）框架的离策略演员 - 评论家方法，我们在第 1.5.3 节中介绍了它。关键的是，尽管 SAC 是离策略的，并使用重放缓冲区来采样过去经验，但策略更新是使用演员自己的概率分布进行的，从而消除了使用重要性采样来纠正行为策略（用于收集数据）和目标策略（用于更新）之间差异的需要，如下所示。

我们首先使用修改后的符号，对公式（ 1.67 ）中的最大熵 RL 目标进行轻微改写：

$$J^{\mathrm{SAC}}(\boldsymbol{\theta}) \triangleq \mathbb{E}_{p^{\gamma}_{\pi_{\boldsymbol{\theta}}}(s)\pi_{\boldsymbol{\theta}}(a|s)}\left[R(s,a)+\alpha\,\mathbb{H}(\pi_{\boldsymbol{\theta}}(\cdot|s))\right] \tag{3.93}$$

注意，熵项使得目标更容易优化，并鼓励探索。

为了优化这一点，我们可以执行软策略评估步骤，然后执行软策略改进步骤。在策略评估步骤中，我们可以反复应用一个修改后的 Bellman 备份算子 $\mathcal{T}^{\pi}$ ，定义为

$$\mathcal{T}^{\pi}Q(\boldsymbol{s}_t,\boldsymbol{a}_t) = r(\boldsymbol{s}_t,\boldsymbol{a}_t)+\gamma\mathbb{E}_{\boldsymbol{s}_{t+1}\sim p}\left[V(\boldsymbol{s}_{t+1})\right] \tag{3.94}$$

在哪里

$$V(\boldsymbol{s}_t) = \mathbb{E}_{\boldsymbol{a}_t\sim\pi}\left[Q(\boldsymbol{s}_t,\boldsymbol{a}_t)-\alpha\log\pi(\boldsymbol{a}_t|\boldsymbol{s}_t)\right] \tag{3.95}$$

是 **软值函数**。如果我们迭代 $Q^{k+1}=\mathcal{T}^{\pi}Q^{k}$ “，这将收敛到软 Q 函数 π。

在策略改进步骤中，我们根据软 Q 函数通过在每个状态的可能动作上应用 softmax 导出新的策略。然后，我们将更新投影回策略类 Π：

$$\pi_{\mathrm{new}} = \arg\min_{\pi'\in\Pi} D_{\mathbb{KL}}\left(\pi'(\cdot|\boldsymbol{s}_t)\;\|\;\frac{\exp(\frac{1}{\alpha}Q^{\pi_{\mathrm{old}}}(\boldsymbol{s}_t,\cdot))}{Z^{\pi_{\mathrm{old}}}(\boldsymbol{s}_t)}\right) \tag{3.96}$$

（连续动作空间中的分函数 $Z^{\pi_{\mathrm{old}}}(\boldsymbol{s}_t)$ 可能难以计算，但在对目标函数求导时它会相互抵消，因此这不是问题，如下所示。）

在解决上述优化问题后，我们保证满足软策略改进定理，即对于所有 $\boldsymbol{s}_t$ 和 $\boldsymbol{a}_t$，有 $Q^{\pi_{\mathrm{new}}}(\boldsymbol{s}_t,\boldsymbol{a}_t)\geq Q^{\pi_{\mathrm{old}}}(\boldsymbol{s}_t,\boldsymbol{a}_t)$。

上述方程在非表格情况下难以处理，因此我们现在扩展到使用函数逼近的设置。

3.5.4.1 政策评估

为了策略评估，我们保持策略参数 π 固定，并通过最小化软贝尔曼残差 $\left[\quad\right]$ 来优化 Q 函数的参数 $\boldsymbol{w}$ 。

$$J_Q(\boldsymbol{w}) = \mathbb{E}_{(\boldsymbol{s}_t,\boldsymbol{a}_t,r_{t+1},\boldsymbol{s}_{t+1})\sim\mathcal{D}}\left[\frac{1}{2}\left(Q_{\boldsymbol{w}}(\boldsymbol{s}_t,\boldsymbol{a}_t)-q(r_{t+1},\boldsymbol{s}_{t+1})\right)^2\right] \tag{3.97}$$

$\mathcal{D}$ 是一个重放缓冲区，

$$q(r_{t+1},\boldsymbol{s}_{t+1}) = r_{t+1}+\gamma V_{\boldsymbol{w}}(\boldsymbol{s}_{t+1}) \tag{3.98}$$

是冻结的目标值，而和 $V_{\boldsymbol{w}}$ （ $\boldsymbol{s}$ ）是方程（ 3.95 ）中软值函数的冻结版本：

$$V_{\boldsymbol{w}}(\boldsymbol{s}_t) = \mathbb{E}_{\pi(\boldsymbol{a}_t|\boldsymbol{s}_t)}\left[Q_{\boldsymbol{w}}(\boldsymbol{s}_t,\boldsymbol{a}_t)-\alpha\log\pi(\boldsymbol{a}_t|\boldsymbol{s}_t)\right] \tag{3.99}$$

其中 $\boldsymbol{w}$ 是 $\boldsymbol{w}$ 的 EMA 版本。使用冻结目标是为了避免在 2.5.2.4 节中讨论的自举不稳定性。 ()

为了避免与 actor-critic 方法相关的正偏差估计，[Haa+18a]，建议独立拟合两个软 Q 函数，通过优化 $J_Q(\boldsymbol{w}_i)$，对于 $i=1$，2。受 TD3 （第 Q3.6.2 ）中使用的 clipped double learning 的启发，目标被定义为 $\left[\quad\right]$

$$q(r_{t+1},\boldsymbol{s}_{t+1};\boldsymbol{w}_{1:2},\boldsymbol{\theta}) = r_{t+1}+\gamma\left[\min_{i=1,2}Q_{\boldsymbol{w}_i}(\boldsymbol{s}_{t+1},\tilde{\boldsymbol{a}}_{t+1})-\alpha\log\pi_{\boldsymbol{\theta}}(\tilde{\boldsymbol{a}}_{t+1}|\boldsymbol{s}_{t+1})\right] \tag{3.100}$$

where $\tilde{\boldsymbol{a}}_{t+1} \sim \pi_{\boldsymbol{\theta}}(\boldsymbol{s}_{t+1})$ is a sampled next action. In [Che+20], they propose the REDQ method (Section 2.5.3.3) which uses a random ensemble of $N \geq 2$ networks instead of just 2.

3.5.4.2 Policy improvement: Gaussian policy

For policy improvement, we hold the value function parameters $\boldsymbol{w}$ fixed and optimize the parameters $\boldsymbol{\theta}$ of the policy by minimizing the objective below, which is derived from the KL term by multiplying by α and dropping the constant Z term:

$$J_\pi(\boldsymbol{\theta}) = \mathbb{E}_{\boldsymbol{s}_t \sim \mathcal{D}} \left[\mathbb{E}_{\boldsymbol{a}_t \sim \pi_{\boldsymbol{\theta}}} \left[\alpha \log \pi_{\boldsymbol{\theta}}(\boldsymbol{a}_t|\boldsymbol{s}_t) - Q_{\boldsymbol{w}}(\boldsymbol{s}_t, \boldsymbol{a}_t) \right] \right] \tag{3.101}$$

Since we are taking gradients wrt $\boldsymbol{\theta}$, which affects the inner expectation term, we need to either use the REINFORCE estimator from Equation (3.15) or the **reparameterization trick** (see e.g., [Moh+20]). The latter is much lower variance, so is preferable.

To explain this in more detail, let us assume the policy distribution has the form $\pi_{\boldsymbol{\theta}}(\boldsymbol{a}_t|\boldsymbol{s}_t) = \mathcal{N}(\boldsymbol{\mu}_{\boldsymbol{\theta}}(\boldsymbol{s}_t), \sigma^2 \mathbf{I})$. We can write the random action as $\boldsymbol{a}_t = f_{\boldsymbol{\theta}}(\boldsymbol{s}_t, \boldsymbol{\epsilon}_t)$, where f is a deterministic function of the state and a noise variable $\boldsymbol{\epsilon}_t$, since $\boldsymbol{a}_t = \boldsymbol{\mu}(\boldsymbol{s}_t) + \sigma^2 \boldsymbol{\epsilon}_t$, where $\boldsymbol{\epsilon}_t \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$. The objective now becomes

$$J_\pi(\boldsymbol{\theta}) = \mathbb{E}_{\boldsymbol{s}_t \sim \mathcal{D}, \boldsymbol{\epsilon}_t \sim \mathcal{N}} \left[\alpha \log \pi_{\boldsymbol{\theta}}(f_{\boldsymbol{\theta}}(\boldsymbol{s}_t, \boldsymbol{\epsilon}_t)|\boldsymbol{s}_t) - Q_{\boldsymbol{w}}(\boldsymbol{s}_t, f_{\boldsymbol{\theta}}(\boldsymbol{s}_t, \boldsymbol{\epsilon}_t)) \right] \tag{3.102}$$

where we have replaced the expectation of $\boldsymbol{a}_t$ wrt $\pi_{\boldsymbol{\theta}}$ with an expectation of $\boldsymbol{\epsilon}_t$ wrt its noise distribution $\mathcal{N}$. Hence we can now safely take stochastic gradients. See Algorithm 8 for the pseudocode. (Note that, for discrete actions, we can avoid the need for the reparameterization trick by computing the expectations explicitly, as discussed in Section 3.5.4.3.)

3.5.4.3 Policy improvement: softmax policy

For discrete actions, we can replace the Gaussian reparameterization with the gumbel-softmax reparameterization [JGP16; MMT17]. Alternatively, we can eschew sampling and compute the expectations over the actions explicitly, to derive lower variance versions of the equations; this is known as **SAC-Discrete** [Chr19]. The J_π objective can now be computed as

$$J'_\pi(\boldsymbol{\theta}) = \mathbb{E}_{\boldsymbol{s}_t \sim \mathcal{D}} \left[\sum_a \pi_{\boldsymbol{\theta}}(a|\boldsymbol{s}_t) [\alpha \log \pi_{\boldsymbol{\theta}}(a|\boldsymbol{s}_t) - Q_{\boldsymbol{w}}(\boldsymbol{s}_t, a)] \right] \tag{3.103}$$

which avoids the need for reparameterization. (In [Zho+22], they propose to augment J'_π with an entropy penalty, adding a term of the form $\frac{1}{2}(\mathbb{H}_{\text{old}} - \mathbb{H}_\pi)^2$, to prevent drastic changes in the policy, where the entropy of the policy can be computed analytically per sampled state.) The J_Q term is similar to before

$$J'_Q(\boldsymbol{w}) = \mathbb{E}_{(\boldsymbol{s}_t, \boldsymbol{a}_t, r_{t+1}, \boldsymbol{s}_{t+1}) \sim \mathcal{D}} \left[\frac{1}{2} \left(Q_{\boldsymbol{w}}(\boldsymbol{s}_t, \boldsymbol{a}_t) - q'(r_{t+1}, \boldsymbol{s}_{t+1}) \right)^2 \right] \tag{3.104}$$

where now the frozen target function is given by

$$q'(r_{t+1}, \boldsymbol{s}_{t+1}) = r_{t+1} + \gamma \left(\sum_{a_{t+1}} \pi_{\boldsymbol{\theta}}(a_{t+1}|\boldsymbol{s}_{t+1}) [\min_{i=1,2} Q_{\overline{\boldsymbol{w}}_i}(\boldsymbol{s}_{t+1}, a_{t+1}) - \alpha \log \pi_{\boldsymbol{\theta}}(a_{t+1}|\boldsymbol{s}_{t+1})] \right) \tag{3.105}$$

3.5.4.4 Adjusting the temperature

In [Haa+18b] they propose to automatically adjust the temperature parameter α by optimizing

$$J(\alpha) = \mathbb{E}_{\boldsymbol{s}_t \sim \mathcal{D}, \boldsymbol{a}_t \sim \pi_{\boldsymbol{\theta}}} \left[-\alpha (\log \pi_{\boldsymbol{\theta}}(\boldsymbol{a}_t|\boldsymbol{s}_t) + \overline{H}) \right]$$

where $\overline{H}$ is the target entropy (a hyper-parameters). This objective is approximated by sampling actions from the replay buffer.

其中 $\tilde{\boldsymbol{a}}_{t+1} \sim \pi_{\boldsymbol{\theta}}(\boldsymbol{s}_{t+1})$ 是采样的下一个动作。在 [Che+20] 中，他们提出了 REDQ 方法（第 2.5.3.3 节），该方法使用随机网络集合而不是仅仅 2 个。$N \geq 2$

3.5.4.2 政策改进：高斯策略

为了政策改进，我们保持值函数参数 $\boldsymbol{w}$ 固定，并通过最小化以下目标来优化策略参数 $\boldsymbol{\theta}$，该目标通过乘以 α 并去掉常数项 Z 从 KL 项导出：

$$J_\pi(\boldsymbol{\theta}) = \mathbb{E}_{\boldsymbol{s}_t \sim \mathcal{D}} \left[\mathbb{E}_{\boldsymbol{a}_t \sim \pi_{\boldsymbol{\theta}}} \left[\alpha \log \pi_{\boldsymbol{\theta}}(\boldsymbol{a}_t|\boldsymbol{s}_t) - Q_{\boldsymbol{w}}(\boldsymbol{s}_t, \boldsymbol{a}_t) \right] \right] \tag{3.101}$$

由于我们正在对 $\boldsymbol{\theta}$ 求梯度，这会影响内部期望项，因此我们需要使用方程（3.15）中的 REINFORCE 估计器或 **重参数化技巧**（例如，参见 [Moh+20]）。后者方差更低，因此更可取。

为了更详细地解释这一点，让我们假设策略分布具有以下形式 $\pi_{\boldsymbol{\theta}}(\boldsymbol{a}_t|\boldsymbol{s}_t) = \mathcal{N}(\boldsymbol{\mu}_{\boldsymbol{\theta}}(\boldsymbol{s}_t)\, \sigma^2 \mathbf{I})$。我们可以将随机动作写成 $\boldsymbol{a}_t = f_{\boldsymbol{\theta}}(\boldsymbol{s}_t, \boldsymbol{\epsilon}_t)$，其中 f 是状态和噪声变量 $\boldsymbol{\epsilon}_t$ 的确定性函数，因为 $\boldsymbol{a}_t = \boldsymbol{\mu}(\boldsymbol{s}_t) + \sigma^2 \boldsymbol{\epsilon}_t$，其中 $\boldsymbol{\epsilon}_t \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$。现在的目标变成了

$$J_\pi(\boldsymbol{\theta}) = \mathbb{E}_{\boldsymbol{s}_t \sim \mathcal{D}, \boldsymbol{\epsilon}_t \sim \mathcal{N}} \left[\alpha \log \pi_{\boldsymbol{\theta}}(f_{\boldsymbol{\theta}}(\boldsymbol{s}_t, \boldsymbol{\epsilon}_t)|\boldsymbol{s}_t) - Q_{\boldsymbol{w}}(\boldsymbol{s}_t, f_{\boldsymbol{\theta}}(\boldsymbol{s}_t, \boldsymbol{\epsilon}_t)) \right] \tag{3.102}$$

我们将相对于的 $\boldsymbol{a}_t$ 的期望替换为相对于其噪声分布 $\mathcal{N}$ 的期望。因此，我们现在可以安全地取随机梯度。请参阅算法 8 的伪代码。（注意，对于离散动作，我们可以通过显式计算期望来避免重新参数化技巧的需求，如第 3.5.4.3 节 3.5.4.3 中所述。）

3.5.4.3 政策改进：softmax 策略

对于离散动作，我们可以用 gumbel-softmax 重新参数化替换高斯重新参数化 [JGP16； MMT17]。或者，我们可以避免采样，并显式计算动作的期望，以推导出方差更低的方程版本；这被称为 **SAC-Discrete** [Chr19]。现在，J_π 目标可以计算为

$$J'_\pi(\boldsymbol{\theta}) = \mathbb{E}_{\boldsymbol{s}_t \sim \mathcal{D}} \left[\sum_a \pi_{\boldsymbol{\theta}}(a|\boldsymbol{s}_t) [\alpha \log \pi_{\boldsymbol{\theta}}(a|\boldsymbol{s}_t) - Q_{\boldsymbol{w}}(\boldsymbol{s}_t, a)] \right] \tag{3.103}$$

该方案避免了重新参数化的需求。（在 [Zho+22] 中，他们提出通过增加熵惩罚来增强 J'_π，添加一个形式为 $^1{}_2(\mathbb{H}_{\text{old}} - \mathbb{H}_\pi)_2$ 的项，以防止策略发生剧烈变化，其中策略的熵可以针对每个采样状态进行解析计算。） J_Q 项与之前类似

$$J'_Q(\boldsymbol{w}) = \mathbb{E}_{(\boldsymbol{s}_t, \boldsymbol{a}_t, r_{t+1}, \boldsymbol{s}_{t+1}) \sim \mathcal{D}} \left[\frac{1}{2} \left(Q_{\boldsymbol{w}}(\boldsymbol{s}_t, \boldsymbol{a}_t) - q'(r_{t+1}, \boldsymbol{s}_{t+1}) \right)^2 \right] \tag{3.104}$$

现在冻结的目标函数由 给出

$$q'(r_{t+1}, \boldsymbol{s}_{t+1}) = r_{t+1} + \gamma \left(\sum_{a_{t+1}} \pi_{\boldsymbol{\theta}}(a_{t+1}|\boldsymbol{s}_{t+1}) [\min_{i=1,2} Q_{\boldsymbol{w}_i}(\boldsymbol{s}_{t+1}, a_{t+1}) - \alpha \log \pi_{\boldsymbol{\theta}}(a_{t+1}|\boldsymbol{s}_{t+1})] \right) \tag{3.105}$$

3.5.4.4 调整温度

在 [Haa+18b] 他们提出通过优化 α 自动调整温度参数

$$J(\alpha) = \mathbb{E}_{\boldsymbol{s}_t \sim \mathcal{D}, \boldsymbol{a}_t \sim \pi_{\boldsymbol{\theta}}} \left[-\alpha (\log \pi_{\boldsymbol{\theta}}(\boldsymbol{a}_t|\boldsymbol{s}_t) + H) \right]$$

H 是目标熵（一个超参数）。此目标通过从重放缓冲区中采样动作来近似。

Algorithm 8: SAC

Initialize environment state $\boldsymbol{s}$, policy parameters $\boldsymbol{\theta}$, N critic parameters $\boldsymbol{w}_i$, target parameters $\overline{\boldsymbol{w}}_i = \boldsymbol{w}_i$, replay buffer $\mathcal{D} = \emptyset$, discount factor γ, EMA rate ρ, step size η_w, η_π.
repeat
  Take action $\boldsymbol{a} \sim \pi_{\boldsymbol{\theta}}(\cdot|\boldsymbol{s})$
  $(\boldsymbol{s}', r) = \text{step}(\boldsymbol{a}, \boldsymbol{s})$
  $\mathcal{D} := \mathcal{D} \cup \{(\boldsymbol{s}, \boldsymbol{a}, r, \boldsymbol{s}')\}$
  $\boldsymbol{s} \leftarrow \boldsymbol{s}'$
  for G *updates* **do**
    Sample a minibatch $\mathcal{B} = \{(\boldsymbol{s}_j, \boldsymbol{a}_j, r_j, \boldsymbol{s}'_j)\}$ from $\mathcal{D}$
    $\boldsymbol{w} = \text{update-critics}(\boldsymbol{\theta}, \boldsymbol{w}, \mathcal{B})$
  Sample a minibatch $\mathcal{B} = \{(\boldsymbol{s}_j, \boldsymbol{a}_j, r_j, \boldsymbol{s}'_j)\}$ from $\mathcal{D}$
  $\boldsymbol{\theta} = \text{update-policy}(\boldsymbol{\theta}, \boldsymbol{w}, \mathcal{B})$
until *converged*
.
def update-critics($\boldsymbol{\theta}, \boldsymbol{w}, \mathcal{B}$):
Let $(\boldsymbol{s}_j, \boldsymbol{a}_j, r_j, \boldsymbol{s}'_j)_{j=1}^B = \mathcal{B}$
$q_j = q(r_j, \boldsymbol{s}'_j; \overline{\boldsymbol{w}}_{1:N}, \boldsymbol{\theta})$ for $j = 1 : B$
for $i = 1 : N$ **do**
  $\mathcal{L}(\boldsymbol{w}_i) = \frac{1}{|\mathcal{B}|} \sum_{(\boldsymbol{s},\boldsymbol{a},r,\boldsymbol{s}')_j \in \mathcal{B}} (Q_{\boldsymbol{w}_i}(\boldsymbol{s}_j, \boldsymbol{a}_j) - \text{sg}(q_j))^2$
  $\boldsymbol{w}_i \leftarrow \boldsymbol{w}_i - \eta_{\boldsymbol{w}} \nabla \mathcal{L}(\boldsymbol{w}_i)$ // Descent
  $\overline{\boldsymbol{w}}_i := \rho \overline{\boldsymbol{w}}_i + (1 - \rho) \boldsymbol{w}_i$ //Update target networks
Return $\boldsymbol{w}_{1:N}, \overline{\boldsymbol{w}}_{1:N}$
.
def update-actor($\boldsymbol{\theta}, \boldsymbol{w}, \mathcal{B}$):
$\hat{Q}(s, a) \triangleq \frac{1}{N} \sum_{i=1}^N Q_{\boldsymbol{w}_i}(s, a)$ // Average critic
$J(\boldsymbol{\theta}) = \frac{1}{|\mathcal{B}|} \sum_{\boldsymbol{s} \in \mathcal{B}} \left(\hat{Q}(\boldsymbol{s}, \tilde{\boldsymbol{a}}_{\boldsymbol{\theta}}(\boldsymbol{s})) - \alpha \log \pi_{\boldsymbol{\theta}}(\tilde{\boldsymbol{a}}(\boldsymbol{s})|\boldsymbol{s}) \right), \ \tilde{\boldsymbol{a}}_{\boldsymbol{\theta}}(\boldsymbol{s}) \sim \pi_{\boldsymbol{\theta}}(\cdot|\boldsymbol{s})$
$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \eta_{\boldsymbol{\theta}} \nabla J(\boldsymbol{\theta})$ // Ascent
Return $\boldsymbol{\theta}$

算法 8： SAC

Initialize environment state $\boldsymbol{s}$, policy parameters $\boldsymbol{\theta}$, N critic parameters $\boldsymbol{w}_i$, target parameters $\boldsymbol{w}_i = \boldsymbol{w}_i$, replay buffer $\mathcal{D} = \emptyset$, discount factor γ, EMA rate ρ, step size η_w, η_π.
repeat
  Take action $\boldsymbol{a} \sim \pi_{\boldsymbol{\theta}}(\cdot|\boldsymbol{s})$
  $(\boldsymbol{s}', r) = \text{step}(\boldsymbol{a}, \boldsymbol{s})$
  $\mathcal{D} := \mathcal{D} \cup \{(\boldsymbol{s}, \boldsymbol{a}, r, \boldsymbol{s}')\}$
  $\boldsymbol{s} \leftarrow \boldsymbol{s}'$
  for G *updates* **do**
    Sample a minibatch $\mathcal{B} = \{(\boldsymbol{s}_j, \boldsymbol{a}_j, r_j, \boldsymbol{s}'_j)\}$ from $\mathcal{D}$
    $\boldsymbol{w} = \text{update-critics}(\boldsymbol{\theta}, \boldsymbol{w}, \mathcal{B})$
  Sample a minibatch $\mathcal{B} = \{(\boldsymbol{s}_j, \boldsymbol{a}_j, r_j, \boldsymbol{s}'_j)\}$ from $\mathcal{D}$
  $\boldsymbol{\theta} = \text{update-policy}(\boldsymbol{\theta}, \boldsymbol{w}, \mathcal{B})$
until *converged*
.
def update-critics($\boldsymbol{\theta}, \boldsymbol{w}, \mathcal{B}$):
Let $(\boldsymbol{s}_j, \boldsymbol{a}_j, r_j, \boldsymbol{s}'_j)_{j=1}^B = \mathcal{B}$
$q_j = q(r_j, \boldsymbol{s}'_j; \boldsymbol{w}_{1:N}, \boldsymbol{\theta})$ for $j = 1 : B$
for $i = 1 : N$ **do**
  $\mathcal{L}(\boldsymbol{w}_i) = \frac{1}{|\mathcal{B}|} \sum_{(\boldsymbol{s}, \boldsymbol{a}, r, \boldsymbol{s}')_j \in \mathcal{B}} (Q_{\boldsymbol{w}_i}(\boldsymbol{s}_j, \boldsymbol{a}_j) - \text{sg}(q_j))^2$
  $\boldsymbol{w}_i \leftarrow \boldsymbol{w}_i - \eta_{\boldsymbol{w}} \nabla \mathcal{L}(\boldsymbol{w}_i)$ // Descent
  $\boldsymbol{w}_i := \rho \boldsymbol{w}_i + (1 - \rho) \boldsymbol{w}_i$ //Update target networks
Return $\boldsymbol{w}_{1:N}, \boldsymbol{w}_{1:N}$
.
def update-actor($\boldsymbol{\theta}, \boldsymbol{w}, \mathcal{B}$):
$\hat{Q}(s, a) \triangleq \frac{1}{N} \sum_{i=1}^N Q_{\boldsymbol{w}_i}(s, a)$ // Average critic
$J(\boldsymbol{\theta}) = \frac{1}{|\mathcal{B}|} \sum_{\boldsymbol{s} \in \mathcal{B}} \left(\hat{Q}(\boldsymbol{s}, \tilde{\boldsymbol{a}}_{\boldsymbol{\theta}}(\boldsymbol{s})) - \alpha \log \pi_{\boldsymbol{\theta}}(\tilde{\boldsymbol{a}}(\boldsymbol{s})|\boldsymbol{s}) \right), \ \tilde{\boldsymbol{a}}_{\boldsymbol{\theta}}(\boldsymbol{s}) \sim \pi_{\boldsymbol{\theta}}(\cdot|\boldsymbol{s})$
$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \eta_{\boldsymbol{\theta}} \nabla J(\boldsymbol{\theta})$ // Ascent
Return $\boldsymbol{\theta}$

For discrete actions, temperature objective is given by

$$J'(\alpha) = \mathbb{E}_{\boldsymbol{s}_t \sim \mathcal{D}}\left[\sum_a \pi_t(a|\boldsymbol{s}_t)[-\alpha(\log \pi_t(\boldsymbol{a}_t|\boldsymbol{s}_t) + \overline{H})]\right] \tag{3.106}$$

3.6 Deterministic policy gradient methods

In this section, we consider the case of a deterministic policy, that predicts a unique action for each state, so $a_t = \mu_{\boldsymbol{\theta}}(s_t)$, rather than $a_t \sim \pi_{\boldsymbol{\theta}}(s_t)$. (We require that the actions are continuous, because we will take the Jacobian of the Q function wrt the actions; if the actions are discrete, we can just use DQN.) The advantage of using a deterministic policy is that we can modify the policy gradient method so that it can work off policy without needing importance sampling, as we will see.

Following Equation (3.7), we define the value of a policy as the expected discounted reward per state:

$$J(\mu_{\boldsymbol{\theta}}) \triangleq \mathbb{E}_{\rho_{\mu_{\boldsymbol{\theta}}}(s)}\left[R(s, \mu_{\boldsymbol{\theta}}(s))\right] \tag{3.107}$$

The **deterministic policy gradient theorem** [Sil+14] tells us that the gradient of this expression is given by

$$\nabla_{\boldsymbol{\theta}} J(\mu_{\boldsymbol{\theta}}) = \mathbb{E}_{\rho_{\mu_{\boldsymbol{\theta}}}(s)}\left[\nabla_{\boldsymbol{\theta}} Q_{\mu_{\boldsymbol{\theta}}}(s, \mu_{\boldsymbol{\theta}}(s))\right] \tag{3.108}$$

$$= \mathbb{E}_{\rho_{\mu_{\boldsymbol{\theta}}}(s)}\left[\nabla_{\boldsymbol{\theta}} \mu_{\boldsymbol{\theta}}(s) \nabla_a Q_{\mu_{\boldsymbol{\theta}}}(s, a)|_{a=\mu_{\boldsymbol{\theta}}(s)}\right] \tag{3.109}$$

where $\nabla_{\boldsymbol{\theta}} \mu_{\boldsymbol{\theta}}(s)$ is the $M \times N$ Jacobian matrix, and M and N are the dimensions of $\mathcal{A}$ and $\boldsymbol{\theta}$, respectively. For stochastic policies of the form $\pi_{\boldsymbol{\theta}}(a|s) = \mu_{\boldsymbol{\theta}}(s) + \text{noise}$, the standard policy gradient theorem reduces to the above form as the noise level goes to zero.

Note that the gradient estimate in Equation (3.109) integrates over the states but not over the actions, which helps reduce the variance in gradient estimation from sampled trajectories. However, since the deterministic policy does not do any exploration, we need to use an off-policy method for training. This collects data from a stochastic behavior policy π_b, whose stationary state distribution is $p^{\gamma}_{\pi_b}$. The original objective, $J(\mu_{\boldsymbol{\theta}})$, is approximated by the following:

$$J_b(\mu_{\boldsymbol{\theta}}) \triangleq \mathbb{E}_{p^{\gamma}_{\pi_b}(s)}\left[V_{\mu_{\boldsymbol{\theta}}}(s)\right] = \mathbb{E}_{p^{\gamma}_{\pi_b}(s)}\left[Q_{\mu_{\boldsymbol{\theta}}}(s, \mu_{\boldsymbol{\theta}}(s))\right] \tag{3.110}$$

with the off-policy deterministic policy gradient from [DWS12] is approximated by

$$\nabla_{\boldsymbol{\theta}} J_b(\mu_{\boldsymbol{\theta}}) \approx \mathbb{E}_{p^{\gamma}_{\pi_b}(s)}\left[\nabla_{\boldsymbol{\theta}}\left[Q_{\mu_{\boldsymbol{\theta}}}(s, \mu_{\boldsymbol{\theta}}(s))\right]\right] = \mathbb{E}_{p^{\gamma}_{\pi_b}(s)}\left[\nabla_{\boldsymbol{\theta}} \mu_{\boldsymbol{\theta}}(s) \nabla_a Q_{\mu_{\boldsymbol{\theta}}}(s, a)|_{a=\mu_{\boldsymbol{\theta}}(s)}\right] \tag{3.111}$$

where we have a dropped a term that depends on $\nabla_{\boldsymbol{\theta}} Q_{\mu_{\boldsymbol{\theta}}}(s, a)$ and is hard to estimate [Sil+14].

To apply Equation (3.111), we may learn $Q_{\boldsymbol{w}} \approx Q_{\mu_{\boldsymbol{\theta}}}$ with TD, giving rise to the following updates:

$$\delta = r_t + \gamma Q_{\boldsymbol{w}}(s_{t+1}, \mu_{\boldsymbol{\theta}}(s_{t+1})) - Q_{\boldsymbol{w}}(s_t, a_t) \tag{3.112}$$

$$\boldsymbol{w}_{t+1} \leftarrow \boldsymbol{w}_t + \eta_{\boldsymbol{w}} \delta \nabla_{\boldsymbol{w}} Q_{\boldsymbol{w}}(s_t, a_t) \tag{3.113}$$

$$\boldsymbol{\theta}_{t+1} \leftarrow \boldsymbol{\theta}_t + \eta_{\boldsymbol{\theta}} \nabla_{\boldsymbol{\theta}} \mu_{\boldsymbol{\theta}}(s_t) \nabla_a Q_{\boldsymbol{w}}(s_t, a)|_{a=\mu_{\boldsymbol{\theta}}(s_t)} \tag{3.114}$$

So we learn both a state-action critic $Q_{\boldsymbol{w}}$ and an actor $\boldsymbol{\mu}_{\boldsymbol{\theta}}$. This method avoids importance sampling in the actor update because of the deterministic policy gradient, and we avoids it in the critic update because of the use of Q-learning.

If $Q_{\boldsymbol{w}}$ is linear in $\boldsymbol{w}$, and uses features of the form $\boldsymbol{\phi}(s, a) = \boldsymbol{a}^{\mathsf{T}} \nabla_{\boldsymbol{\theta}} \mu_{\boldsymbol{\theta}}(s)$, then we say the function approximator for the critic is **compatible** with the actor; in this case, one can show that the above approximation does not bias the overall gradient.

The basic off-policy DPG method has been extended in various ways, some of which we describe below.

对于离散动作，温度目标由 α 给出

$$J'(\alpha) = \mathbb{E}_{\boldsymbol{s}_t \sim \mathcal{D}}\left[\sum_a \pi_t(a|\boldsymbol{s}_t)[-\alpha(\log \pi_t(\boldsymbol{a}_t|\boldsymbol{s}_t) + H)]\right] \tag{3.106}$$

3.6 确定性策略梯度方法

在这个部分，我们考虑确定性策略的情况，该策略为每个状态预测一个唯一动作，即 $a_t = \mu_{\boldsymbol{\theta}}(s_t)$，而不是 $a_t \sim \pi_{\boldsymbol{\theta}}(s_t)$。（我们要求动作是连续的，因为我们将对 Q 函数相对于动作的雅可比矩阵进行求导；如果动作是离散的，我们就可以直接使用 DQN。）使用确定性策略的优势在于，我们可以修改策略梯度方法，使其能够在离线策略下工作，无需重要性采样，正如我们将看到的。

根据方程（3.7），我们定义策略的值为每个状态期望的折现奖励：

$$J(\mu_{\boldsymbol{\theta}}) \triangleq \mathbb{E}_{\rho_{\mu_{\boldsymbol{\theta}}}(s)}\left[R(s, \mu_{\boldsymbol{\theta}}(s))\right] \tag{3.107}$$

确定性策略梯度定理 [Sil+14] 告诉我们，该表达式的梯度由以下给出

$$\nabla_{\boldsymbol{\theta}} J(\mu_{\boldsymbol{\theta}}) = \mathbb{E}_{\rho_{\mu_{\boldsymbol{\theta}}}(s)}\left[\nabla_{\boldsymbol{\theta}} Q_{\mu_{\boldsymbol{\theta}}}(s, \mu_{\boldsymbol{\theta}}(s))\right] \tag{3.108}$$

$$= \mathbb{E}_{\rho_{\mu_{\boldsymbol{\theta}}}(s)}\left[\nabla_{\boldsymbol{\theta}} \mu_{\boldsymbol{\theta}}(s) \nabla_a Q_{\mu_{\boldsymbol{\theta}}}(s, a)|_{a=\mu_{\boldsymbol{\theta}}(s)}\right] \tag{3.109}$$

$\nabla_{\boldsymbol{\theta}} \mu_{\boldsymbol{\theta}}(s)$ 是 $M \times N$ 雅可比矩阵，而 M 和 N 分别是 $\mathcal{A}$ 和 $\boldsymbol{\theta}$ 的维度。对于形式为 $\pi_{\boldsymbol{\theta}}(a|s) = \mu_{\boldsymbol{\theta}}(s) +$ 噪声的随机策略，当噪声水平趋近于零时，标准策略梯度定理简化为上述形式。

请注意，方程（3.109）中的梯度估计是在状态上而不是在动作上积分，这有助于减少从采样轨迹中估计梯度的方差。然而，由于确定性策略不进行任何探索，我们需要使用离线策略进行训练。这从具有平稳状态分布的随机行为策略 π_b 收集数据。原始目标 $J(\mu_{\boldsymbol{\theta}})$ 被以下近似：

$$J_b(\mu_{\boldsymbol{\theta}}) \triangleq \mathbb{E}_{p^{\gamma}_{\pi_b}(s)}\left[V_{\mu_{\boldsymbol{\theta}}}(s)\right] = \mathbb{E}_{p^{\gamma}_{\pi_b}(s)}\left[Q_{\mu_{\boldsymbol{\theta}}}(s, \mu_{\boldsymbol{\theta}}(s))\right] \tag{3.110}$$

与 [DWS12] 的离策略确定性策略梯度被近似

$$\nabla_{\boldsymbol{\theta}} J_b(\mu_{\boldsymbol{\theta}}) \approx \mathbb{E}_{p^{\gamma}_{\pi_b}(s)}\left[\nabla_{\boldsymbol{\theta}}\left[Q_{\mu_{\boldsymbol{\theta}}}(s, \mu_{\boldsymbol{\theta}}(s))\right]\right] = \mathbb{E}_{p^{\gamma}_{\pi_b}(s)}\left[\nabla_{\boldsymbol{\theta}} \mu_{\boldsymbol{\theta}}(s) \nabla_a Q_{\mu_{\boldsymbol{\theta}}}(s, a)|_{a=\mu_{\boldsymbol{\theta}}(s)}\right] \tag{3.111}$$

我们在其中省略了一个依赖于 $\nabla_{\boldsymbol{\theta}} Q_{\mu_{\boldsymbol{\theta}}}(s, a)$ 且难以估计 [Sil+14].

应用方程（3.111），我们可以通过 TD 学习 $Q_{\boldsymbol{w}} \approx Q_{\mu_{\boldsymbol{\theta}}}$，从而产生以下更新：

$$\delta = r_t + \gamma Q_{\boldsymbol{w}}(s_{t+1}, \mu_{\boldsymbol{\theta}}(s_{t+1})) - Q_{\boldsymbol{w}}(s_t, a_t) \tag{3.112}$$

$$\boldsymbol{w}_{t+1} \leftarrow \boldsymbol{w}_t + \eta_{\boldsymbol{w}} \delta \nabla_{\boldsymbol{w}} Q_{\boldsymbol{w}}(s_t, a_t) \tag{3.113}$$

$$\boldsymbol{\theta}_{t+1} \leftarrow \boldsymbol{\theta}_t + \eta_{\boldsymbol{\theta}} \nabla_{\boldsymbol{\theta}} \mu_{\boldsymbol{\theta}}(s_t) \nabla_a Q_{\boldsymbol{w}}(s_t, a)|_{a=\mu_{\boldsymbol{\theta}}(s_t)} \tag{3.114}$$

因此，我们学习了一个状态 - 动作评论家 $Q_{\boldsymbol{w}}$ 和一个演员 $\boldsymbol{\mu}_{\boldsymbol{\theta}}$。这种方法由于确定性的策略梯度而避免了在演员更新中的重要性采样，并且由于使用了 Q 学习，我们在评论家更新中也避免了它。

如果 $Q_{\boldsymbol{w}}$ 相对于 $\boldsymbol{w}$ 是线性的，并且使用形式为 $\phi(s,a) = \boldsymbol{a}^{\mathsf{T}} \nabla_{\boldsymbol{\theta}} \mu_{\boldsymbol{\theta}}(s)$ 的特征，那么我们说对于批评家的函数逼近器是**与演员**兼容的；在这种情况下，可以证明上述逼近不会偏置整体梯度。基本的离策略 DPG 方法已被以各种方式扩展，其中一些以下我们将描述。

3.6.1 DDPG

The **DDPG** algorithm of [Lil+16], which stands for "deep deterministic policy gradient", uses the DQN method (Section 2.5.2.2) to update Q that is represented by deep neural networks. In more detail, the actor tries to minimize the output of the critic by optimize

$$\mathcal{L}_{\boldsymbol{\theta}}(s) = Q_{\boldsymbol{w}}(s, \mu_{\boldsymbol{\theta}}(s)) \tag{3.115}$$

averaged over states s drawn from the replay buffer. The critic tries to minimize the 1-step TD loss

$$\mathcal{L}_{\boldsymbol{w}}(s, a, r, s') = [Q_{\boldsymbol{w}}(s, a) - (r + \gamma Q_{\overline{\boldsymbol{w}}}(s', \mu_{\boldsymbol{\theta}}(s')))]^2 \tag{3.116}$$

where $Q_{\overline{\boldsymbol{w}}}$ is the target critic network, and the samples (s, a, r, a') are drawn from a replay buffer. (See Section 2.5.2.5 for a discussion of target networks.)

The **D4PG** algorithm [BM+18], which stands for "distributed distributional DDPG", extends DDPG to handle distributed training, and to handle distributional RL (see Section 5.1).

3.6.2 Twin Delayed DDPG (TD3)

The **TD3** ("twin delayed deep deterministic") method of [FHM18] extends DDPG in 3 main ways. First, it uses **target policy smoothing**, in which noise is added to the action, to encourage generalization:

$$\tilde{\boldsymbol{a}} = \boldsymbol{\mu}_{\boldsymbol{\theta}}(\boldsymbol{s}) + \text{noise} = \pi_{\boldsymbol{\theta}}(\boldsymbol{s}) \tag{3.117}$$

Second it uses **clipped double Q learning**, which is an extension of the double Q-learning discussed in Section 2.5.3.1 to avoid over-estimation bias. In particular, the target values for TD learning are defined using

$$q(r, \boldsymbol{s}'; \overline{\boldsymbol{w}}_{1:2}, \overline{\boldsymbol{\theta}}) = r + \gamma \min_{i=1,2} Q_{\overline{\boldsymbol{w}}_i}(\boldsymbol{s}', \pi_{\overline{\boldsymbol{\theta}}}(\boldsymbol{s}')) \tag{3.118}$$

Third, it uses **delayed policy updates**, in which it only updates the policy after the value function has stabilized. (See also Section 3.3.3.) See Algorithm 9 for the pseudcode.

Lil 的 **DDPG** 算法，代表 “ 深度确定性策略梯度 ”，使用 DQN 方法（第 2.5.2.2 节）来更新由深度神经网络表示的内容。更详细地说，演员试图通过优化来最小化评论家的输出。

$$\mathcal{L}_{\boldsymbol{\theta}}(s) = Q_{\boldsymbol{w}}(s, \mu_{\boldsymbol{\theta}}(s)) \tag{3.115}$$

averaged over states s draw从重放缓冲区中提取 n。评论家试图最小化 1-step TD loss

$$\mathcal{L}_{\boldsymbol{w}}(s, a, r, s') = [Q_{\boldsymbol{w}}(s, a) - (r + \gamma Q_{\boldsymbol{w}}(s', \mu_{\boldsymbol{\theta}}(s')))]^2 \tag{3.116}$$

其中 $Q_{\boldsymbol{w}}$ 是目标批评网络，样本 (s, a,r, a') 是从重放缓冲区中抽取的。（有关目标网络的讨论，请参阅第 2.5.2.5 节。）

分布式分布式 DDPG 算法，**代表 “ 分布式分布式 DDPG”，扩展了 DDPG 以处理分布式训练和处理分布式强化学习（见第 5.1 节）**。[BM+18]

3.6.2 双延迟深度确定性策略梯度（ TD3 ）

TD3 （ “ 双延迟深度确定性 ” ）方法在 [FHM18] 中扩展了 DDPG 的 3 个主要方面。首先，它使用 **目标策略平滑**，即在动作中添加噪声，以鼓励泛化：

$$\tilde{\boldsymbol{a}} = \boldsymbol{\mu}_{\boldsymbol{\theta}}(\boldsymbol{s}) + \text{noise} = \pi_{\boldsymbol{\theta}}(\boldsymbol{s}) \tag{3.117}$$

第二次，它使用了**剪枝双 Q 学习**，这是对第 2.5.3.1 节中讨论的双 Q 学习的扩展。为了避免过估计偏差。特别是，TD 学习的目标值使用

$$q(r, \boldsymbol{s}'; \boldsymbol{w}_{1:2}, \boldsymbol{\theta}) = r + \gamma \min_{i=1,2} Q_{\boldsymbol{w}_i}(\boldsymbol{s}', \pi_{\boldsymbol{\theta}}(\boldsymbol{s}')) \tag{3.118}$$

第三，它使用**延迟策略更新**，其中它仅在值函数稳定后更新策略。（另见第 3.3.3 节。）有关伪代码，请参阅算法 9 。

Algorithm 9: TD3

Initialize environment state $\boldsymbol{s}$, policy parameters $\boldsymbol{\theta}$, target policy parameters $\overline{\boldsymbol{\theta}}$, critic parameters $\boldsymbol{w}_i$, target critic parameters $\overline{\boldsymbol{w}}_i = \boldsymbol{w}_i$, replay buffer $\mathcal{D} = \emptyset$, discount factor γ, EMA rate ρ, step size $\eta_{\boldsymbol{w}}$, $\eta_{\boldsymbol{\theta}}$.
repeat
  $\boldsymbol{a} = \mu_{\boldsymbol{\theta}}(\boldsymbol{s}) + \text{noise}$
  $(\boldsymbol{s}', r) = \text{step}(\boldsymbol{a}, \boldsymbol{s})$
  $\mathcal{D} := \mathcal{D} \cup \{(\boldsymbol{s}, \boldsymbol{a}, r, \boldsymbol{s}')\}$
  $\boldsymbol{s} \leftarrow \boldsymbol{s}'$
  for G *updates* **do**
    Sample a minibatch $\mathcal{B} = \{(\boldsymbol{s}_j, \boldsymbol{a}_j, r_j, \boldsymbol{s}'_j)\}$ from $\mathcal{D}$
    $\boldsymbol{w} = \text{update-critics}(\boldsymbol{\theta}, \boldsymbol{w}, \mathcal{B})$
  Sample a minibatch $\mathcal{B} = \{(\boldsymbol{s}_j, \boldsymbol{a}_j, r_j, \boldsymbol{s}'_j)\}$ from $\mathcal{D}$
  $\boldsymbol{\theta} = \text{update-policy}(\boldsymbol{\theta}, \boldsymbol{w}, \mathcal{B})$
until *converged*
.
def update-critics$(\boldsymbol{\theta}, \boldsymbol{w}, \mathcal{B})$:
Let $(\boldsymbol{s}_j, \boldsymbol{a}_j, r_j, \boldsymbol{s}'_j)_{j=1}^{B} = \mathcal{B}$
for $j = 1 : B$ **do**
  $\tilde{\boldsymbol{a}}_j = \mu_{\overline{\boldsymbol{\theta}}}(\boldsymbol{s}'_j) + \text{clip}(\text{noise}, -c, c)$
  $q_j = r_j + \gamma \min_{i=1,2} Q_{\overline{\boldsymbol{w}}_i}(\boldsymbol{s}'_j, \tilde{\boldsymbol{a}}_j)$
for $i = 1 : 2$ **do**
  $\mathcal{L}(\boldsymbol{w}_i) = \frac{1}{|\mathcal{B}|} \sum_{(\boldsymbol{s}, \boldsymbol{a}, r, \boldsymbol{s}')_j \in \mathcal{B}} (Q_{\boldsymbol{w}_i}(\boldsymbol{s}_j, \boldsymbol{a}_j) - \text{sg}(q_j))^2$
  $\boldsymbol{w}_i \leftarrow \boldsymbol{w}_i - \eta_{\boldsymbol{w}} \nabla \mathcal{L}(\boldsymbol{w}_i)$ // Descent
  $\overline{\boldsymbol{w}}_i := \rho \overline{\boldsymbol{w}}_i + (1 - \rho) \boldsymbol{w}_i$ //Update target networks with EMA
Return $\boldsymbol{w}_{1:N}, \overline{\boldsymbol{w}}_{1:N}$
.
def update-actor$(\boldsymbol{\theta}, \boldsymbol{w}, \mathcal{B})$:
$J(\boldsymbol{\theta}) = \frac{1}{|\mathcal{B}|} \sum_{\boldsymbol{s} \in \mathcal{B}} \left(Q_{\boldsymbol{w}_1}(\boldsymbol{s}, \mu_{\boldsymbol{\theta}}(\boldsymbol{s}))\right)^2$
$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \eta_{\boldsymbol{\theta}} \nabla J(\boldsymbol{\theta})$ // Ascent
$\overline{\boldsymbol{\theta}} := \rho \overline{\boldsymbol{\theta}} + (1 - \rho) \boldsymbol{\theta}$ //Update target policy network with EMA
Return $\boldsymbol{\theta}, \overline{\boldsymbol{\theta}}$

算法 9： TD3

初始化环境状态 $\boldsymbol{s}$，策略参数 $\boldsymbol{\theta}$，目标策略参数 $\boldsymbol{\theta}$，评论家参数 $\boldsymbol{w}_i$，目标评论家参数 $\boldsymbol{w}_i = \boldsymbol{w}_i$，重放缓冲区 $\mathcal{D} = \emptyset$，折扣因子 γ，EMA 率 ρ，步长 $\eta_{\boldsymbol{w}}$，$\eta_{\boldsymbol{\theta}}$ **重复** $\boldsymbol{a} = \mu_{\boldsymbol{\theta}}(\boldsymbol{s}) +$ 噪声 $(\boldsymbol{s}', r) =$ 步 $(\boldsymbol{a}, \boldsymbol{s})$ $\mathcal{D} := \mathcal{D} \cup \{(\boldsymbol{s}, \boldsymbol{a}, r, \boldsymbol{s}')\}$ $\boldsymbol{s} \leftarrow \boldsymbol{s}'$ **对于** G 更新 **做** 从 $\mathcal{B} = \{(\boldsymbol{s}_j, \boldsymbol{a}_j, r_j, \boldsymbol{s}'_j)\}$ 中采样一个迷你批次 $\mathcal{D}$ $\boldsymbol{w} =$ 更新评论家 $(\boldsymbol{\theta}, \boldsymbol{w}, \mathcal{B})$ 从 $\mathcal{B} = \{(\boldsymbol{s}_j, \boldsymbol{a}_j, r_j, \boldsymbol{s}'_j)\}$ 中采样一个迷你批次 $\mathcal{D}$ $\boldsymbol{\theta} =$ 更新策略 $(\boldsymbol{\theta}, \boldsymbol{w}, \mathcal{B})$ **直到**收敛 . 定义更新评论家 $(\boldsymbol{\theta}, \boldsymbol{w}, \mathcal{B})$: 让 $(\boldsymbol{s}_j, \boldsymbol{a}_j, r_j, \boldsymbol{s}'_j)_{j=1}^{B} = \mathcal{B}$ **对于** $j = 1 : B$ **做** $\tilde{\boldsymbol{a}}_j = \mu_{\boldsymbol{\theta}}(\boldsymbol{s}'_j) + \text{clip}(\text{noise} - c, c)$ $q_j = r_j + \gamma \min_{i=1,2} Q_{\boldsymbol{w}_i}(\boldsymbol{s}'_j, \tilde{\boldsymbol{a}}_j)$ **对于** $i = 1 : 2$ **做** $\mathcal{L}(\boldsymbol{w}_i) = \frac{1}{|\mathcal{B}|} \sum_{(\boldsymbol{s}, \boldsymbol{a}, r, \boldsymbol{s}')_j \in \mathcal{B}} (Q_{\boldsymbol{w}_i}(\boldsymbol{s}_j, \boldsymbol{a}_j) - \text{sg}(q_j))^2$ $\boldsymbol{w}_i \leftarrow \boldsymbol{w}_i - \eta_{\boldsymbol{w}} \nabla \mathcal{L}(\boldsymbol{w}_i)$ // 下降 $\boldsymbol{w}_i := \rho \boldsymbol{w}_i + (1 - \rho) \boldsymbol{w}_i$ // 使用 EMA 更新目标网络 返回 $\boldsymbol{w}_{1:N}, \boldsymbol{w}_{1:N}$. 定义更新演员 $(\boldsymbol{\theta}, \boldsymbol{w}, \mathcal{B})$: $J(\boldsymbol{\theta}) = \frac{1}{|\mathcal{B}|} \sum_{\boldsymbol{s} \in \mathcal{B}} (Q_{\boldsymbol{w}_1}(\boldsymbol{s}, \mu_{\boldsymbol{\theta}}(\boldsymbol{s})))^2$ $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \eta_{\boldsymbol{\theta}} \nabla J(\boldsymbol{\theta})$ // 上升 $\boldsymbol{\theta} := \rho \boldsymbol{\theta} + (1 - \rho) \boldsymbol{\theta}$ //使用 EMA 更新目标策略网络 返回 $\boldsymbol{\theta}, \boldsymbol{\theta}$

Chapter 4

Model-based RL

Model-free approaches to RL typically need a lot of interactions with the environment to achieve good performance. For example, state of the art methods for the Atari benchmark, such as rainbow (Section 2.5.2.2), use millions of frames, equivalent to many days of playing at the standard frame rate. By contrast, humans can achieve the same performance in minutes [Tsi+17]. Similarly, OpenAI's robot hand controller [And+20] needs 100 years of simulated data to learn to manipulate a rubiks cube.

One promising approach to greater sample efficiency is **model-based RL** (**MBRL**). In the simplest approach to MBRL, we first learn the state transition or dynamics model $p_S(s'|s,a)$ — also called a **world model** — and the reward function $R(s,a)$, using some offline trajectory data, and then we use these models to compute a policy (e.g., using dynamic programming, as discussed in Section 2.2, or using some model-free policy learning method on simulated data, as discussed in Chapter 3). It can be shown that the sample complexity of learning the dynamics is less than the sample complexity of learning the policy [ZHR24].

However, the above two-stage approach — where we first learn the model, and then plan with it — can suffer from the usual problems encountered in offline RL (Section 5.5), i.e., the policy may query the model at a state for which no data has been collected, so predictions can be unreliable, causing the policy to learn the wrong thing. To get better results, we have to interleave the model learning and policy learning, so that one helps the other (since the policy determines what data is collected).

There are two main ways to perform MBRL. In the first approach, known as **decision-time planning** or **model predictive control**, we use the model to choose the next action by searching over possible future trajectories. We then score each trajectory, pick the action corresponding to the best one, take a step in the environment, and repeat. (We can also optionally update the model based on the rollouts.) This is discussed in Section 4.1.

The second approach is to use the current model and policy to rollout imaginary trajectories, and to use this data (optionally in addition to empirical data) to improve the policy using model-free RL; this is called **background planning**, and is discussed in Section 4.2.

The advantage of decision-time planning is that it allows us to train a world model on reward-free data, and then use that model to optimize any reward function. This can be particularly useful if the reward contains changing constraints, or if it is an intrinsic reward (Section 5.2.4) that frequently changes based on the knowledge state of the agent. The downside of decision-time planning is that it is much slower. However, it is possible to combine the two methods, as we discuss below. For an empirical comparison of background planning and decision-time planning, see [AP24].

Some generic pseudo-code for an MBRL agent is given in Algorithm 10. (The `rollout` function is defined in Algorithm 11; some simple code for model learning is shown in Algorithm 12, although we discuss other loss functions in Section 4.3; finally, the code for the policy learning is given in other parts of this manuscript.) For more details on general MBRL, see e.g., [Wan+19; Moe+23; PKP21].

第四章

基于模型的强化学习

基于模型的强化学习通常需要与环境进行大量交互才能达到良好的性能。例如，用于 Atari 基准测试的最先进方法，如 rainbow （第 2.5.2.2 节），需要数百万帧画面，相当于在标准帧率下玩好几天。相比之下，人类可以在几分钟内达到同样的性能 [Tsi+17]。同样，OpenAI 的机器人手控制器 [And+20] 需要 100 年的模拟数据来学习操作魔方。

提高样本效率的一个有希望的方法是 **基于模型的强化学习** (**MBRL**)。在 MBRL 的最简单方法中，我们首先学习状态转移或动力学模型 $p_S(s'|s,a)$ —— 也称为 **世界模型** —— 以及奖励函数 $R(s,a)$，使用一些离线轨迹数据，然后我们使用这些模型来计算策略（例如，使用动态规划，如第 2.2 节所述，或使用一些无模型策略学习方法在模拟数据上，如第 3 章所述）。可以证明，学习动力学样本的复杂度小于学习策略样本的复杂度 [ZHR24]。

然而，上述两阶段方法 —— 我们首先学习模型，然后使用它进行规划 —— 可能会遇到在线 RL （第 5.5 节）中常见的通常问题，即策略可能在没有收集到数据的状态下查询模型，因此预测可能不可靠，导致策略学习错误。为了获得更好的结果，我们必须交错进行模型学习和策略学习，以便一个帮助另一个（因为策略决定了收集哪些数据）。

执行 MBRL 主要有两种方法。在第一种方法中，被称为**决策时规划**或**模型预测控制**，我们通过搜索可能的未来轨迹来使用模型选择下一个动作。然后我们为每条轨迹评分，选择对应最佳轨迹的动作，在环境中迈出一步，并重复。（我们还可以根据模拟结果选择性地更新模型。）这将在第 4.1 节中讨论。

第二种方法是使用当前模型和政策来生成虚拟轨迹，并使用这些数据（可选地，加上经验数据）来改进政策，使用无模型强化学习；这被称为**背景规划**，在 4.2 节中讨论。

决策时规划的优势在于，它允许我们在无奖励数据上训练一个世界模型，然后使用该模型来优化任何奖励函数。如果奖励包含变化的约束，或者它是一个基于代理知识状态频繁变化的内在奖励（第 5.2.4 节），这尤其有用。决策时规划的缺点是它要慢得多。然而，我们可以结合这两种方法，如下所述。关于背景规划和决策时规划的实证比较，请参阅 AP。

一些关于 MBRL 代理的通用伪代码在算法 10 中给出。（`rollout` 函数在算法 11 中定义；在算法 12 中展示了用于模型学习的一些简单代码，尽管我们在第 4.3 节讨论了其他损失函数；最后，策略学习的代码在本文的其他部分给出。）有关通用 MBRL 的更多详细信息，请参阅例如 [Wan+19； Moe+23； PKP21]。

Algorithm 10: MBRL agent

def MBRL-agent($M_{\text{env}}; T, H, N$):
Initialize state $s \sim M_{\text{env}}$
Initialize data buffer $\mathcal{D} = \emptyset$, model $\hat{M}$
Initialize value function V, policy proposal π
repeat
  // Collect data from environment
  $\tau_{\text{env}} = \text{rollout}(s, \pi, T, M_{\text{env}})$,
  $s = \tau_{\text{env}}[-1]$,
  $\mathcal{D} = \mathcal{D} \cup \tau_{\text{env}}$
  // Update model
  if *Update model online* **then**
    $\hat{M} = \text{update-model}(\hat{M}, \tau_{\text{env}})$
  if *Update model using replay* **then**
    $\tau^n_{\text{replay}} = \text{sample-trajectory}(\mathcal{D}), n = 1 : N$
    $\hat{M} = \text{update-model}(\hat{M}, \tau^{1:N}_{\text{replay}})$
  // Update policy
  if *Update on-policy with real* **then**
    $(\pi, V) = \text{update-on-policy}(\pi, V, \tau_{\text{env}})$
  if *Update on-policy with imagination* **then**
    $\tau^n_{\text{imag}} = \text{rollout}(\text{sample-init-state}(\mathcal{D}), \pi, T, \hat{M}), n = 1 : N$
    $(\pi, V) = \text{update-on-policy}(\pi, V, \tau^{1:N}_{\text{imag}})$
  if *Update off-policy with real* **then**
    $\tau^n_{\text{replay}} = \text{sample-trajectory}(\mathcal{D}), n = 1 : N$
    $(\pi, V) = \text{update-off-policy}(\pi, V, \tau^{1:N}_{\text{replay}})$
  if *Update off-policy with imagination* **then**
    $\tau^n_{\text{imag}} = \text{rollout}(\text{sample-state}(\mathcal{D}), \pi, T, \hat{M}), n = 1 : N$
    $(\pi, V) = \text{update-off-policy}(\pi, V, \tau^{1:N}_{\text{imag}})$
until *until converged*

Algorithm 11: Rollout

def rollout(s_1, π, T, M)
$\tau = [s_1]$
for $t = 1 : T$ **do**
  $a_t = \pi(s_t)$
  $(s_{t+1}, r_{t+1}) \sim M(s_t, a_t)$
  $\tau += [a_t, r_{t+1}, s_{t+1}]$
Return τ

Algorithm 12: Model learning

def update-model($M, \tau^{1:N}$) :
$\ell(M) = -\frac{1}{NT} \sum_{n=1}^{N} \sum_{(s_t, a_t, r_{t+1}, s_{t+1}) \in \tau^n} \log M(s_{t+1}, r_{t+1} | s_t, a_t)$ // NLL
$M = M - \eta_M \nabla_M \ell(M)$
Return M

算法 10： MBRL 代理

1 定义MBRL-agent(M_{env}; T, H, N):2 初始化状态 $s \sim M_{\text{env}}$3 初始化数据缓冲区 $\mathcal{D} = \emptyset$, 模型 $\hat{M}$4 初始化价值函数 V, 政策提议 π5 **重复** 6 // 从环境中收集数据 7 $\tau_{\text{env}} = \text{rollout}(s, \pi, T, M_{\text{env}})$, 8 $s = \tau_{\text{env}}[-1]$,9 $\mathcal{D} = \mathcal{D} \cup \tau_{\text{env}}$10 // 更新模型 11 **如果** 在线更新模型 **那么** 12 $\hat{M} = \text{update-model}(\hat{M}, \tau_{\text{env}})$13 **如果** 使用重放更新模型 **那么** 14 $\tau^n_{\text{replay}} = \text{sample-trajectory}(\mathcal{D})\ n = 1 : N$15 $\hat{M} = \text{update-model}(\hat{M}, \tau^{1:N}_{\text{replay}})$16 // 更新策略 17 **如果** 使用真实数据在线更新策略 **那么** 18 $(\pi, V) = \text{update-on-policy}(\pi, V, \tau_{\text{env}})$19 **如果** 使用想象数据在线更新策略 **那么** 20 $\tau^n_{\text{imag}} = \text{rollout}(\text{sample-init-state}(\mathcal{D}), \pi, T, \hat{M})\ n = 1 : N$21 $(\pi, V) = \text{update-on-policy}(\pi, V, \tau^{1:N}_{\text{imag}})$22 **如果** 使用真实数据离线更新策略 **那么** 23 $\tau^n_{\text{replay}} = \text{sample-trajectory}(\mathcal{D})\ n = 1 : N$24 $(\pi, V) = \text{update-off-policy}(\pi, V, \tau^{1:N}_{\text{replay}})$25 **如果** 使用想象数据离线更新策略 **那么** 26 $\tau^n_{\text{imag}} = \text{rollout}(\text{sample-state}(\mathcal{D}), \pi, T, \hat{M})\ n = 1 : N$27 $(\pi, V) = \text{update-off-policy}(\pi, V, \tau^{1:N}_{\text{imag}})$

28 **直到**直到收敛

算法 11： 滚动

1 定义函数 rollout($s_1, \pi, T,$ $\mathcal{D}$)6 $(r_{t+1}, s_{t+1}) \sim M(s_t, a_t)$ 3 **for** $t = 1(s_t)$ 4 $= [s_t]$ $a_t = \pi$ 7 返回 τ $[a_t, r_{t+1}, s_{t+1}]$

算法 12： 模型学习

1 def update-model($M, \tau^{1:N}$) :
2 $\ell(M) = -\frac{1}{NT} \sum_{n=1}^{N} \sum_{(s_t, a_t, r_{t+1}, s_{t+1}) \in \tau^n} \log M(s_{t+1}, r_{t+1} | s_t, a_t)$ // NLL
3 $M = M - \eta_M \nabla_M \ell(M)$
4 Return M

4.1 Decision-time planning

If the model is known, and the state and action space is discrete and low dimensional, we can use exact techniques based on dynamic programming to compute the policy, as discussed in Section 2.2. However, for the general case, approximate methods must be used for planning, whether the model is known (e.g., for board games like Chess and Go) or learned.

One approach to approximate planning is to be lazy, and just wait until we know what state we are in, call it s_t, and then decide what to do, rather than trying to learn a policy that maps any state to the best action. This is called **decision time planning** or "**planning in the now**" [KLP11]. We discuss some variants of this approach below.

4.1.1 Model predictive control (MPC)

We now describe a method known as **receeding horizon control** or **model predictive control** (**MPC**) [MM90; CA13; RMD22]: We use the world model to predict future states and rewards that might follow from the current state for each possible sequence of future actions we might pursue, and we then take the action from the sequence that looks most promising. More precisely, at each step, we compute

$$\boldsymbol{a}^*_{t:t+H-1} = \text{planning}(s_t, M, R, \hat{V}, H) \tag{4.1}$$

$$= \underset{\boldsymbol{a}_{t:t+H-1}}{\text{argmax}}\, \mathbb{E}_{s_{t+1:t+H} \sim M(\cdot|s_t, a_{t:t+H-1})} \left[\sum_{h=0}^{H-1} R(s_{t+h}, a_{t+h}) + \hat{V}(s_{t+H}) \right] \tag{4.2}$$

$$\pi_{\text{MPC}}(s_t) = a^*_t \tag{4.3}$$

Here, H is called the **planning horizon**, and $\hat{V}(s_{t+H})$ is an estimate of the reward-to-go at the end of this H-step look-ahead process. We can often speed up the optimization process by using a pre-trained proposal policy $a_t = \pi(s_t)$, which can be used to guide the search process, as we discuss below.

Note that MPC computes a fixed sequence of actions, $\boldsymbol{a}_{t:t+H-1}$, also called a plan, given the current state $\boldsymbol{s}_t$; since the future actions $\boldsymbol{a}_{t'}$ for $t' > t$ are independent of the future states $\boldsymbol{s}_{t'}$, this is called an **open loop controller**. Such a controller can work well in deterministic environments (where $\boldsymbol{s}_{t'}$ can be computed from $\boldsymbol{s}_t$ and the action sequence), but in general, we will need to replan at each step, as the actual next state is observed. Thus MPC is a way of creating a **closed loop controller**.

We can combine MPC with model and policy/proposal learning using the pseudocode in Algorithm 10, where the decision policy $a_t = \pi_{\text{MPC}}(s_t)$ is implemented by Equation (4.2). If we want to learn the propsoal policy $a_t = \pi(s_t)$, we should use off-policy methods, since the training data (even if imaginary) will be collected by π_{MPC} rather than by π. When learning the world model, we only need it to be locally accurate, around the current state, which means we can often use simpler models in MPC than in background planning approaches.

In the sections below, we discuss particular kinds of MPC methods. Further connections between MPC and RL are discussed in [Ber24].

4.1.2 Heuristic search

If the state and action spaces are finite, we can solve Equation (4.2) exactly, although the time complexity will typically be exponential in H. However, in many situations, we can prune off unpromising trajectories, thus making the approach feasible in large scale problems.

In particular, consider a discrete, deterministic MDP where reward maximization corresponds to finding a shortest path to a goal state. We can expand the successors of the current state according to all possible actions, trying to find the goal state. Since the search tree grows exponentially with depth, we can use a **heuristic function** to prioritize which nodes to expand; this is called **best-first search**, as illustrated in Figure 4.1.

If the heuristic function is an optimistic lower bound on the true distance to the goal, it is called **admissible**. If we aim to maximize total rewards, admissibility means the heuristic function is an upper

4.1 决策时间规划

如果模型已知，并且状态和动作空间是离散且低维的，我们可以使用基于动态规划的精确技术来计算策略，如第 2.2 节所述。然而，对于一般情况，无论模型是否已知（例如，对于像国际象棋和围棋这样的棋类游戏）或已学习，都必须使用近似方法进行规划。

一种近似规划的方法是懒惰一点，等待我们知道我们处于什么状态，称之为 s_t，然后决定做什么，而不是试图学习一个将任何状态映射到最佳动作的策略。这被称为**决策时间规划**或“**现在规划**”[KLP11]。我们下面讨论这种方法的几种变体。

4.1.1 模型预测控制（MPC）

我们现在描述一种称为**递减视距控制**或**模型预测控制**（**MPC**）[MM90；CA13；RMD22]：我们使用世界模型来预测从当前状态可能产生的未来状态和奖励，对于我们可能追求的每个可能的未来动作序列，然后从看起来最有希望的序列中采取行动。更准确地说，在每一步，我们计算

$$\boldsymbol{a}^*_{t:t+H-1} = \text{planning}(s_t, M, R, \hat{V}, H) \tag{4.1}$$

$$= \underset{\boldsymbol{a}_{t:t+H-1}}{\text{argmax}}\, \mathbb{E}_{s_{t+1:t+H}\sim M(\cdot|s_t,a_{t:t+H-1})}\left[\sum_{h=0}^{H-1} R(s_{t+h}, a_{t+h}) + \hat{V}(s_{t+H})\right] \tag{4.2}$$

$$\pi_{\text{MPC}}(s_t) = a^*_t \tag{4.3}$$

这里，H 被称为**规划范围**，而 $\hat{V}(s_{t+H})$ 是此 H 步前瞻过程中的奖励估计。我们可以通过使用预训练的建议策略 $a_t = \pi(s_t)$ 来加速优化过程，这可以用来指导搜索过程，如下所述。

请注意，MPC 根据当前状态 $\boldsymbol{s}_t$ 计算一个固定的动作序列 $\boldsymbol{a}_{t:t+H-1}$，也称为计划；由于未来动作 $\boldsymbol{a}_{t'}$ 对于 $t' > t$ 是独立于未来状态 $\boldsymbol{s}_{t'}$ 的，这被称为**开环控制器**。这种控制器在确定性环境中（其中 $\boldsymbol{s}_{t'}$ 可以从 $\boldsymbol{s}_t$ 和动作序列中计算出来）可以很好地工作，但在一般情况下，我们需要在每一步重新规划，因为实际的下一个状态是观察到的。因此，MPC 是一种创建**闭环控制器**的方法。

我们可以通过算法 10 中的伪代码将 MPC 与模型和政策 / 提案学习相结合，其中决策策略 $a_t = \pi_{\text{MPC}}(s_t)$ 是通过方程 (4.2) 实现的。如果我们想学习提案策略 $a_t = \pi(s_t)$，我们应该使用离线策略方法，因为训练数据（即使是在想象中）将由 π_{MPC} 收集，而不是由 π 收集。在学习和世界模型时，我们只需要它在当前状态附近局部准确，这意味着我们通常可以在 MPC 中使用比在背景规划方法中更简单的模型。

以下部分，我们讨论特定类型的多智能体协同控制（MPC）方法。MPC 与强化学习（RL）之间的进一步联系在 [Ber24]. 中讨论。

4.1.2 启发式搜索

如果状态空间和动作空间是有限的，我们可以精确地求解方程（4.2），尽管时间复杂度通常在 H 上是指数级的。然而，在许多情况下，我们可以剪枝掉无望的轨迹，从而使该方法在大规模问题中可行。

特别地，考虑一个离散、确定性的马尔可夫决策过程（MDP），其中奖励最大化对应于找到目标状态的最短路径。我们可以根据所有可能的行为扩展当前状态的后继状态，试图找到目标状态。由于搜索树随着深度的增加呈指数增长，我们可以使用一个启发式函数来优先选择要扩展的节点；这被称为最佳优先搜索，如图 4.1 所示。

如果启发式函数是对目标真实距离的乐观下界，则称为**可接受**。如果我们旨在最大化总奖励，可接受性意味着启发式函数是一个上界

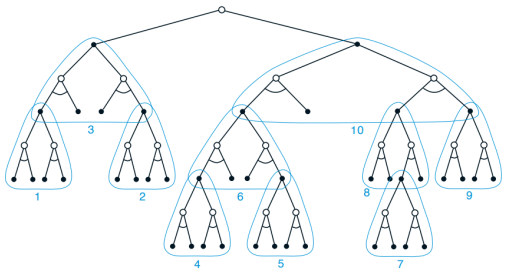


Figure 4.1: Illustration of heuristic search. In this figure, the subtrees are ordered according to a depth-first search procedure. From Figure 8.9 of [SB18]. Used with kind permission of Richard Sutton.

bound of the true value function. Admissibility ensures we will never incorrectly prune off parts of the search space. In this case, the resulting algorithm is known as A^* **search**, and is optimal. For more details on classical AI **heuristic search** methods, see [Pea84; RN19].

4.1.3 Monte Carlo tree search

Monte Carlo tree search or **MCTS** is similar to heuristic search, but learns a value function for each encountered state, rather than relying on a manually designed heuristic (see e.g., [Mun14] for details). MCTS is inspired by the upper confidence bound (UCB) method for bandits, but works for general MDPs [KS06].

4.1.3.1 AlphaGo and AlphaZero

The famous **AlphaGo** system [Sil+16], which was the first AI system to beat a human grandmaster at the board game Go, used the MCTS method, combined with a value function learned using RL, and a policy that was initialized using supervised learning from human demonstrations. This was followed up by **AlphaGoZero** [Sil+17a], which had a much simpler design, and did not train on any human data, i.e., it was trained entirely using RL and self play. It significantly outperformed the original AlphaGo. This was generalized to **AlphaZero** [Sil+18], which can play expert-level Go, chess, and shogi (Japanese chess), using a known model of the environment.

4.1.3.2 MuZero

AlphaZero assumes the world model is known. The **MuZero** method of [Sch+20] learns a world model, by training a latent representation of the observations, $\boldsymbol{z}_t = \phi(\boldsymbol{o}_t)$, and a corresponding latent dynamics model $\boldsymbol{z}_t = M(\boldsymbol{z}_t, a_t)$. The world model is trained to predict the immediate reward, the future reward (i..e, the value), and the optimal policy, where the optimal policy is computed using MCTS.

In more detail, to learn the model, MuZero uses a sum of 3 loss terms applied to each $(\boldsymbol{z}_{t-1}, a_t, \boldsymbol{z}_t, r_t)$ tuple in the replay buffer. The first loss is $\mathcal{L}(r_t, \hat{r}_t)$, where r_t is the observed reward and $\hat{r}_t = R(\boldsymbol{z}_t)$ is the predicted reward. The second loss is $\mathcal{L}(\boldsymbol{\pi}_t^{\text{MCTS}}, \boldsymbol{\pi}_t)$, where $\boldsymbol{\pi}_t^{\text{MCTS}}$ is the target policy from MCTS search (see below) and $\boldsymbol{\pi}_t = f(\boldsymbol{z}_t)$ is the predicted policy. The third loss is $\mathcal{L}(G_t^{\text{MCTS}}, v_t)$, where $G_t^{\text{MCTS}} = \sum_{i=0}^{n-1} \gamma^i r_{t+i} + \gamma^k v_{t+n}$ is the n-step bootstrap target value derived from MCTS search (see below), and $v_t = V(\boldsymbol{z}_t)$ is the predicted value from the current model.

To pick an action, MuZero does not use the policy directly. Instead it uses MCTS to rollout a search tree using the dynamics model, starting from the current state $\boldsymbol{z}_t$. It uses the predicted policy $\boldsymbol{\pi}_t$ and value v_t as heuristics to limit the breadth and depth of the search. Each time it expands a node in the tree, it assigns it a unique integer id (since we are assuming the dynamics are deterministic), thus lazily creating a discrete MDP. It then partially solves for the tabular Q function for this MDP using Monte Carlo rollouts, similar to real-time dynamic programming (Section 2.2.2).

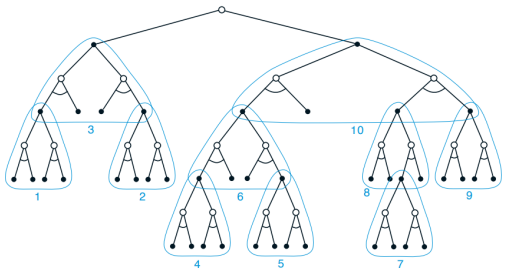


图 4.1：启发式搜索的示意图。在此图中，子树按照深度优先搜索程序进行排序。参见 *[SB18]* 的第 8.9 图。经 Richard Sutton 许可使用。

真实值函数的边界。可接受性确保我们永远不会错误地剪枝掉搜索空间的部分。在这种情况下，得到的算法被称为 A^* **搜索**，并且是最佳的。有关经典人工智能**启发式搜索**方法的更多详细信息，请参阅 [Pea84； RN19].

4.1.3 蒙特卡洛树搜索

蒙特卡洛树搜索 或 **MCTS** 类似于启发式搜索，但它为每个遇到的状态学习一个值函数，而不是依赖于手动设计的启发式（例如，参见 [Mun14] 的详细信息）。MCTS 受到赌徒上界（ UCB ）方法的启发，但适用于一般的 MDP [KS06]。

4.1.3.1 AlphaGo 和 AlphaZero

著名的 **AlphaGo** 系统 [Sil+16]，这是第一个击败人类围棋大师的人工智能系统，它使用了 MCTS 方法，结合了使用 RL 学习到的价值函数和通过人类演示进行监督学习初始化的策略。随后是 **AlphaGoZero** [Sil+17a], ，它具有更简单的结构，并且没有在人类数据上进行训练，即完全使用 RL 和自我对弈进行训练。它显著优于原始的 AlphaGo。这被推广到 **AlphaZero** [Sil+18]，它可以使用环境模型以专家水平玩围棋、象棋和将棋（日本象棋）。

4.1.3.2 MuZero

AlphaZero 假设世界模型已知。**MuZero** 方法通过训练观察的潜在表示 [Sch+20] 学习世界模型，$\boldsymbol{z}_t = \phi(\boldsymbol{o}_t)$，以及相应的潜在动力学模型 $\boldsymbol{z}_t = M(\boldsymbol{z}_t, a_t)$。世界模型被训练来预测即时奖励、未来奖励（即价值）和最佳策略，其中最佳策略是通过 MCTS 计算的。

更详细地说，为了学习模型，MuZero 使用对重放缓冲区中每个（ $\boldsymbol{z}_{t-1}, a_t, \boldsymbol{z}_t, r_t$ ）元组应用的总和 3 个损失项。第一个损失是 $\mathcal{L}(r_t, \hat{r}_t)$，其中 r_t 是观察到的奖励，$\hat{r}_t = R(\boldsymbol{z}_t)$ 是预测的奖励。第二个损失是 $\mathcal{L}(\boldsymbol{\pi}_t^{\text{MCTS}}, \boldsymbol{\pi}_t)$，其中 $\boldsymbol{\pi}_t^{\text{MCTS}}$ 是 MCTS 搜索中的目标策略（见下文），$\boldsymbol{\pi}_t = f(\boldsymbol{z}_t)$ 是预测的策略。第三个损失是 $\mathcal{L}(G_t^{\text{MCTS}}, v_t)$，其中 $G_t^{\text{MCTS}} = \sum_{i=0}^{n-1} \gamma^i r_{t+i} + \gamma^k v_{t+n}$ 是从 MCTS 搜索中得到的 n 步引导目标值（见下文），$v_t = V(\boldsymbol{z}_t)$ 是当前模型预测的值。

MuZero 选择动作时，不直接使用策略。相反，它使用 MCTS （蒙特卡洛树搜索）通过动力学模型展开搜索树，从当前状态 $\boldsymbol{z}_t$ 开始。它使用预测策略 $\boldsymbol{\pi}_t$ 和价值 v_t 作为启发式方法来限制搜索的广度和深度。每次它在树中展开一个节点时，都会给它分配一个唯一的整数 ID （因为我们假设动力学是确定性的），从而懒加载地创建一个离散的 MDP。然后，它使用蒙特卡洛回滚部分解决这个 MDP 的表格 Q 函数，类似于实时动态规划（第 2.2.2 节）。

In more detail, the MCTS process is as follows. Let $s^k = \boldsymbol{z}_t$ be the root node, for $k = 0$. We initialize $Q(s^k, a) = 0$ and $P(s^k, a) = \boldsymbol{\pi}_t(a|s^k)$, where the latter is the prior for each action. To select the action a^k to perform next (in the rollout), we use the UCB heuristic (Section 1.4.3) based on the empirical counts $N(s, a)$ combined with the prior policy, $P(s, a)$, which act as pseudocounts. After expanding this node, we create the child node $s^{k+1} = M(s^k, a^k)$; we initialize $Q(s^{k+1}, a) = 0$ and $P(s^{k+1}, a) = \boldsymbol{\pi}_t(a|s^{k+1})$, and repeat the process until we reach a maximum depth, where we apply the value function to the corresponding leaf node. We then compute the empirical sum of discounted rewards along each of the explored paths, and use this to update the $Q(s, a)$ and $N(s, a)$ values for all visited nodes. After performing 50 such rollouts, we compute the empirical distribution over actions at the root node to get the MCTS visit count policy, $\boldsymbol{\pi}_t^{\text{MCTS}}(a) = [N(s^0, a)/(\sum_b N(s^0, b))]^{1/\tau}$, where τ is a temperature. Finally we sample an action a_t from $\boldsymbol{\pi}_t^{\text{MCTS}}$, take a step, add $(o_t, a_t, r_t, \boldsymbol{\pi}_t^{\text{MCTS}}, G_t^{\text{MCTS}})$ to the replay buffer, compute the losses, update the model and policy parameters, and repeat.

The **Stochastic MuZero** method of [Ant+22] extends MuZero to allow for stochastic environments. The **Sampled MuZero** method of [Hub+21] extends MuZero to allow for large action spaces.

4.1.3.3 EfficientZero

The **Efficient Zero** paper [Ye+21] extends MuZero by adding an additional self-prediction loss to help train the world model. (See Section 4.3.2.2 for a discussion of such losses.) It also makes several other changes. In particular, it replaces the empirical sum of instantaneous rewards, $\sum_{i=0}^{n-1} \gamma^i r_{t+i}$, used in computing G_t^{MCTS}, with an LSTM model that predicts the sum of rewards for a trajectory starting at $\boldsymbol{z}_t$; they call this the value prefix. In addition, it replaces the stored value at the leaf nodes of trajectories in the replay buffer with new values, by rerunning MCTS using the current model applied to the leaves. They show that all three changes help, but the biggest gain is from the self-prediction loss. The recent **Efficient Zero V2** [Wan+24b] extends this to also work with continuous actions, by replacing tree search with sampling-based Gumbel search, amongst other changes.

4.1.4 Trajectory optimization for continuous actions

For continuous actions, we cannot enumerate all possible branches in the search tree. Instead, we can view Equation (4.2) as a standard optimization problem over the real valued sequence of vectors $\boldsymbol{a}_{t:t+H-1}$.

4.1.4.1 Random shooting

For general nonlinear models (such as neural networks), a simple approach is to pick a sequence of random actions to try, evaluate the reward for each trajectory, and pick the best. This is called **random shooting** [Die+07; Rao10].

4.1.4.2 LQG

If the system dynamics are linear and the reward function corresponds to negative quadratic cost, the optimal action sequence can be solved mathematically, as in the **linear-quadratic-Gaussian** (**LQG**) controller (see e.g., [AM89; HR17]).

If the model is nonlinear, we can use **differential dynamic programming** (**DDP**) [JM70; TL05]. In each iteration, DDP starts with a reference trajectory, and linearizes the system dynamics around states on the trajectory to form a locally quadratic approximation of the reward function. This system can be solved using LQG, whose optimal solution results in a new trajectory. The algorithm then moves to the next iteration, with the new trajectory as the reference trajectory.

4.1.4.3 CEM

It common to use black-box (gradient-free) optimization methods like the **cross-entropy method** or **CEM** in order to find the best action sequence. The CEM method is a simple derivative-free optimization method for

更详细地说，MCTS 过程如下。设 $s^k = \boldsymbol{z}_t$ 为根节点，对于 $k = 0$。我们初始化 $Q\ （\ s^k,\ \ a\ ）\ = 0$ 和 $P\ （\ s^k,\ \ a\ ）\ = \boldsymbol{\pi}_t\ （\ a|s^k\ ）$，其中后者是每个动作的先验。为了选择下一个要执行的动作 a^k（在 rollout 中），我们使用基于经验计数（，）并结合先验策略（，）的 UCB 启发式（第 1.4.3 节）。在展开此节点后，我们创建子节点 $s^{k+1} = M\ （\ s^k,\ \ a^k\ ）$；我们初始化 $Q\ （\ s^{k+1},\ \ a\ ）\ = 0$ 和 $P\ （\ s^{k+1},\ \ a\ ）\ = \boldsymbol{\pi}_t\ （\ a|s^{k+1}\ ）$，并重复此过程，直到达到最大深度，此时我们对相应的叶节点应用价值函数。然后我们计算每个探索路径上的折扣奖励的经验总和，并使用此信息更新所有访问节点的 $Q\ （\ s,\ \ a\ ）$ 和 $N\ （\ s,\ \ a\ ）$ 值。执行 50 次这样的 rollout 后，我们计算根节点上动作的经验分布以获得 MCTS 访问计数策略 $\boldsymbol{\pi}_t^{\mathrm{MCTS}}\ （\ a\ ）\ = [N\ （\ s^0,\ a\ ）\ /\ （\ \sum_b N\ （\ s^0,\ \ b\ ）\ ）\ ]^{1/\tau}$，其中 τ 是温度。最后，我们从 a_t 中采样一个动作，执行一步，将 $（\ o_t,\ a_t,\ \ r_t,\ \boldsymbol{\pi}_t^{\mathrm{MCTS}},\ \ G_t^{\mathrm{MCTS}}\ ）$ 添加到重放缓冲区，计算损失，更新模型和政策参数，并重复。

随机 MuZero **方法将 MuZero 扩展到随机环境**。采样 MuZero [方法将 MuZero 扩展到允许大动作空间。+22]

4.1.3.3 高效零

高效零**论文** Ye[扩展 MuZero，通过添加额外的自我预测损失来帮助训练世界模型。（参见第 +21] 4.3.2.2 节，讨论此类损失。）它还进行了其他一些更改。特别是，它用 LSTM 模型替换了在计算时使用的经验性即时奖励总和 $\sum_{i=0}^{n-1} \gamma^i r_{t+i}$，该模型预测从 G_t^{MCTS}开始的轨迹的奖励总和；他们称之为价值前缀。此外，它通过重新运行 MCTS 并使用当前模型应用于叶子节点来替换重放缓冲区中轨迹叶子节点存储的值。他们表明，这三个更改都有帮助，但最大的收益来自自我预测损失。最近的 $\boldsymbol{z}_t$ 高效零 V**Wan2** [b+24 扩展了这一方法，使其也能处理连续动作，通过替换树搜索为基于采样的 Gumbel 搜索，以及其他一些更改。]

4.1.4 连续动作的轨迹优化

对于连续动作，我们无法在搜索树中枚举所有可能的分支。相反，我们可以将方程（ 4.2 ）视为在向量 $\boldsymbol{a}_{t:t+H-1}$ 的实值序列上的标准优化问题。

4.1.4.1 随机射击

对于一般的非线性模型（如神经网络），一种简单的方法是选择一系列随机动作进行尝试，评估每条轨迹的奖励，并选择最佳方案。这被称为**随机射击** [Die+07； Rao10]。

4.1.4.2 LQG（原文中包含缩写和代码，翻译可能不必要）

如果系统动力学是线性的，并且奖励函数对应于负二次成本，则最优动作序列可以通过数学方法求解，如在 **线性 - 二次 - 高斯 （ LQG ）**控制器中（例如，参见 [AM89； HR17] ）。

如果模型是非线性的，我们可以使用 **微分动态规划 （ DDP ）** [JM70； TL05]。在每次迭代中，DDP 从参考轨迹开始，将系统动力学在轨迹上的状态线性化，以形成奖励函数的局部二次近似。该系统可以使用 LQG 求解，其最优解产生新的轨迹。然后算法移动到下一次迭代，以新的轨迹作为参考轨迹。

CEM

通常使用黑盒（无梯度）优化方法，如**交叉熵方法**或 **CEM**，以找到最佳动作序列。CEM 方法是一种简单的无导数优化方法，用于

continuous black-box functions $f : \mathbb{R}^D \to \mathbb{R}$. We start with a multivariate Gaussian, $\mathcal{N}(\boldsymbol{\mu}_0, \boldsymbol{\Sigma}_0)$, representing a distribution over possible action $\boldsymbol{a}$. We sample from this, evaluate all the proposals, pick the top K, then refit the Gaussian to these top K, and repeat, until we find a sample with sufficiently good score (or we perform moment matching on the top K scores). For details, see [Rub97; RK04; Boe+05].

In Section 4.1.4.4, we discuss the MPPI method, which is a common instantiation of CEM method. Another example is in the **TD-MPC** paper [HSW22a]. They learn the world model (dynamics model) in a latent space so as to predict future value and reward using temporal difference learning, and then use CEM to implement MPC for this world model. In [BXS20] they discuss how to combine CEM with gradient-based planning.

4.1.4.4 MPPI

The **model predictive path integral** or **MPPI** approach [WAT17] is a version of CEM. Originally MPPI was limited to models with linear dynamics, but it was extended to general nonlinear models in [Wil+17]. The basic idea is that the initial mean of the Gaussian at step t, namely $\boldsymbol{\mu}_t = \boldsymbol{a}_{t:t+H}$, is computed based on shifting $\hat{\boldsymbol{\mu}}_{t-1}$ forward by one step. (Here $\boldsymbol{\mu}_t$ is known as a reference trajectory.)

In [Wag+19], they apply this method for robot control. They consider a state vector of the form $\boldsymbol{s}_t = (\boldsymbol{q}_t, \dot{\boldsymbol{q}}_t)$, where $\boldsymbol{q}_t$ is the configuration of the robot. The deterministic dynamics has the form

$$\boldsymbol{s}_{t+1} = F(\boldsymbol{s}_t, \boldsymbol{a}_t) = \begin{pmatrix} \boldsymbol{q}_t + \dot{\boldsymbol{q}}_t \Delta t \\ \dot{\boldsymbol{q}}_t + f(\boldsymbol{s}_t, \boldsymbol{a}_t)\Delta t \end{pmatrix} \tag{4.4}$$

where f is a 2 layer MLP. This is trained using the **Dagger** method of [RGB11], which alternates between fitting the model (using supervised learning) on the current replay buffer (initialized with expert data), and then deploying the model inside the MPPI framework to collect new data.

4.1.4.5 GP-MPC

[KD18] proposed **GP-MPC**, which combines a Gaussian process dynamics model with model predictive control. They compute a Gaussian approximation to the future state trajectory given a candidate action trajectory, $p(\boldsymbol{s}_{t+1:t+H}|\boldsymbol{a}_{t:t+H-1}, \boldsymbol{s}_t)$, by moment matching, and use this to deterministically compute the expected reward and its gradient wrt $\boldsymbol{a}_{t:t+H-1}$. Using this, they can solve Equation (4.2) to find $\boldsymbol{a}^*_{t:t+H-1}$; finally, they execute the first step of this plan, a^*_t, and repeat the whole process.

The key observation is that moment matching is a deterministic operator that maps $p(\boldsymbol{s}_t|\boldsymbol{a}_{1:t-1})$ to $p(\boldsymbol{s}_{t+1}|\boldsymbol{a}_{1:t})$, so the problem becomes one of deterministic optimal control, for which many solution methods exist. Indeed the whole approach can be seen as a generalization of the **LQG** method from classical control, which assumes a (locally) linear dynamics model, a quadratic cost function, and a Gaussian distribution over states [Rec19]. In GP-MPC, the moment matching plays the role of local linearization.

The advantage of GP-MPC over the earlier method known as **PILCO** ("probabilistic inference for learning control"), which learns a policy by maximizing the expected reward from rollouts (see [DR11; DFR15] for details), is that GP-MPC can handle constraints more easily, and it can be more data efficient, since it continually updates the GP model after every step (instead of at the end of an trajectory).

4.1.5 SMC for MPC

A general way to tackle MPC — which supports discrete and continuous actions, as well as discrete and continuous states and linear and nonlinear world models — is to formulate it as the problem of posterior inference over state-action sequences with high reward. That is, following the control as inference framework discussed in Section 1.5, we define the goal as computing the following posterior:

$$p(\boldsymbol{x}_{1:T}|\boldsymbol{s}_1, O_{1:T}) \propto p(\boldsymbol{x}_{1:T}, O_{1:T}|\boldsymbol{s}_1) = \prod_{t=1}^{T-1} p(\boldsymbol{s}_{t+1}|\boldsymbol{a}_t, \boldsymbol{s}_t) \exp\left(\sum_{t=1}^{T} R(\boldsymbol{s}_t, \boldsymbol{a}_t) + \log p(\boldsymbol{a}_t)\right) \tag{4.5}$$

连续的黑盒函数 $f: \mathbb{R}^D \to \mathbb{R}$。我们从多元高斯分布 $\mathcal{N}(\boldsymbol{\mu}_0, \boldsymbol{\Sigma}_0)$ 开始，表示可能动作的分布。我们从其中采样，评估所有提案，选择前 K 个，然后对这些前 K 个进行高斯拟合，然后重复，直到找到一个得分足够好的样本（或者我们对前 K 个得分进行矩匹配）。有关详细信息，请参阅 [Rub97； RK04； Boe+05]。

在第 4.1.4.4 节中，我们讨论了 MPPI 方法，这是 CEM 方法的常见实例。另一个例子是在 **TD-MPC** 论文 [HSW22a] 中。他们在潜在空间中学习世界模型（动力学模型），以便使用时间差分学习预测未来值和奖励，然后使用 CEM 为此世界模型实现 MPC。在 [BXS20] 中，他们讨论了如何将 CEM 与基于梯度的规划相结合。

4.1.4.4 MPPI （原文中包含缩写和编号，翻译可能不必要，因此保留原文）

模型预测路径积分或 **MPPI** 方法 [WAT17] 是 CEM 的一种版本。最初 MPPI 仅限于线性动力学模型，但在 [Wil+17] 中扩展到了一般非线性模型。基本思想是在第 t 步计算高斯初始均值 $\boldsymbol{\mu}_t = \boldsymbol{a}_{t:t+H}$，即基于前移 $\hat{\boldsymbol{\mu}}_{t-1}$ 一步。（这里 $\boldsymbol{\mu}_t$ 被称为参考轨迹。）

在 [Wag+19]，他们应用这种方法进行机器人控制。他们考虑一种形式为 $\boldsymbol{s}_t = (\boldsymbol{q}_t, \dot{\boldsymbol{q}}_t)$ 的状态向量，其中 $\boldsymbol{q}_t$ 是机器人的配置。确定性动力学具有以下形式

$$\boldsymbol{s}_{t+1} = F(\boldsymbol{s}_t, \boldsymbol{a}_t) = \begin{pmatrix} \boldsymbol{q}_t + \dot{\boldsymbol{q}}_t \Delta t \\ \dot{\boldsymbol{q}}_t + f(\boldsymbol{s}_t, \boldsymbol{a}_t)\Delta t \end{pmatrix} \tag{4.4}$$

其中 f 是一个 2 层 MLP。这是使用 **Dagger** 方法 [RGB11]，进行训练的，该方法交替地在当前重放缓冲区（用专家数据初始化）上使用监督学习拟合模型，然后在 MPPI 框架内部部署模型以收集新数据。

4.1.4.5 GP-MPC

[KD18] 提出了 **GP-MPC**，该算法结合了高斯过程动力学模型与模型预测控制。他们通过矩匹配计算给定候选动作轨迹的未来状态轨迹的高斯近似 $p(\boldsymbol{s}_{t+1:t+H} | \boldsymbol{a}_{t:t+H-1}, \boldsymbol{s}_t)$，并使用此结果确定性地计算期望奖励及其关于 $\boldsymbol{a}_{t:t+H-1}$ 的梯度。利用这一点，他们可以求解方程（ 4.2 ）以找到 $\boldsymbol{a}^*_{t:t+H-1}$；最后，他们执行该计划的第一个步骤 a^*_t，并重复整个过程。

关键观察结果是，矩匹配是一个确定性算子，它将 $p(\boldsymbol{s}_t | \boldsymbol{a}_{1:t-1})$ 映射到 $p(\boldsymbol{s}_{t+1} | \boldsymbol{a}_{1:t})$，因此问题变成了确定性最优控制问题，对此存在许多解决方案。事实上，整个方法可以看作是从经典控制中推广的 **LQG** 方法，该方法假设一个（局部）线性动力学模型、一个二次成本函数以及状态上的高斯分布 [Rec19]。在 GP-MPC 中，矩匹配扮演局部线性化的角色。

GP-MPC 相对于之前称为 **PILCO** （ “ 用于学习控制的概率推理 ” ）的方法的优势在于，GP-MPC 可以更容易地处理约束，并且它可以更高效地使用数据，因为它在每一步之后都会不断更新 GP 模型（而不是在轨迹结束时）。

4.1.5 SMC for MPC

解决多智能体控制（ MPC ）的一种通用方法 —— 它支持离散和连续动作，以及离散和连续状态和线性和非线性世界模型 —— 是将它表述为具有高奖励状态 - 动作序列的后验推理问题。也就是说，遵循第 1.5 节中讨论的控制作为推理框架，我们将目标定义为计算以下后验：

$$p(\boldsymbol{x}_{1:T} | \boldsymbol{s}_1, O_{1:T}) \propto p(\boldsymbol{x}_{1:T}, O_{1:T} | \boldsymbol{s}_1) = \prod_{t=1}^{T-1} p(\boldsymbol{s}_{t+1} | \boldsymbol{a}_t, \boldsymbol{s}_t) \exp\left(\sum_{t=1}^{T} R(\boldsymbol{s}_t, \boldsymbol{a}_t) + \log p(\boldsymbol{a}_t) \right) \tag{4.5}$$

where $\boldsymbol{x}_t = (\boldsymbol{s}_t, \boldsymbol{a}_t)$, and O_t is the "optimality variable" which is clamped to the value 1, with distribution $p(O_t = 1|\boldsymbol{s}_t, \boldsymbol{a}_t) = \exp(R(s_t, a_t))$. (Henceforth we will assume a uniform prior over actions, so $p(\boldsymbol{a}_t) \propto 1$.) If we can sample from this distribution, we can find state-action sequences with high expected reward, and then we can just extract the first action from one of these sampled trajectories.[1]

In practice we only compute the posterior for h steps into the future, although we still condition on optimality out to the full horizon T. Thus we define our goal as computing

$$p(\boldsymbol{x}_{1:h}|O_{1:T}) \propto \underbrace{p(\boldsymbol{x}_{1:h}|O_{1:h})}_{\alpha_h(\boldsymbol{x}_{1:h})} \underbrace{p(O_{h+1:T}|\boldsymbol{x}_h)}_{\beta_h(\boldsymbol{x}_h)} \tag{4.6}$$

where $p(O_t = 1|s_t, a_t) = \exp(R(s_t, a_t))$ is the probability that the "optimality variable" obtains its observed (clamped) value of 1. We have decomposed the posterior as a forwards filtering term, $\alpha_h(\boldsymbol{x}_{1:h})$, and a backwards likelihood or smoothing term, $\beta_h(\boldsymbol{x}_h)$, as is standard in the literature on inference in state-space models (see e.g., [Mur23, Ch.8-9]). Note that if we define the value function as $V(\boldsymbol{s}_h) = \log p(O_{h:T}|\boldsymbol{s}_h)$, then the backwards message can be rewritten as follows [Pic+19]:

$$p(O_{h+1:T}|\boldsymbol{x}_h) = \mathbb{E}_{p(\boldsymbol{s}_{h+1}|\boldsymbol{x}_h)}\left[\exp(V(\boldsymbol{s}_{h+1}))\right] \tag{4.7}$$

A standard way to perform posterior inference in models such as these is to use **Sequential Monte Carlo** or **SMC**, which is an extension of particle filtering (i.e., sequential importance sampling with resampling) to a general sequence of distributions over a growing state space (see e.g., [Mur23, Ch 13.]). When combined with an approximation to the backwards message, the approach is called **twisted SMC** [BDM10; WL14; AL+16; Law+22; Zha+24]. This was applied to MPC in [Pic+19]. In particular, they suggest using SAC to learn a value function V, analogous to the backwards twist function, and policy π, which can be used to create the forwards proposal. More precisely, the policy can be combined with the world model $M(\boldsymbol{s}_t|\boldsymbol{s}_{t-1}, \boldsymbol{a}_{t-1})$ to give a (Markovian) proposal disribution over the next state and action:

$$q(\boldsymbol{x}_t|\boldsymbol{x}_{1:t-1}) = M(\boldsymbol{s}_t|\boldsymbol{s}_{t-1}, \boldsymbol{a}_{t-1})\pi(\boldsymbol{a}_t|\boldsymbol{s}_t) \tag{4.8}$$

This can then be used inside of an SMC algorithm to sample trajectories from the posterior in Equation (4.6). In particular, at each step, we sample from the proposal to extend each previous particle (sampled trajectory) by one step, and then reweight the corresponding particle using

$$w_t = \frac{p(\boldsymbol{x}_{1:T}|O_{1:T})}{q(\boldsymbol{x}_{1:t})} = \frac{p(\boldsymbol{x}_{1:t-1}|O_{1:T})p(\boldsymbol{x}_t|\boldsymbol{x}_{1:t-1}, O_{1:T})}{q(\boldsymbol{x}_{1:t-1})q(\boldsymbol{x}_t|\boldsymbol{x}_{1:t-1})} \tag{4.9}$$

$$= w_{t-1}\frac{p(\boldsymbol{x}_t|\boldsymbol{x}_{1:t-1}, O_{1:T})}{q(\boldsymbol{x}_t|\boldsymbol{x}_{1:t-1})} \propto w_{t-1}\frac{1}{q(\boldsymbol{x}_t|\boldsymbol{x}_{1:t-1})}\frac{p(\boldsymbol{x}_{1:t}|O_{1:T})}{p(\boldsymbol{x}_{1:t-1}|O_{1:T})} \tag{4.10}$$

Now plugging in the forward-backward equation from Equation (4.6), and doing some algebra, we get the following (see [Pic+19, App. A.4] for the detailed derivation):

$$w_t \propto w_{t-1}\frac{1}{q(\boldsymbol{x}_t|\boldsymbol{x}_{1:t-1})}\frac{p(\boldsymbol{x}_{1:t}|O_{1:t})p(O_{t+1:T}|\boldsymbol{x}_t)}{p(\boldsymbol{x}_{1:t-1}|O_{1:t-1})p(O_{t:T}|\boldsymbol{x}_{t-1})} \tag{4.11}$$

$$\propto w_{t-1}\mathbb{E}_{p(\boldsymbol{s}_{t+1}|\boldsymbol{s}_t, a_t)}\left[\exp(A(\boldsymbol{s}_t, \boldsymbol{a}_t, \boldsymbol{s}_{t+1}))\right] \tag{4.12}$$

where

$$A(\boldsymbol{s}_t, \boldsymbol{a}_t, \boldsymbol{s}_{t+1}) = r_t - \log\pi(\boldsymbol{a}_t|\boldsymbol{s}_t) + V(\boldsymbol{s}_{t+1}) - \mathbb{E}_{p(\boldsymbol{s}_t|\boldsymbol{s}_{t-1}, \boldsymbol{a}_{t-1})}\left[\exp(V(\boldsymbol{s}_t))\right] \tag{4.13}$$

is a maximum entropy version of an advantage function. We show the overall pseudocode in Algorithm 13.

An improved version of the above method, called **Critic SMC**, is presented in [Lio+22]. The main difference is that they first extend each of the N particles (sampled trajectories) by K possible "putative actions" a_i^{nk}, then score them using a learned heuristic function $Q(s_i^n, a_i^{nk})$, then resample N winners a_i^n from

[1] We should really marginalize over the state sequences, and then find the maximum marginal probability action sequence, as in Equation (4.2), but we approximate this by joint sampling, for simplicity. For more discussion on this point, see [LG+24].

$\boldsymbol{x}_t = (\boldsymbol{s}_t, \boldsymbol{a}_t)$，其中 O_t 是 “ 最优性变量 ”，其被限制为值 1，具有分布 $p(O_t = 1|\boldsymbol{s}_t, \boldsymbol{a}_t) = \exp(R(s_t, a_t))$。此后，我们将假设对动作具有均匀先验，因此 $p(\boldsymbol{a}_t) \propto 1$。如果我们能从这个分布中采样，我们可以找到具有高期望奖励的状态 - 动作序列，然后我们只需从这些采样轨迹中提取第一个动作即可。[1]

在实践中，我们只计算未来 h 步的后验，尽管我们仍然对整个预测范围 T 进行优化。因此，我们定义我们的目标为计算

$$p(\boldsymbol{x}_{1:h}|O_{1:T}) \propto \underbrace{p(\boldsymbol{x}_{1:h}|O_{1:h})}_{\alpha_h(\boldsymbol{x}_{1:h})} \underbrace{p(O_{h+1:T}|\boldsymbol{x}_h)}_{\beta_h(\boldsymbol{x}_h)} \tag{4.6}$$

其中 $p(O_t = 1|s_t, a_t) = \exp(R(s_t, a_t))$ 是 “ 最优性变量 ” 获得其观察到的（夹紧的）值为 1 的概率。我们将后验分解为前向滤波项，$\alpha_h(\boldsymbol{x}_{1:h})$，以及后向似然或平滑项，$\beta_h(\boldsymbol{x}_h)$，这是状态空间模型推断文献中的标准做法（例如，参见 [Mur23, 第 8 章 -9] ）。注意，如果我们定义值函数为 $V(\boldsymbol{s}_h) = \log p(O_{h:T}|\boldsymbol{s}_h)$，则后向消息可以重写如下 [Pic+19]:

$$p(O_{h+1:T}|\boldsymbol{x}_h) = \mathbb{E}_{p(\boldsymbol{s}_{h+1}|\boldsymbol{x}_h)} \left[\exp(V(\boldsymbol{s}_{h+1}))\right] \tag{4.7}$$

在类似这些模型中进行后验推理的标准方法是用 **顺序蒙特卡洛**或 **SMC**，它是对粒子滤波（即带有重采样的顺序重要性采样）的扩展，应用于不断增长的状态空间上的一个一般分布序列（例如，参见 [Mur23, 第 13 章。] ）。当与向后消息的近似结合时，该方法被称为 **扭曲 SMC** [BDM10； WL14； AL+16； Law +22； Zha+24]。这被应用于 [Pic+19] 中的 MPC。特别是，他们建议使用 SAC 学习一个价值函数 V，类似于向后扭曲函数，以及策略 π，可以用它来创建前向建议。更精确地说，策略可以与世界模型 $M(\boldsymbol{s}_t|\boldsymbol{s}_{t-1}, \boldsymbol{a}_{t-1})$ 结合，以给出（马尔可夫的）下一个状态和动作的提议分布：

$$q(\boldsymbol{x}_t|\boldsymbol{x}_{1:t-1}) = M(\boldsymbol{s}_t|\boldsymbol{s}_{t-1}, \boldsymbol{a}_{t-1})\pi(\boldsymbol{a}_t|\boldsymbol{s}_t) \tag{4.8}$$

这可以用于 SMC 算法中，从后验分布（方程 4.6 ）中采样轨迹。特别是，在每一步中，我们从提议分布中采样以扩展每个先前的粒子（采样轨迹）一步，然后使用重新加权相应的粒子。

$$w_t = \frac{p(\boldsymbol{x}_{1:T}|O_{1:T})}{q(\boldsymbol{x}_{1:t})} = \frac{p(\boldsymbol{x}_{1:t-1}|O_{1:T})p(\boldsymbol{x}_t|\boldsymbol{x}_{1:t-1}, O_{1:T})}{q(\boldsymbol{x}_{1:t-1})q(\boldsymbol{x}_t|\boldsymbol{x}_{1:t-1})} \tag{4.9}$$

$$= w_{t-1} \frac{p(\boldsymbol{x}_t|\boldsymbol{x}_{1:t-1}, O_{1:T})}{q(\boldsymbol{x}_t|\boldsymbol{x}_{1:t-1})} \propto w_{t-1} \frac{1}{q(\boldsymbol{x}_t|\boldsymbol{x}_{1:t-1})} \frac{p(\boldsymbol{x}_{1:t}|O_{1:T})}{p(\boldsymbol{x}_{1:t-1}|O_{1:T})} \tag{4.10}$$

现在将方程（ 4.6 ）中的前向 - 后向方程代入，并进行一些代数运算，我们得到以下结果（参见 [图 +19，附录 A4] 中的详细推导）：

$$w_t \propto w_{t-1} \frac{1}{q(\boldsymbol{x}_t|\boldsymbol{x}_{1:t-1})} \frac{p(\boldsymbol{x}_{1:t}|O_{1:t})p(O_{t+1:T}|\boldsymbol{x}_t)}{p(\boldsymbol{x}_{1:t-1}|O_{1:t-1})p(O_{t:T}|\boldsymbol{x}_{t-1})} \tag{4.11}$$

$$\propto w_{t-1} \mathbb{E}_{p(\boldsymbol{s}_{t+1}|\boldsymbol{s}_t, a_t)} \left[\exp(A(\boldsymbol{s}_t, \boldsymbol{a}_t, \boldsymbol{s}_{t+1}))\right] \tag{4.12}$$

在哪里

$$A(\boldsymbol{s}_t, \boldsymbol{a}_t, \boldsymbol{s}_{t+1}) = r_t - \log \pi(\boldsymbol{a}_t|\boldsymbol{s}_t) + V(\boldsymbol{s}_{t+1}) - \mathbb{E}_{p(\boldsymbol{s}_t|\boldsymbol{s}_{t-1}, \boldsymbol{a}_{t-1})} \left[\exp(V(\boldsymbol{s}_t))\right] \tag{4.13}$$

是一个优势函数的最大熵版本。我们在算法 13 中展示了整体伪代码。

上述方法的改进版本，称为 **Critic SMC**，在 [Lio+22] 中提出。主要区别在于他们首先将每个 N 粒子（采样轨迹）扩展到可能的 “ 假想动作 ” K，然后使用学习到的启发式函数 $Q(s_i^n, a_i^{nk})$ 对其进行评分，然后从 N 赢家 a_i^n 中进行重采样

[1] 我们应该对状态序列进行边缘化，然后找到最大边缘概率动作序列，如公式（4.2）所示，但为了简化，我们通过联合采样来近似。关于这一点更详细的讨论，请参阅[LG+24]。

Algorithm 13: SMC for MPC

def SMC-MPC($\boldsymbol{s}_t, M, \pi, V, H$)
Initialize particles: $\{\boldsymbol{s}_t^n = \boldsymbol{s}_t\}_{n=1}^N$
Initialize weights: $\{w_t^n = 1\}_{n=1}^N$
for $i = t : t + H$ **do**
    // Propose one-step extension
    $\{\boldsymbol{a}_i^n \sim \pi(\cdot|\boldsymbol{s}_i^n)\}$
    $\{(\boldsymbol{s}_{i+1}^n, r_i^n) \sim M(\cdot|\boldsymbol{s}_i^n, \boldsymbol{a}_i^n)\}$
    // Update weights
    $\{w_i^n \propto w_{i-1}^n \exp(A(\boldsymbol{s}_i^n, \boldsymbol{a}_i^n, \boldsymbol{s}_{i+1}^n))\}$
    // Resampling
    $\{\boldsymbol{x}_{1:i}^n\} \sim \text{Multinom}(n; w_i^1, \ldots, w_i^N)$
    $\{w_i^n = 1\}$
Sample $n \sim \text{Unif}(1 : N)$ // Pick one of the top samples
Return $\boldsymbol{a}_t^n$

this set of $N \times K$ particles, and then push these winners through the dynamics model to get $s_{i+1}^n \sim M(\cdot|s_i^n, a_i^n)$. Finally, they reweight the N particles by the advantage and resample, as before. This can be advantageous if the dynamics model is slow to evaluate, since we can evaluate K possible extensions just using the heuristic function. We can think of this as a form of stochastic beam search, where the beam has N candidates, and you expand each one using K possible actions, and then reduce the population (beam) back to N

4.2 Background planning

In Section 4.1, we discussed how to use models to perform decision time planning. However, this can be slow. Fortunately, we can amortize the planning process into a reactive policy. To do this, we can use the model to generate synthetic trajectories "in the background" (while executing the current policy), and use this imaginary data to train the policy; this is called "**background planning**". We discuss a game theoretic formulation of this setup in Section 4.2.1. Then in Section 4.2.2, we discuss ways to combine model-based and model-free learning. Finally, in Section 4.2.3, we discuss ways to deal with model errors, that might lead the policy astray.

4.2.1 A game-theoretic perspective on MBRL

In this section, we discuss a game-theoretic framework for MBRL, as proposed in [RMK20]. This provides a theoretical foundation for many of the more heuristic methods in the literature.

We denote the true world model by M_{env}. To simplify the notation, we assume an MDP setup with a known reward function, so all that needs to be learned is the world model, $\hat{M}$, representing $p(s'|s, a)$. (It is trivial to also learn the reward function.) We define the value of a policy π when rolled out in some model M' as the (discounted) sum of expected rewards:

$$J(\pi, M') = \mathbb{E}_{M',\pi}\left[\sum_{t=0}^{\infty} \gamma^t R(s_t)\right]$$

We define the loss of a model $\hat{M}$ given a distribution $\mu(s, a)$ over states and actions as

$$\ell(\hat{M}, \mu) = \mathbb{E}_{(s,a)\sim\mu}\left[D_{\mathbb{KL}}\left(M_{\text{env}}(\cdot|s, a) \parallel \hat{M}(\cdot|s, a)\right)\right]$$

Algorithm 13: SMC for MPC

def SMC-MPC($\boldsymbol{s}_t, M, \pi, V, H$)
Initialize particles: $\{\boldsymbol{s}_t^n = \boldsymbol{s}_t\}_{n=1}^N$
Initialize weights: $\{w_t^n = 1\}_{n=1}^N$
for $i = t : t+H$ **do**
  // Propose one-step extension
  $\{\boldsymbol{a}_i^n \sim \pi(\cdot|\boldsymbol{s}_i^n)\}$
  $\{(\boldsymbol{s}_{i+1}^n, r_i^n) \sim M(\cdot|\boldsymbol{s}_i^n, \boldsymbol{a}_i^n)\}$
  // Update weights
  $\{w_i^n \propto w_{i-1}^n \exp(A(\boldsymbol{s}_i^n, \boldsymbol{a}_i^n, \boldsymbol{s}_{i+1}^n))\}$
  // Resampling
  $\{\boldsymbol{x}_{1:i}^n\} \sim \text{Multinom}(n; w_i^1, \ldots, w_i^N)$
  $\{w_i^n = 1\}$
Sample $n \sim \text{Unif}(1:N)$ // Pick one of the top samples
Return $\boldsymbol{a}_t^n$

这组 $N \times K$ 粒子，然后将这些优胜者通过动力学模型推到 $s_{i+1}^n \sim M(\cdot|s_i^n, a_i^n)$。最后，他们通过优势重新加权 N 粒子，并像以前一样进行重采样。如果动力学模型评估速度慢，这可以是有利的，因为我们只需使用启发式函数就可以评估 K 可能的扩展。我们可以将这视为一种随机束搜索的形式，其中束有 N 个候选者，你使用 K 可能的行为扩展每一个，然后将人群（束）减少回 N

4.2 背景规划

在第 4.1 节中，我们讨论了如何使用模型进行决策时间规划。然而，这可能会很慢。幸运的是，我们可以将规划过程分摊到反应策略中。为此，我们可以使用模型在“后台”生成合成轨迹（在执行当前策略的同时），并使用这些想象中的数据来训练策略；这被称为“**后台规划**”。我们在第 4.2.1 节中讨论了这种设置的博弈论公式。然后，在第 4.2.2 节中，我们讨论了结合基于模型和无模型学习的方法。最后，在第 4.2.3 节中，我们讨论了处理可能导致策略偏离的模型错误的方法。

4.2.1 MBRL 的博弈论视角

在本节中，我们讨论了 RMK20] 中提出的 MBRL 博弈论框架。这为文献中许多更启发式的方法提供了理论基础。

我们用 M_{env} 表示真实世界模型。为了简化符号，我们假设一个具有已知奖励函数的 MDP 设置，因此需要学习的是世界模型 $\hat{M}$，它代表 $p(s'|s,a)$。(学习奖励函数也是一件简单的事情。) 我们定义在某个模型 M' 中执行策略 π 的价值为期望奖励的（折现）总和。

$$J(\pi, M') = \mathbb{E}_{M',\pi}\left[\sum_{t=0}^{\infty} \gamma^t R(s_t)\right]$$

我们定义在给定状态和动作分布 $\mu(s,a)$ 的情况下，模型 $\hat{M}$ 的损失为

$$\ell(\hat{M}, \mu) = \mathbb{E}_{(s,a)\sim\mu}\left[D_{\mathbb{KL}}\left(M_{\text{env}}(\cdot|s,a) \parallel \hat{M}(\cdot|s,a)\right)\right]$$

We now define MBRL as a two-player general-sum game:

$$\overbrace{\max_{\pi} J(\pi, \hat{M})}^{\text{policy player}}, \overbrace{\min_{\hat{M}} \ell(\hat{M}, \mu^{\pi}_{M_{\text{env}}})}^{\text{model player}}$$

where $\mu^{\pi}_{M_{\text{env}}} = \frac{1}{T}\sum_{t=0}^{T} M_{\text{env}}(s_t = s, a_t = a)$ as the induced state visitation distribution when policy π is applied in the real world M_{env}, so that minimizing $\ell(\hat{M}, \mu^{\pi}_{M_{\text{env}}})$ gives the **maximum likelihood estimate** for $\hat{M}$.

Now consider a **Nash equilibrium** of this game, that is a pair $(\pi, \hat{M})$ that satisfies $\ell(\hat{M}, \mu^{\pi}_{M_{\text{env}}}) \leq \epsilon_{M_{\text{env}}}$ and $J(\pi, \hat{M}) \geq J(\pi', \hat{M}) - \epsilon_{\pi}$ for all π'. (That is, the model is accurate when predicting the rollouts from π, and π cannot be improved when evaluated in $\hat{M}$). In [RMK20] they prove that the Nash equilibirum policy π is near optimal wrt the real world, in the sense that $J(\pi^*, M_{\text{env}}) - J(\pi, M_{\text{env}})$ is bounded by a constant, where π^* is an optimal policy for the real world M_{env}. (The constant depends on the ϵ parameters, and the TV distance between $\mu^{\pi^*}_{M_{\text{env}}}$ and $\mu^{\pi*}_{\hat{M}}$.)

A natural approach to trying to find such a Nash equilibrium is to use **gradient descent ascent** or **GDA**, in which each player updates its parameters simultaneously, using

$$\pi_{k+1} = \pi_k + \eta_{\pi} \nabla_{\pi} J(\pi_k, \hat{M}_k)$$
$$\hat{M}_{k+1} = \hat{M}_k - \eta_M \nabla_{\hat{M}} \ell(\hat{M}_k, \mu^{\pi_k}_{M_{\text{env}}})$$

Unfortunately, GDA is often an unstable algorithm, and often needs very small learning rates η. In addition, to increase sample efficiency in the real world, it is better to make multiple policy improvement steps using synthetic data from the model $\hat{M}_k$ at each step.

Rather than taking small steps in parallel, the **best response** strategy fully optimizes each player given the previous value of the other player, in parallel:

$$\pi_{k+1} = \operatorname*{argmax}_{\pi} J(\pi, \hat{M}_k)$$
$$\hat{M}_{k+1} = \operatorname*{argmin}_{\hat{M}} \ell(\hat{M}, \mu^{\pi_k}_{M_{\text{env}}})$$

Unfortunately, making such large updates in parallel can often result in a very unstable algorithm.

To avoid the above problems, [RMK20] propose to replace the min-max game with a **Stackelberg game**, which is a generalization of min-max games where we impose a specific player ordering. In particular, let the players be A and B, let their parameters be θ_A and θ_B, and let their losses be $\mathcal{L}_A(\theta_A, \theta_B)$ and $\mathcal{L}_B(\theta_A, \theta_B)$. If player A is the leader, the Stackelberg game corresponds to the following nested optimization problem, also called a bilevel optimization problem:

$$\min_{\theta_A} \mathcal{L}_A(\theta_A, \theta^*_B(\theta_A)) \quad \text{s.t.} \quad \theta^*_B(\theta_A) = \operatorname*{argmin}_{\theta} \mathcal{L}_B(\theta_A, \theta)$$

Since the follower B chooses the best response to the leader A, the follower's parameters are a function of the leader's. The leader is aware of this, and can utilize this when updating its own parameters.

The main advantage of the Stackelberg approach is that one can derive gradient-based algorithms that will provably converge to a local optimum [CMS07; ZS22]. In particular, suppose we choose the **policy as leader** (**PAL**). We then just have to solve the following optimization problem:

$$\hat{M}_{k+1} = \operatorname*{argmin}_{\hat{M}} \ell(\hat{M}, \mu^{\pi_k}_{M_{\text{env}}})$$
$$\pi_{k+1} = \pi_k + \eta_{\pi} \nabla_{\pi} J(\pi_k, \hat{M}_{k+1})$$

We can solve the first step by executing π_k in the environment to collect data $\mathcal{D}_k$ and then fitting a local (policy-specific) dynamics model by solving $\hat{M}_{k+1} = \operatorname{argmin} \ell(\hat{M}, \mathcal{D}_k)$. (For example, this could be a locally

我们现在将 MBRL 定义为两人零和博弈。

$$\overbrace{\max_{\pi} J(\pi, \hat{M})}^{\text{policy player}}, \overbrace{\min_{\hat{M}} \ell(\hat{M}, \mu^{\pi}_{M_{\text{env}}})}^{\text{model player}}$$

作为在真实世界 M_{env} 中应用策略 π 时诱导状态访问分布的 where$\mu^{\pi}_{M_{\text{env}}} = {}^{1}_{T} \sum^{T}_{t=0} M_{\text{env}}(s_t = s, a_t = a)$，因此最小化 $\ell(\hat{M}, \mu^{\pi}_{M_{\text{env}}})$ 给出了**最大似然估计**对于$\hat{M}$。

现在考虑一个 **纳什均衡**，即一对 $(\pi, \hat{M})$，它满足 $\ell(\hat{M}, \mu^{\pi}_{M_{\text{env}}}) \leq \epsilon_{M_{\text{env}}}$ 和 $J(\pi, \hat{M}) \geq J(\pi', \hat{M}) - \epsilon_{\pi}$ 对于所有 π'。（也就是说，当从 π 预测 rollout 时，模型是准确的，并且 π 在 $\hat{M}$评估时无法改进）。在 [RMK20] 中，他们证明了纳什均衡策略 π 在现实世界中的近似最优，即 $J(\pi^*, M_{\text{env}}) - J(\pi, M_{\text{env}})$ 被一个常数所限制，其中 π^* 是现实世界的最优策略 M_{env}。（该常数取决于 ϵ 参数，以及 $\mu^{\pi^*}{}_{M_{\text{env}}}$ 和 $\mu^{\pi^*}_{\hat{M}}$ 之间的 TV 距离。）

一种寻找此类纳什均衡的自然方法是使用**梯度下降上升**或 **GDA**，其中每个玩家同时更新其参数。

$$\pi_{k+1} = \pi_k + \eta_{\pi} \nabla_{\pi} J(\pi_k, \hat{M}_k)$$
$$\hat{M}_{k+1} = \hat{M}_k - \eta_M \nabla_{\hat{M}} \ell(\hat{M}_k, \mu^{\pi_k}_{M_{\text{env}}})$$

不幸的是，GDA 通常是一个不稳定的算法，并且通常需要非常小的学习率 η。此外，为了在现实世界中提高样本效率，最好在每一步使用模型 $\hat{M}_k$ 的合成数据进行多次策略改进步骤。

而不是并行采取小步骤，最佳响应策略策略性地优化每个玩家，考虑到其他玩家的先前值，并行进行：

$$\pi_{k+1} = \operatorname*{argmax}_{\pi} J(\pi, \hat{M}_k)$$
$$\hat{M}_{k+1} = \operatorname*{argmin}_{\hat{M}} \ell(\hat{M}, \mu^{\pi_k}_{M_{\text{env}}})$$

很不幸，同时进行如此大的更新往往会导致算法非常不稳定。

为避免上述问题，[RMK20] 建议用 Stackelberg 博弈替换 min-max 博弈，这是一种对 min-max 博弈的推广，其中我们施加特定的玩家排序。特别是，让玩家为 A 和 B，让他们的参数为 θ_A 和 θ_B，让他们的损失为 $\mathcal{L}_A(\theta_A, \theta_B)$ 和 $\mathcal{L}_B(\theta_A, \theta_B)$。如果玩家 A 是领导者，Stackelberg 博弈对应以下嵌套优化问题，也称为双层优化问题：

$$\min_{\theta_A} \mathcal{L}_A(\theta_A, \theta^*_B(\theta_A)) \quad \text{s.t.} \quad \theta^*_B(\theta_A) = \operatorname*{argmin}_{\theta} \mathcal{L}_B(\theta_A, \theta)$$

由于追随者 B 选择了对领导者 A 的最佳回应，追随者的参数是领导者参数的函数。领导者知道这一点，并且可以利用这一点来更新自己的参数。

Stackelberg 方法的主要优势在于可以推导出基于梯度的算法，这些算法可以保证收敛到局部最优解 [CMS07； ZS22]。特别是，假设我们选择**策略作为领导者（PAL）**。然后我们只需解决以下优化问题：

$$\hat{M}_{k+1} = \operatorname*{argmin}_{\hat{M}} \ell(\hat{M}, \mu^{\pi_k}_{M_{\text{env}}})$$
$$\pi_{k+1} = \pi_k + \eta_{\pi} \nabla_{\pi} J(\pi_k, \hat{M}_{k+1})$$

我们可以通过在环境中执行 π_k 来解决第一步，收集数据 $\mathcal{D}_k$，然后通过求解 $\hat{M}_{k+1} = \operatorname{argmin}_{\ell} \ell(\hat{M}, \mathcal{D}_k)$ 来拟合一个局部（策略特定）动力学模型。（例如，这可以是局部

linear model, such as those used in trajectory optimization methods discussed in Section 4.1.4.4.) We then (slightly) improve the policy to get π_{k+1} using a conservative update algorithm, such as natural actor-critic (Section 3.3.4) or TRPO (Section 3.4.2), on "imaginary" model rollouts from $\hat{M}_{k+1}$.

Alternatively, suppose we choose the **model as leader** (**MAL**). We now have to solve

$$\pi_{k+1} = \operatorname*{argmax}_{\pi} J(\pi, \hat{M}_k)$$
$$\hat{M}_{k+1} = \hat{M}_k - \eta_M \nabla_{\hat{M}} \ell(\hat{M}, \mu_{M_{\text{env}}}^{\pi_{k+1}})$$

We can solve the first step by using any RL algorithm on "imaginary" model rollouts from $\hat{M}_k$ to get π_{k+1}. We then apply this in the real world to get data $\mathcal{D}_{k+1}$, which we use to slightly improve the model to get $\hat{M}_{k+1}$ by using a conservative model update applied to $\mathcal{D}_{k+1}$. (In practice we can implement a conservative model update by mixing $\mathcal{D}_{k+1}$ with data generated from earlier models, an approach known as **data aggregation** [RB12].) Compared to PAL, the resulting model will be a more global model, since it is trained on data from a mixture of policies (including very suboptimal ones at the beginning of learning).

4.2.2 Dyna

The **Dyna** paper [Sut90] proposed an approach to MBRL that is related to the approach discussed in Section 4.2.1, in the sense that it trains a policy and world model in parallel, but it differs in one crucial way: the policy is also trained on real data, not just imaginary data. That is, we define $\pi_{k+1} = \pi_k + \eta_\pi \nabla_\pi J(\pi_k, \hat{D}_k \cup \mathcal{D}_k)$, where $\mathcal{D}_k$ is data from the real environment and $\hat{D}_k = \text{rollout}(\pi_k, \hat{M}_k)$ is imaginary data from the model. This makes Dyna a hybrid model-free and model-based RL method, rather than a "pure" MBRL method.

In more detail, at each step of Dyna, the agent collects new data from the environment and adds it to a real replay buffer. This is then used to do an off-policy update. It also updates its world model given the real data. Then it simulates imaginary data, starting from a previously visited state (see `sample-init-state` function in Algorithm 10), and rolling out the current policy in the learned model. The imaginary data is then added to the imaginary replay buffer and used by an on-policy learning algorithm. This process continue until the agent runs out of time and must take the next step in the environment.

4.2.2.1 Tabular Dyna

The original Dyna paper was developed under the assumption that the world model $s' = M(s, a)$ is deterministic and tabular, and the Q function is also tabular. See Algorithm 14 for the simplified pseudocode for this case. Since we assume a deterministic world model of the form $s' = M(s, a)$, then sampling a single step from this starting at a previously visited state is equivalent to experience replay (Section 2.5.2.3). Thus we can think of ER as a kind of non-parametric world model [HHA19].

4.2.2.2 Dyna with function approximation

It is easy to extend Dyna to work with function approximation and policy gradient methods. The code is identical to the MBRL code in Algorithm 10, where now we train the policy on real as well as imaginary data. ([Lai+21] argues that we should gradually increase the fraction of real data that is used to train the policy, to avoid suboptimal performance due to model limitations.) If we use real data from the replay buffer, we have to use an off-policy learner, since the replay buffer contains trajectories that may have been generated from old policies. (The most recent real trajectory, and all imaginary trajectories, are always from the current policy.)

We now mention some examples of this "generalized Dyna" framework. In [Sut+08] they extended Dyna to the case where the Q function is linear, and in [HTB18] they extended it to the DQN case. In [Jan+19a], they present the **MBPO** (model based policy optimization) algorithm, which uses Dyna with the off-policy SAC method. Their world model is an **ensemble of DNNs**, which generates diverse predictions (an approach which was originally proposed in the **PETS** (probabilistic ensembles with trajectory sampling) paper of [Chu+18]). In [Kur+19], they combine Dyna with TRPO (Section 3.4.2) and ensemble world models, and

线性模型，例如在第 4.1.4.4 节中讨论的轨迹优化方法中使用的模型。然后我们（略微）改进策略，通过使用保守的更新算法，例如自然演员 - 评论家（第 3.3.4 ）或 TRPO （第 3.4.2 ），在 “ 想象 ” 的模型回滚中使用 $\hat{M}_{k+1}$。

或者，假设我们选择**模型作为领导者**（ **MAL** ）。我们现在必须解决

$$\pi_{k+1} = \operatorname*{argmax}_{\pi} J(\pi, \hat{M}_k)$$
$$\hat{M}_{k+1} = \hat{M}_k - \eta_M \nabla_{\hat{M}} \ell(\hat{M}, \mu_{M_{\text{env}}}^{\pi_{k+1}})$$

我们可以通过在 “ 想象 ” 模型回放上使用任何 RL 算法从 $\hat{M}_k$ 中获取 π_{k+1} 来解决问题。然后我们将此应用于现实世界以获取数据 $\mathcal{D}_{k+1}$，我们使用这些数据略微改进模型，通过在上应用保守的模型更新来获得 $\hat{M}_{k+1}$。在实践中，我们可以通过将 $\mathcal{D}_{k+1}$ 与从早期模型生成的数据混合来实现保守的模型更新，这种方法被称为**数据聚合** [RB12]。与 PAL 相比，生成的模型将是一个更全局的模型，因为它是在来自策略混合（包括学习初期非常次优的策略）的数据上训练的。

4.2.2 动态

Dyna **论文** 提出了一个与第 [4.2.190] 节中讨论的方法相关的方法，即 MBRL，其意义在于它并行训练策略和世界模型，但在一个关键方面有所不同：策略也在真实数据上训练，而不仅仅是想象中的数据。也就是说，我们定义了 ($\pi_{k+1} = \pi_k + \eta_\pi \nabla_\pi J, \pi_k$)，其中 $\hat{D}_k \cup \mathcal{D}_k$是真实环境中的数据，而 $\mathcal{D}_k$ rollout($\hat{D}_k =, \pi_k$) $\hat{M}_k$是模型中的想象数据。这使得 Dyna 成为一个混合的无模型和基于模型的 RL 方法，而不是 “ 纯粹 ” 的 MBRL 方法。

更详细地说，在 Dyna 的每个步骤中，智能体从环境中收集新数据并将其添加到真实重放缓冲区。然后使用这些数据执行离线策略更新。它还会根据真实数据更新其世界模型。然后，它从之前访问过的状态开始模拟想象中的数据（参见 `sample-init-state` 函数在算法 10 中），并在学习模型中执行当前策略。然后将想象中的数据添加到想象的重放缓冲区，并用于在线学习算法。这个过程一直持续到智能体耗尽时间，必须在环境中采取下一步行动。

4.2.2.1 表格式动态

原始的 Dyna 论文是在假设世界模型 $s' = M(s, a)$ 是确定性和表格形式的，并且该 Q 函数也是表格形式的。参见算法 14 ，其中给出了此情况简化的伪代码。由于我们假设世界模型是 $s' = M(s, a)$ 的形式，那么从先前访问的状态开始采样一个步骤等同于经验回放（第 2.5.2.3 节）。因此，我们可以将 ER 视为一种非参数化的世界模型 [HHA19]。

4.2.2.2 动态函数逼近

将 Dyna 扩展到与函数逼近和策略梯度方法一起工作很容易。代码与算法 10 中的 MBRL 代码相同，现在我们在真实数据和虚拟数据上训练策略。（ [Lai+21] 认为，我们应该逐渐增加用于训练策略的真实数据比例，以避免由于模型限制导致的次优性能。）如果我们使用重放缓冲区中的真实数据，我们必须使用离线策略学习者，因为重放缓冲区包含可能由旧策略生成的轨迹。（最近的真实轨迹和所有虚拟轨迹始终来自当前策略。）

我们现在提到了这个 “ 广义 Dyna” 框架的一些例子。在 [Sut+08] 他们扩展了 Dyna 到函数线性的情况，在 [HTB18] 他们将其扩展到 DQN 的情况。在 [Jan+19a]，他们提出了 **MBPO** （基于模型的策略优化）算法，该算法使用 Dyna 与离线策略 SAC 方法。他们的世界模型是一个 **DNN 的集成**，它生成多样化的预测（这是一种最初在 **PETS** （具有轨迹采样的概率集成）论文中提出的 [Chu+18] ）。在 [Kur+19]，他们将 Dyna 与 TRPO （第 3.4.2 节）和集成世界模型相结合，并且

Algorithm 14: Tabular Dyna-Q

def dyna-Q-agent($s, M_{\text{env}}; \epsilon, \eta, \gamma$):
Initialize data buffer $\mathcal{D} = \emptyset$, $Q(s,a) = 0$ and $\hat{M}(s,a) = 0$
repeat
  // Collect real data from environment
  $a = \text{eps-greedy}(Q, \epsilon)$
  $(s', r) = \text{env.step}(s, a)$
  $\mathcal{D} = \mathcal{D} \cup \{(s, a, r, s')\}$
  // Update policy on real data
  $Q(s,a) := Q(s,a) + \eta[r + \gamma \max_{a'} Q(s', a') - Q(s,a)]$
  // Update model on real data
  $\hat{M}(s,a) = (s', r)$
  $s := s'$
  // Update policy on imaginary data
  for *n=1:N* **do**
    Select (s,a) from $\mathcal{D}$
    $(s', r) = \hat{M}(s,a)$
    $Q(s,a) := Q(s,a) + \eta[r + \gamma \max_{a'} Q(s', a') - Q(s,a)]$
until *until converged*

in [Wu+23] they combine Dyna with PPO and GP world models. (Technically speaking, these on-policy approaches are not valid with Dyna, but they can work if the replay buffer used for policy training is not too stale.)

4.2.3 Dealing with model errors and uncertainty

The theory in Section 4.2.1 tells us that the model-as-leader approach, which trains a new policy in imagination at each inner iteration while gradually improving the model in the outer loop, will converge to the optimal policy, provided the model converges to the true model (or one that is value equivalent to it, see Section 4.3.2.1). This can be assured provided the model is sufficiently powerful, and the policy explores sufficiently widely to collect enough diverse but task-relevant data. Nevertheless, models will inevitably have errors, and it can be useful for the policy learning to be aware of this. We discuss some approaches to this below.

4.2.3.1 Avoiding compounding errors in rollouts

In MBRL, we have to rollout imaginary trajectories to use for training the policy. It makes intuitive sense to start from a previously visited real-world state, since the model will likely be reliable there. We should start rollouts from different points along each real trajectory, to ensure good state coverage, rather than just expanding around the initial state [Raj+17]. However, if we roll out too far from a previously seen state, the trajectories are likely to become less realistic, due to **compounding errors** from the model [LPC22].

In [Jan+19a], they present the MBPO method, which uses short rollouts (inside Dyna) to prevent compounding error (an approach which is justified in [Jia+15]). [Fra+24] is a recent extension of MBPO which dynamically decides how much to roll out, based on model uncertainty.

Another approach to mitigating compounding errors is to learn a trajectory-level dynamics model, instead of a single-step model, see e.g., [Zho+24] which uses diffusion to train $p(s_{t+1:t+H}|s_t, a_{t:t+H-1})$, and uses this inside an MPC loop.

If the model is able to predict a reliable distribution over future states, then we can leverage this uncertainty estimate to compute an estimate of the expected reward. For example, PILCO [DR11; DFR15] uses Gaussian processes as the world model, and uses this to analytically derive the expected reward over trajectories as a function of policy parameters, which are then optimized using a deterministic second-order

Algorithm 14: Tabular Dyna-Q

def dyna-Q-agent($s, M_{\text{env}}; \epsilon, \eta, \gamma$):
Initialize data buffer $\mathcal{D} = \emptyset$, $Q(s,a) = 0$ and $\hat{M}(s,a) = 0$
repeat
  // Collect real data from environment
  $a = \text{eps-greedy}(Q, \epsilon)$
  $(s', r) = \text{env.step}(s, a)$
  $\mathcal{D} = \mathcal{D} \cup \{(s, a, r, s')\}$
  // Update policy on real data
  $Q(s,a) := Q(s,a) + \eta[r + \gamma \max_{a'} Q(s', a') - Q(s,a)]$
  // Update model on real data
  $\hat{M}(s,a) = (s', r)$
  $s := s'$
  // Update policy on imaginary data
  for *n=1:N* **do**
    Select (s,a) from $\mathcal{D}$
    $(s', r) = \hat{M}(s,a)$
    $Q(s,a) := Q(s,a) + \eta[r + \gamma \max_{a'} Q(s', a') - Q(s,a)]$
until *until converged*

在 [吴 +23] 他们结合了 Dyna 、 PPO 和 GP 世界模型。 (从技术上讲，这些基于策略的方法与 Dyna 不兼容，但如果用于策略训练的重放缓冲区不太陈旧，它们仍然可以工作。)

4.2.3 处理模型错误和不确定性

第 4.2.1 节中的理论告诉我们，模型作为领导者方法，在每次内部迭代中在想象中训练新的策略，同时在外部循环中逐渐改进模型，将收敛到最优策略，前提是模型收敛到真实模型（或与其价值等效的模型，见第 4.3.2.1 节）。只要模型足够强大，并且策略探索足够广泛以收集足够多样但与任务相关的数据，就可以保证这一点。然而，模型不可避免地会有错误，对于策略学习来说，意识到这一点可能是有用的。我们下面讨论一些处理这个问题的方法。

4.2.3.1 避免在部署中累积错误

在 MBRL 中，我们必须推出想象中的轨迹来用于训练策略。从之前访问过的真实世界状态开始似乎很直观，因为模型在那里可能更可靠。我们应该从每个真实轨迹的不同点开始推出，以确保良好的状态覆盖，而不仅仅是围绕初始状态扩展 [Raj+17]。然而，如果我们从之前看到的状态推出得太远，由于 **累积误差** 来自模型 [LPC22]，轨迹可能变得不太现实。

在 [1 月 +19a], ，他们提出了 MBPO 方法，该方法使用短滚动（在 Dyna 内部）来防止累积误差（这种方法在 [Jia+15] 中得到证实）。[Fra+24] 是 MBPO 的最近扩展，它根据模型不确定性动态决定滚动多少。

另一种减轻累积误差的方法是学习一个轨迹级动力学模型，而不是单步模型，例如，参见 [Zho+24] 该模型使用扩散进行训练 $p(s_{t+1:t+H}|s_t, a_{t:t+H-1})$，并在 MPC 循环中使用它。

如果模型能够预测未来状态的可靠分布，那么我们可以利用这个不确定性估计来计算期望奖励的估计。例如，PILCO [DR11 ； DFR15] 使用高斯过程作为世界模型，并利用此模型分析地推导出作为策略参数函数的轨迹上的期望奖励，然后使用确定性二阶进行优化。

gradient-based solver. In [Man+19], they combine the MPO algorithm (Section 3.4.4) for continuous control with **uncertainty sets** on the dynamics to learn a policy that optimizes for a worst case expected return objective.

4.2.3.2 End-to-end differentiable learning of model and planner

One solution to the mismatch problem between model fitting and policy learning is to use **differentiable planning**, in which we learn the model so as to minimize the planning loss. This bilevel optimization problem was first proposed in the **Value Iteration Network** paper of [Tam+16] and extended in the **TreeQN** paper of [Far+18]. In [AY20] they proposed a version of this for continuous actions based on the differentiable cross entropy method. In [Nik+22; Ban+23] they propose to use implicit differentation to avoid explicitly unrolling the inner optimization.

4.2.3.3 Unified model and planning variational lower bound

In [Eys+22], they propose a method called **Mismatched No More** (MNM) to solve the objective mismatch problem. They define an optimality variable (see Section 1.5) based on the entire trajectory, $p(O=1|\tau) = R(\tau) = \sum_{t=1}^{\infty} \gamma^t R(s_t, a_t)$. This gives rise to the following variational lower bound on the log probability of optimality:

$$\log p(O=1) = \log \int_\tau P(O=1,\tau) = \log \mathbb{E}_{P(\tau)}\left[P(O=1|\tau)\right] \geq \mathbb{E}_{Q(\tau)}\left[\log R(\tau) + \log P(\tau) - \log Q(\tau)\right]$$

where $P(\tau)$ is the distribution over trajectories induced by policy applied to the true world model, $P(\tau) = \mu(s_0)\prod_{t=0}^{\infty} M(s_{t+1}|s_t,a_t)\pi(a_t|s_t)$, and $Q(\tau)$ is the distribution over trajectories using the estimated world model, $Q(\tau) = \mu(s_0)\prod_{t=0}^{\infty} \hat{M}(s_{t+1}|s_t,a_t)\pi(a_t|s_t)$. They then maximize this bound wrt π and $\hat{M}$.

In [Ghu+22] they extend MNM to work with images (and other high dimensional states) by learning a latent encoder $\hat{E}(\boldsymbol{z}_t|\boldsymbol{o}_t)$ as well as latent dynamics $\hat{M}(\boldsymbol{z}_{t+1}|\boldsymbol{z}_t, a_t)$, similar to other self-predictive methods (Section 4.3.2.2). They call their method **Aligned Latent Models**.

4.2.3.4 Dynamically switching between MFRL and MBRL

One problem with the above methods is that, if the model is of limited capacity, or if it learns to model "irrelevant" aspects of the environment, then any MBRL method may be dominated by a MFRL method that directly optimizes the true expected reward. A safer approach is to use a model-based policy only when the agent is confident it is better, but otherwise to fall back to a model-free policy. This is the strategy proposed in the **Unified RL** method of [Fre+24].

4.3 World models

In this section, we discuss various kinds of world models that have been proposed in the literature. These can be used for decision-time planning or for background planning

4.3.1 Generative world models

In this section, we discuss different kinds of world model $M(\boldsymbol{s}'|\boldsymbol{s}, a)$. We can use this to generate imaginary trajectories by sampling from the following joint distribution:

$$p(\boldsymbol{s}_{t+1:T}, \boldsymbol{r}_{t+1:T}, \boldsymbol{a}_{t:T-1}|\boldsymbol{s}_t) = \prod_{i=t}^{T-1} \pi(\boldsymbol{a}_i|\boldsymbol{s}_i) M(\boldsymbol{s}_{i+1}|\boldsymbol{s}_i, \boldsymbol{a}_i) R(r_{i+1}|\boldsymbol{s}_i, \boldsymbol{a}_i) \tag{4.14}$$

基于梯度的求解器。在 [Man+19] 中，他们结合了 MPO 算法（第 3.4.4 节）进行连续控制，与 **不确定性集** 对动力学进行学习，以学习一个优化最坏情况期望回报目标的策略。

4.2.3.2 端到端可微分的模型和 PL 学习 anner

解决模型拟合与策略学习不匹配问题的方法之一是使用**可微规划**，其中我们学习模型以最小化规划损失。这个双层优化问题首先在**价值迭代网络**论文中提出，由 [Tam+16] 提出，并在 **TreeQN** 论文中扩展，由 [Far+18] 提出。在 [AY20] 中，他们提出了基于可微交叉熵方法的连续动作版本。在 [Nik+22； Ban+23] 中，他们提出使用隐式微分来避免显式展开内部优化。

4.2.3.3 统一模型和规划变分下界

在 [Eys+22] 中，他们提出了一种称为 **Mismatched No More** (MNM) 的方法来解决目标不匹配问题。他们定义了一个基于整个轨迹的优化变量（见第 1.5 节），$p(O=1|\tau) = R(\tau) = \sum_{t=1}^{\infty} \gamma^t R(s_t, a_t)$。这导致了以下关于优化对数概率的下界变分：

$$\log p(O=1) = \log \int_{\tau} P(O=1,\tau) = \log \mathbb{E}_{P(\tau)}\left[P(O=1|\tau)\right] \geq \mathbb{E}_{Q(\tau)}\left[\log R(\tau) + \log P(\tau) - \log Q(\tau)\right]$$

$P(\tau)$ 是应用于真实世界模型的策略所诱导的轨迹分布，$P(\tau) = \mu(s_0) \prod_{t=0}^{\infty} M(s_{t+1}|s_t, a_t)\pi(a_t|s_t)$，以及 $Q(\tau)$ 是使用估计的世界模型的轨迹分布，$Q(\tau) = \mu(s_0) \prod_{t=0}^{\infty} \hat{M}(s_{t+1}|s_t, a_t)\pi(a_t|s_t)$。然后他们最大化这个界限相对于 π 和 $\hat{M}$。

在 [Ghu+22] 中，他们通过学习潜在编码器 $\hat{E}(\boldsymbol{z}_t|\boldsymbol{o}_t)$ 以及潜在动力学 $\hat{M}(\boldsymbol{z}_{t+1}|\boldsymbol{z}_t, a_t)$，将 MNM 扩展到可以处理图像（以及其他高维状态），类似于其他自预测方法（第 4.3.2.2 节）。他们将这种方法称为 **对齐潜在模型**。

4.2.3.4 动态切换 MFRL 和 MBRL

上述方法的一个问题是，如果模型容量有限，或者如果模型学习到环境中的 “ 无关 ” 方面，那么任何 MB RL 方法都可能被直接优化真实期望奖励的 MFRL 方法所支配。一种更安全的方法是在代理有信心模型更好时才使用基于模型的策略，否则退回到无模型策略。这是 Fre+24]. 提出的统一 RL 方法 [的策略。

4.3 世界模型

在这个部分，我们讨论了文献中提出的各种世界模型。这些模型可用于决策时规划或背景规划

4.3.1 生成式世界模型

在这一节中，我们讨论了不同种类的世界模型 $M(\boldsymbol{s}'|\boldsymbol{s}, a)$。我们可以通过从以下联合分布中进行采样来生成想象中的轨迹：

$$p(\boldsymbol{s}_{t+1:T}, \boldsymbol{r}_{t+1:T}, \boldsymbol{a}_{t:T-1}|\boldsymbol{s}_t) = \prod_{i=t}^{T-1} \pi(\boldsymbol{a}_i|\boldsymbol{s}_i) M(\boldsymbol{s}_{i+1}|\boldsymbol{s}_i, \boldsymbol{a}_i) R(r_{i+1}|\boldsymbol{s}_i, \boldsymbol{a}_i) \tag{4.14}$$

4.3.1.1 Observation-space world models

The simplest approach is to define $M(\boldsymbol{s}'|\boldsymbol{s}, a)$ as a conditional generative model over states. If the state space is high dimensional (e.g., images), we can use standard techniques for image generation such as diffusion (see e.g., the **Diamond** method of [Alo+24]). If the observed states are low-dimensional vectors, such as proprioceptive states, we can use transformers (see e.g., the **Transformer Dynamics Model** of [Sch+23a]).

4.3.1.2 Factored models

In some cases, the dimensions of the state vector $\boldsymbol{s}$ represent distinct variables, and the joint Markov transition matrix $p(\boldsymbol{s}'|\boldsymbol{s}, a)$ has conditional independence properties which can be represented as a sparse graphical model, This is called a **factored MDP** [BDG00].

4.3.1.3 Latent-space world models

In this section, we describe some methods that use latent variables as part of their world model. We let $\boldsymbol{z}_t$ denote the latent (or hidden) state at time t; this can be a discrete or continuous variable (or vector of variables). The generative model has the form of a controlled HMM:

$$p(\boldsymbol{o}_{t+1:T}, \boldsymbol{z}_{t+1:T}, \boldsymbol{r}_{t+1:T}, \boldsymbol{a}_{t:T-1}|\boldsymbol{z}_t) = \prod_{i=t}^{T-1} [\pi(\boldsymbol{a}_i|\boldsymbol{z}_i) M(\boldsymbol{z}_{i+1}|\boldsymbol{z}_i, \boldsymbol{a}_i) R(r_i|\boldsymbol{z}_{i+1}, \boldsymbol{a}_i) D(\boldsymbol{o}_i|\boldsymbol{z}_{i+1})] \tag{4.15}$$

where $p(\boldsymbol{o}_t|\boldsymbol{z}_t) = D(\boldsymbol{o}_t|\boldsymbol{z}_t)$ is the decoder, or likelihood function, and $\pi(\boldsymbol{a}_t|\boldsymbol{z}_t)$ is the policy.

The world model is usually trained by maximizing the marginal likelihood of the observed outputs given an action sequence. (We discuss non-likelihood based loss functions in Section 4.3.2.) Computing the marginal likelihood requires marginalizing over the hidden variables $\boldsymbol{z}_{t+1:T}$. To make this computationally tractable, it is common to use amortized variational inference, in which we train an encoder network, $p(\boldsymbol{z}_t|\boldsymbol{o}_t)$, to approximate the posterior over the latents. Many papers have followed this basic approach, such as the **"world models"** paper [HS18], and the methods we discuss below.

4.3.1.4 Dreamer

In this section, we summarize the approach used in **Dreamer** paper [Haf+20] and its recent extensions, such as DreamerV2 [Haf+21] and DreamerV3 [Haf+23]. These are all based on the background planning approach, in which the policy is trained on imaginary trajectories generated by a latent variable world model. (Note that Dreamer is based on an earlier approach called **PlaNet** [Haf+19], which used MPC instead of background planning.)

In Dreamer, the stochastic dynamic latent variables in Equation (4.15) are replaced by deterministic dynamic latent variables $\boldsymbol{h}_t$, since this makes the model easier to train. (We will see that $\boldsymbol{h}_t$ acts like the posterior over the hidden state at time $t-1$; this is also the prior predictive belief state before we see $\boldsymbol{o}_t$.) A "static" stochastic variable $\boldsymbol{\epsilon}_t$ is now generated for each time step, and acts like a "random effect" in order to help generate the observations, without relying on $\boldsymbol{h}_t$ to store all of the necessary information. (This simplifies the recurrent latent state.) In more detail, Dreamer defines the following functions:[2]

- A hidden dynamics (sequence) model: $\boldsymbol{h}_{t+1} = U(\boldsymbol{h}_t, \boldsymbol{a}_t, \boldsymbol{\epsilon}_t)$
- A latent state prior: $\hat{\boldsymbol{\epsilon}}_t \sim P(\hat{\boldsymbol{\epsilon}}_t|\boldsymbol{h}_t)$
- A latent state decoder (observation predictor): $\hat{\boldsymbol{o}}_t \sim D(\hat{\boldsymbol{o}}_t|\boldsymbol{h}_t, \hat{\boldsymbol{\epsilon}}_t)$.
- A reward predictor: $\hat{r}_t \sim R(\hat{r}_t|\boldsymbol{h}_t, \hat{\boldsymbol{\epsilon}}_t)$
- A latent state encoder: $\boldsymbol{\epsilon}_t \sim E(\boldsymbol{\epsilon}_t|\boldsymbol{h}_t, \boldsymbol{o}_t)$.

[2] To map from our notation to the notation in the paper, see the following key: $\boldsymbol{o}_t \to x_t$, $U \to f_\phi$ (sequence model), $P \to p_\phi(\hat{z}_t|h_t)$ (dynamics predictor),ion model), $D \to p_\phi(\hat{x}_t|h_t, \hat{z}_t)$ (decoder), $E \to q_\phi(\boldsymbol{\epsilon}_t|h_t, x_t)$ (encoder).

4.3.1.1 观察空间世界模型

最简单的方法是将 $M(\boldsymbol{s}'|\boldsymbol{s},a)$ 定义为对状态的条件生成模型。如果状态空间是高维的（例如，图像），我们可以使用图像生成的标准技术，如扩散（例如，参见 **Diamond** 方法 [Alo+24]）。如果观察到的状态是低维向量，例如本体感觉状态，我们可以使用变压器（例如，参见 **Transformer Dynamics Model** of [Sch+23a]）。

4.3.1.2 分解模型

在某些情况下，状态向量的维度 $\boldsymbol{s}$ 代表不同的变量，联合马尔可夫转移矩阵 p（$\boldsymbol{s}'|\boldsymbol{s}$，$a$）具有条件独立性属性，这可以表示为一个稀疏图模型，这被称为 **分解的 MDP** [BDG00]。

4.3.1.3 潜在空间世界模型

在这一节中，我们描述了一些将潜在变量作为其世界模型一部分的方法。我们让 $\boldsymbol{z}_t$ 表示时间 t 处的潜在（或隐藏）状态；这可以是一个离散或连续变量（或变量的向量）。生成模型的形式为受控 HMM：

$$p(\boldsymbol{o}_{t+1:T},\boldsymbol{z}_{t+1:T},\boldsymbol{r}_{t+1:T},\boldsymbol{a}_{t:T-1}|\boldsymbol{z}_t)=\prod_{i=t}^{T-1}[\pi(\boldsymbol{a}_i|\boldsymbol{z}_i)M(\boldsymbol{z}_{i+1}|\boldsymbol{z}_i,\boldsymbol{a}_i)R(r_i|\boldsymbol{z}_{i+1},\boldsymbol{a}_i)D(\boldsymbol{o}_i|\boldsymbol{z}_{i+1})] \tag{4.15}$$

where $p(\boldsymbol{o}_t|\boldsymbol{z}_t)=D(\boldsymbol{o}_t|\boldsymbol{z}_t)$ 是解码器，或似然函数，而 $\pi(\boldsymbol{a}_t|\boldsymbol{z}_t)$ 是 policy.

世界模型通常通过最大化给定动作序列的观察输出的边际似然来训练。（我们在第 4.3.2 节讨论了非似然损失函数。）计算边际似然需要对隐藏变量 $\boldsymbol{z}_{t+1:T}$ 进行边缘化。为了使计算变得可行，通常使用摊销变分推理，其中我们训练一个编码网络，$p(\boldsymbol{z}_t|\boldsymbol{o}_t)$，来近似对潜变量的后验。许多论文都遵循了这种方法，例如 “**世界模型**” 论文 [HS18]，以及我们下面讨论的方法。

4.3.1.4 梦想家

在本节中，我们总结了在 **Dreamer** 论文 [Haf+20] 及其最近扩展中使用的方法，例如 DreamerV2 [Haf+21] 和 DreamerV3 [Haf+23]。这些都是基于背景规划方法，其中策略是在由潜在变量世界模型生成的虚拟轨迹上训练的。（注意，Dreamer 基于一个更早的方法，即 **PlaNet** [Haf+19]，它使用 MPC 而不是背景规划。）

在 Dreamer 中，方程（4.15）中的随机动态潜在变量被确定性动态潜在变量 $\boldsymbol{h}_t$ 所取代，因为这使得模型更容易训练。（我们将看到 $\boldsymbol{h}_t$ 在时间 $t-1$ 的隐藏状态上的作用类似于后验；这在我们看到 $\boldsymbol{o}_t$ 之前也是先验预测信念状态。）现在为每个时间步生成一个 “静态” 随机变量 $\boldsymbol{\epsilon}_t$，它充当 “随机效应”，以帮助生成观测值，而不依赖于 $\boldsymbol{h}_t$ 来存储所有必要的信息。（这简化了循环潜在状态。）更详细地说，Dreamer 定义了以下函数：[2]

- 一个隐藏动力学（序列）模型：$\boldsymbol{h}_{t+1}=U(\boldsymbol{h}_t,\boldsymbol{a}_t,\boldsymbol{\epsilon}_t)$
- 一个潜在状态先验：$\hat{\boldsymbol{\epsilon}}_t\sim P(\hat{\boldsymbol{\epsilon}}_t|\boldsymbol{h}_t)$
- 一个潜在状态解码器（观测预测器）：$\hat{\boldsymbol{o}}_t\sim D(\hat{\boldsymbol{o}}_t|\boldsymbol{h}_t,\hat{\boldsymbol{\epsilon}}_t)$。
- 一个奖励预测器：$\hat{r}_t\sim R(\hat{r}_t|\boldsymbol{h}_t,\hat{\boldsymbol{\epsilon}}_t)$
- 一个潜在状态编码器：$\boldsymbol{\epsilon}_t\sim E(\boldsymbol{\epsilon}_t|\boldsymbol{h}_t,\boldsymbol{o}_t)$。

[2] 将我们的符号映射到论文中的符号，请参阅以下键：$\boldsymbol{o}_t\to x_t$，$U\to f_\phi$（序列模型），$P\to p_\phi(\hat{z}_t|h_t)$（动力学预测器），ion模型），$D\to p_\phi(\hat{x}_t|h_t,\ \hat{z}_t)$（解码器），$E\to q_\phi$（$\boldsymbol{\epsilon}_t|h_t,\ x_t$）（编码器）。

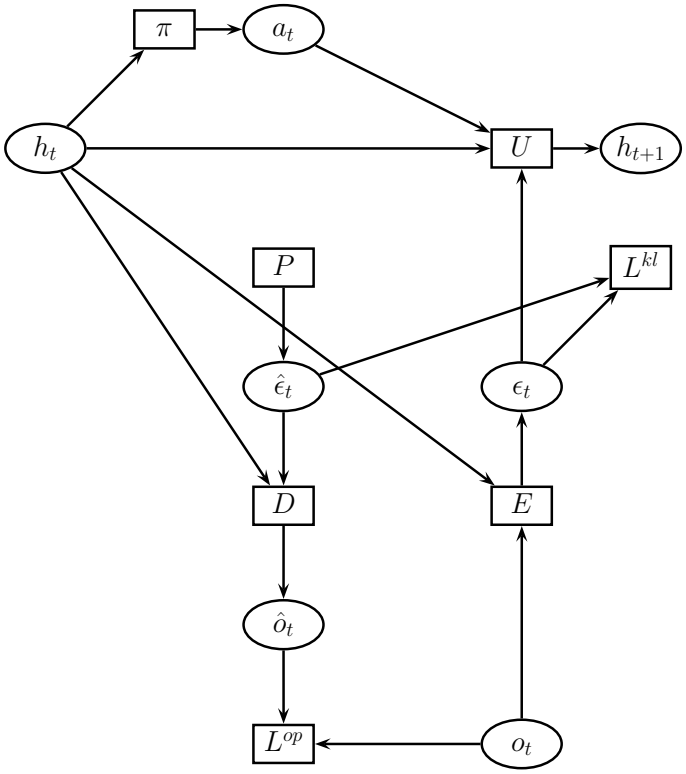


Figure 4.2: Illustration of Dreamer world model as a factor graph (so squares are functions, circles are variables). We have unrolled the forwards prediction for only 1 step. Also, we have omitted the reward prediction loss.

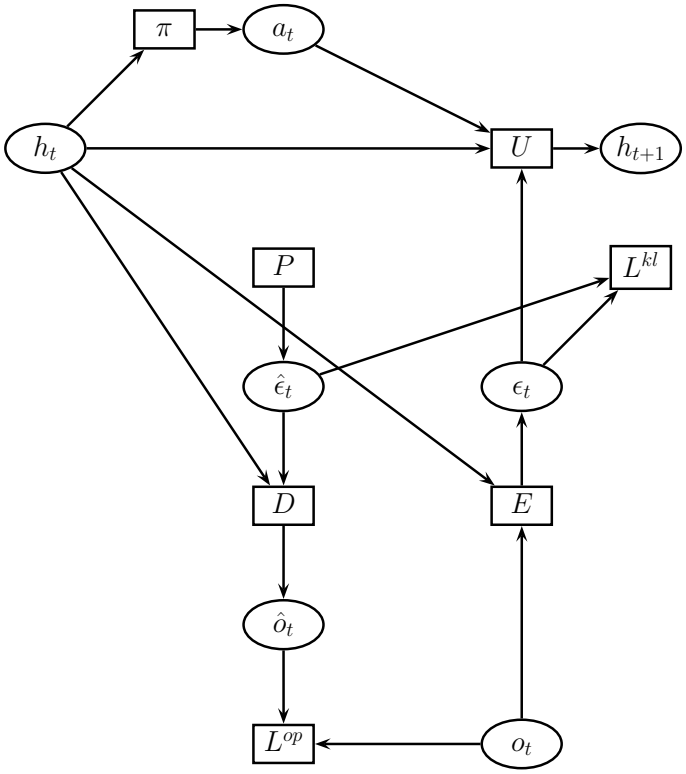


图 4.2：将 Dreamer 世界模型作为因子图进行说明（因此正方形是函数，圆形是变量）。我们只展开前向预测了 1 步。此外，我们还省略了奖励预测损失。

- A policy function: $\boldsymbol{a}_t \sim \pi(\boldsymbol{a}_t|\boldsymbol{h}_t)$

See Figure 4.2 for an illustration of the system.

We now give a simplified explanation of how the world model is trained. The loss has the form

$$\mathcal{L}^{\text{WM}} = \mathbb{E}_{q(\boldsymbol{\epsilon}_{1:T})}\left[\sum_{t=1}^{T} \beta_o \mathcal{L}^o(\boldsymbol{o}_t, \hat{\boldsymbol{o}}_t) + \beta_z \mathcal{L}^z(\boldsymbol{\epsilon}_t, \hat{\boldsymbol{\epsilon}}_t)\right] \tag{4.16}$$

where the β terms are different weights for each loss, and q is the posterior over the latents, given by

$$q(\boldsymbol{\epsilon}_{1:T}|\boldsymbol{h}_0, \boldsymbol{o}_{1:T}, \boldsymbol{a}_{1:T}) = \prod_{t=1}^{T} E(\boldsymbol{\epsilon}_t|\boldsymbol{h}_t, \boldsymbol{o}_t)\delta(\boldsymbol{h}_t - U(\boldsymbol{h}_{t-1}, \boldsymbol{a}_{t-1}, \boldsymbol{\epsilon}_{t-1})) \tag{4.17}$$

The loss terms are defined as follows:

$$\mathcal{L}^o = -\ln D(\boldsymbol{o}_t|\boldsymbol{\epsilon}_t, \boldsymbol{h}_t) \tag{4.18}$$

$$\mathcal{L}^z = D_{\mathbb{KL}}\left(E(\boldsymbol{\epsilon}_t|\boldsymbol{h}_t, \boldsymbol{o}_t)) \parallel P(\boldsymbol{\epsilon}_t|\boldsymbol{h}_t))\right) \tag{4.19}$$

where we abuse notation somewhat, since $\mathcal{L}^z$ is a function of two distributions, not of the variables $\boldsymbol{\epsilon}_t$ and $\hat{\boldsymbol{\epsilon}}_t$.

In addition to the world model loss, we have the following actor-critic losses

$$\mathcal{L}^{\text{critic}} = \sum_{t=1}^{T}(V(\boldsymbol{h}_t) - \text{sg}(G_t^\lambda))^2 \tag{4.20}$$

$$\mathcal{L}^{\text{actor}} = -\sum_{t=1}^{T} \text{sg}((G_t^\lambda - V(\boldsymbol{h}_t))) \log \pi(\boldsymbol{a}_t|\boldsymbol{h}_t) \tag{4.21}$$

where G_t^λ is the GAE estimate of the reward to go:

$$G_t^\lambda = r_t + \gamma\left((1-\lambda)V(\boldsymbol{h}_t) + \lambda G_{t+1}^\lambda\right) \tag{4.22}$$

There have been several extensions to the original Dreamer paper. DreamerV2 [Haf+21] adds categorical (discrete) latents and KL balancing between prior and posterior estimates. This was the first imagination-based agent to outperform humans in Atari games. DayDreamer [Wu+22] applies DreamerV2 to real robots. DreamerV3 [Haf+23] builds upon DreamerV2 using various tricks — such as symlog encodings[3] for the reward, critic, and decoder — to enable more stable optimization and domain independent choice of hyper-parameters. It was the first method to create diamonds in the Minecraft game without requiring human demonstration data. (However, reaching this goal took 17 days of training.) [Lin+24a] extends DreamerV3 to also model language observations.

Variants of Dreamer such as TransDreamer [Che+21a] and STORM [Zha+23b] have also been explored, where transformers replace the recurrent network. The DreamingV2 paper of [OT22] replaces the generative loss with a non-generative self-prediction loss (see Section 4.3.2.2).

4.3.1.5 Iris

The **Iris** method of [MAF22] follows the MBRL paradigm, in which it alternates beween (1) learning a world model using real data D_r and then generate imaginary rollouts D_i using the WM, and (2) learning the policy given D_i and collecting new data D_r' for learning. In the model learning stage, Iris learns a discrete latent encoding using the VQ-VAE method, and then fits a transformer dynamics model to the latent codes. In the policy learning stage, it uses actor critic methods. The **Delta-Iris** method of [MAF24] extends this by training the model to only predict the delta between neighboring frames. Note that, in both cases, the policy has the form $a_t = \pi(\boldsymbol{o}_t)$, where $\boldsymbol{o}_t$ is an image, so the the rollouts need to ground to pixel space, and cannot only be done in latent space.

[3] The symlog function is defined as $\text{symlog}(x) = \text{sign}(x)\ln(|x|+1)$, and its inverse is $\text{symexp}(x) = \text{sign}(x)(\exp(|x|)-1)$. The symlog function squashes large positive and negative values, while preserving small values.

- 策略函数：$\boldsymbol{a}_t \sim \pi(\boldsymbol{a}_t|\boldsymbol{h}_t)$

参见图 4.2 以了解系统的说明。

我们现在给出世界模型训练的简化解释。损失具有形式

$$\mathcal{L}^{\mathrm{WM}} = \mathbb{E}_{q(\boldsymbol{\epsilon}_{1:T})}\left[\sum_{t=1}^{T} \beta_o \mathcal{L}^o(\boldsymbol{o}_t, \hat{\boldsymbol{o}}_t) + \beta_z \mathcal{L}^z(\boldsymbol{\epsilon}_t, \hat{\boldsymbol{\epsilon}}_t)\right] \tag{4.16}$$

where the β terms are 不同的损失有不同的权重，且 q 是后验概率。 the latents, given by

$$q(\boldsymbol{\epsilon}_{1:T}|\boldsymbol{h}_0, \boldsymbol{o}_{1:T}, \boldsymbol{a}_{1:T}) = \prod_{t=1}^{T} E(\boldsymbol{\epsilon}_t|\boldsymbol{h}_t, \boldsymbol{o}_t)\delta(\boldsymbol{h}_t - U(\boldsymbol{h}_{t-1}, \boldsymbol{a}_{t-1}, \boldsymbol{\epsilon}_{t-1})) \tag{4.17}$$

损失项如下定义：

$$\mathcal{L}^o = -\ln D(\boldsymbol{o}_t|\boldsymbol{\epsilon}_t, \boldsymbol{h}_t) \tag{4.18}$$

$$\mathcal{L}^z = D_{\mathbb{KL}}\left(E(\boldsymbol{\epsilon}_t|\boldsymbol{h}_t, \boldsymbol{o}_t)) \parallel P(\boldsymbol{\epsilon}_t|\boldsymbol{h}_t))\right) \tag{4.19}$$

我们在某种程度上滥用符号，因为 $\mathcal{L}^z$ 是两个分布的函数，而不是变量 $\boldsymbol{\epsilon}_t$ 和 $\hat{\boldsymbol{\epsilon}}_t$ 的函数。除了世界模型损失之外，我们还有以下演员 - 评论家损失

$$\mathcal{L}^{\mathrm{critic}} = \sum_{t=1}^{T} (V(\boldsymbol{h}_t) - \mathrm{sg}(G_t^\lambda))^2 \tag{4.20}$$

$$\mathcal{L}^{\mathrm{actor}} = -\sum_{t=1}^{T} \mathrm{sg}((G_t^\lambda - V(\boldsymbol{h}_t))) \log \pi(\boldsymbol{a}_t|\boldsymbol{h}_t) \tag{4.21}$$

G_t^λ 是到达奖励的 GAE 估计：

$$G_t^\lambda = r_t + \gamma\left((1-\lambda)V(\boldsymbol{h}_t) + \lambda G_{t+1}^\lambda\right) \tag{4.22}$$

原始 Dreamer 论文已有几个扩展。DreamerV2 [Haf+21] 增加了分类（离散）潜在变量和先验估计与后验估计之间的 KL 平衡。这是第一个在 Atari 游戏中超越人类的基于想象力的智能体。DayDreamer [Wu+22] 将 DreamerV2 应用于真实机器人。DreamerV3 [Haf+23] 基于 DreamerV2，使用各种技巧 —— 如对称对数编码 [3] 用于奖励、批评家和解码器 —— 以实现更稳定的优化和域无关的超参数选择。这是第一个在 Minecraft 游戏中创建钻石的方法，无需人类演示数据。（然而，达到这个目标需要 17 天的训练。） [Lin+24a] 将 DreamerV3 扩展到也建模语言观察。

梦游者变体，如 TransDreamer[Che+21a] 以及 STORM [Zha+23b] 也已被探索，其中变换器取代了循环网络。OT[的 DreamingV2 论文 22] 用非生成性自预测损失替换了生成损失（见第 4.3.2.2 节）。

4.3.1.5 眼睛

Iris 方法遵循 MBRL 范式，它在以下两个方面交替进行：(1) 使用真实数据学习世界模型，然后使用 WM 生成想象中的 rollouts ； (2) 在给定 D_i 的情况下学习策略，并收集新的数据 D'_r 用于学习。在模型学习阶段，Iris 使用 VQ-VAE 方法学习离散的潜在编码，然后将 transformer 动力学模型拟合到潜在代码。在策略学习阶段，它使用 actor critic 方法。**Delta-Iris** 方法通过训练模型仅预测相邻帧之间的 delta 来扩展这一点。请注意，在这两种情况下，策略的形式为 $a_t = \pi(\boldsymbol{o}_t)$，其中 $\boldsymbol{o}_t$ 是图像，因此 rollouts 需要映射到像素空间，而不仅仅是潜在空间。

[3] 对称对数函数定义为 $\mathrm{symlog}(x) = \mathrm{sign}(x)\ln(|x|+1)$，其逆函数为 $\mathrm{symexp}(x) = \mathrm{sign}(x)(\exp(|x|)-1)$。对称对数函数将大正数和大负数压缩，同时保留小数值。

Loss	Policy	Usage	Examples
OP	Observables	Dyna	Diamond [Alo+24], Delta-Iris [MAF24]
OP	Observables	MCTS	TDM [Sch+23a]
OP	Latents	Dyna	Dreamer [Haf+23]
RP, VP, PP	Latents	MCTS	MuZero [Sch+20]
RP, VP, PP, ZP	Latents	MCTS	EfficientZero [Ye+21]
RP, VP, ZP	Latents	MPC-CEM	TD-MPC [HSW22b]
VP, ZP	Latents	Aux.	Minimalist [Ni+24]
VP, ZP	Latents	Dyna	DreamingV2 [OT22]
VP, ZP, OP	Latents	Dyna	AIS [Sub+22]
POP	Latents	Dyna	Denoised MDP [Wan+22]

Table 4.1: Summary of some world-modeling methods. The "loss" column refers to the loss used to train the latent encoder (if present) and the dynamics model (OP = observation prediction, ZP = latent state prediction, RP = reward prediction, VP = value prediction, PP = policy prediction, POP = partial observation prediction). The "policy" column refers to the input that is passed to the policy. (For MCTS methods, the policy is just used as a proposal over action sequences to initialize the search/ optimization process.) The "usage" column refers to how to the world model is used: for background planning (which we call "Dyna"), or for decision-time planning (which we call "MCTS"), or just as an auxiliary loss on top of standard policy/value learning (which we call "Aux"). Thus Aux methods are single-stage ("end-to-end"), whereas the other methods alternate are two-phase, and alternate between improving the world model and then using it for improving the policy (or searching for the optimal action).

4.3.2 Non-generative world models

In Section 4.2.1, we argued that, if we can learn a sufficiently accurate world model, then solving for the optimal policy in simulation will give a policy that is close to optimal in the real world. However, a simple agent may not be able to capture the full complexity of the true environment; this is called the "**small agent, big world**" problem [DVRZ22; Lu+23; Aru+24a; Kum+24].

Consider what happens when the agent's model is misspecified (i.e., it cannot represent the true world model), which is nearly always the case. The agent will train its model to reduce state (or observation) prediction error, by minimizing $\ell(\hat{M}, \mu_M^\pi)$. However, not all features of the state are useful for planning. For example, if the states are images, a dynamics model with limited representational capacity may choose to focus on predicting the background pixels rather than more control-relevant features, like small moving objects, since predicting the background reliably reduces the MSE more. This is due to "**objective mismatch**" [Lam+20; Wei+24], which refers to the discrepancy between the way a model is usually trained (to predict the observations) vs the way its representation is used for control. To tackle this problem, in this section we discuss methods for learning representations and models that don't rely on predicting all the observations. Our presentation is based in part on [Ni+24] (which in turn builds on [Sub+22]). See Table 4.1 for a summary of some of the methods we will discuss.

4.3.2.1 Value prediction

Let $\mathcal{D}_t = (\mathcal{D}_{t-1}, \boldsymbol{a}_{t-1}, r_{t-1}, \boldsymbol{o}_t)$ be the observed history at time t, and let $\boldsymbol{z}_t = \phi(\mathcal{D}_t)$ be a latent representation (compressed encoding) of this history, where ϕ is called an encoder or a **state abstraction** function. We will train the policy $\boldsymbol{a}_t = \pi(\boldsymbol{z}_t)$ in the usual way, so our focus will be on how to learn good latent representations.

An optimal representation $\boldsymbol{z}_t = \phi(\mathcal{D}_t)$ is a sufficient statistic for the optimal action-value function Q^*. Thus it satifies the **value equivalence** principle [LWL06; Cas11; Gri+20; GBS22; AP23; ARKP24], which says that two states s_1 and s_2 are value equivalent (given a policy) if $V^\pi(s_1) = V^\pi(s_2)$. In particular, if the representation is optimal, it will satisfy value equivalence wrt the optimal policy, i.e., if $\phi(\mathcal{D}_i) = \phi(\mathcal{D}_j)$ then $Q^*(\mathcal{D}_i, a) = Q^*(\mathcal{D}_j, a)$. We can train such a representation function by using its output $\boldsymbol{z} = \phi(\mathcal{D})$ as input to the Q function or to the policy. (We call such a loss **VP**, for value prediction.) This will cause the model to focus its representational power on the relevant parts of the observation history.

Note that there is a stronger property than value equivalence called **bisimulation** [GDG03]. This says

Loss	Policy	Usage	Examples
OP	Observables	Dyna	Diamond [Alo+24], Delta-Iris [MAF24]
OP	Observables	MCTS	TDM [Sch+23a]
OP	Latents	Dyna	Dreamer [Haf+23]
RP, VP, PP	Latents	MCTS	MuZero [Sch+20]
RP, VP, PP, ZP	Latents	MCTS	EfficientZero [Ye+21]
RP, VP, ZP	Latents	MPC-CEM	TD-MPC [HSW22b]
VP, ZP	Latents	Aux.	Minimalist [Ni+24]
VP, ZP	Latents	Dyna	DreamingV2 [OT22]
VP, ZP, OP	Latents	Dyna	AIS [Sub+22]
POP	Latents	Dyna	Denoised MDP [Wan+22]

表 4.1：一些世界建模方法的总结。“损失”列指的是用于训练潜在编码器（如果存在）和动力学模型（OP = 观察预测，ZP = 潜在状态预测，RP = 奖励预测，VP= 价值预测，PP = 策略预测，POP = 部分观察预测）所使用的损失。“策略”列指的是传递给策略的输入。（对于 MCTS 方法，策略仅用作动作序列的提议，以初始化搜索 / 优化过程。）“使用”列指的是如何使用世界模型：用于背景规划（我们称之为“Dyna”），或用于决策时规划（我们称之为“MCTS”），或仅作为标准策略 / 价值学习之上的辅助损失（我们称之为“Aux”）。因此，Aux 方法是一阶段的（“端到端”），而其他方法则是两阶段的，交替在改进世界模型和使用它来改进策略（或搜索最优动作）之间。

4.3.2 非生成式世界模型

在章节 4.2.1 中，我们论证了，如果我们能够学习到一个足够准确的世界模型，那么在模拟中求解最优策略将给出一个在现实世界中接近最优的策略。然而，一个简单的智能体可能无法捕捉到真实环境的全部复杂性；这被称为“**小智能体，大世界**”问题 [DVRZ22； Lu+23； Aru+24a； Kum+24]。

考虑当代理的模型被误指定（即，它不能表示真实世界模型）时会发生什么，这几乎是始终如此的情况。代理将通过最小化 ℓ（$\hat{M}$，μ_M^π）来训练其模型以减少状态（或观察）预测误差。然而，并非所有状态的特征都对规划有用。例如，如果状态是图像，具有有限表示能力的动力学模型可能会选择专注于预测背景像素，而不是更相关的控制特征，如小移动物体，因为可靠地预测背景可以减少均方误差。这是由于“**目标不匹配**”[Lam+20； Wei+24]”，这指的是模型通常被训练（以预测观察结果）与其表示用于控制的方式之间的差异。为了解决这个问题，在本节中，我们讨论了学习表示和模型的方法，这些方法不依赖于预测所有观察结果。我们的介绍部分基于 [Ni+24]（它反过来又基于 [Sub+22]）。有关我们将讨论的一些方法的总结，请参阅表 4.1 。

4.3.2.1 值预测

让 $\mathcal{D}_t = (\mathcal{D}_{t-1}, \boldsymbol{a}_{t-1}, r_{t-1}, \boldsymbol{o}_t)$ 成为时间 t 观察到的历史，让 $\boldsymbol{z}_t = \phi(\mathcal{D}_t)$ 成为该历史的潜在表示（压缩编码），其中 ϕ 被称为编码器或状态抽象 函数。我们将以通常的方式训练策略 $\boldsymbol{a}_t = \pi(\boldsymbol{z}_t)$，因此我们的重点将是如何学习良好的潜在表示。

一个最优表示 $\boldsymbol{z}_t = \phi(\mathcal{D}_t)$ 是最佳动作值函数 Q^* 的充分统计量。因此，它满足 **价值等价** 原则 [LWL06； Cas11； Gri+20； GBS22； AP23； ARKP24]，，该原则指出，如果两个状态 s_1 和 s_2 （给定一个策略）价值等价，则 $V^\pi(s_1) = V^\pi(s_2)$。特别是，如果表示是最佳的，它将满足与最佳策略的价值等价，即如果 $\phi(\mathcal{D}_i) = \phi(\mathcal{D}_j)$ 那么 $Q^*(\mathcal{D}_i, a) = Q^*(\mathcal{D}_j, a)$。我们可以通过使用其输出 $\boldsymbol{z} = \phi(\mathcal{D})$ 作为 Q 函数或策略的输入来训练这样的表示函数。（我们称这种损失为 **VP**，即价值预测。）这将导致模型将它的表示能力集中在观察历史的相关部分。注意，有一个更强的正确比值等价称为 **bisimulation** [GDG03] 这表示

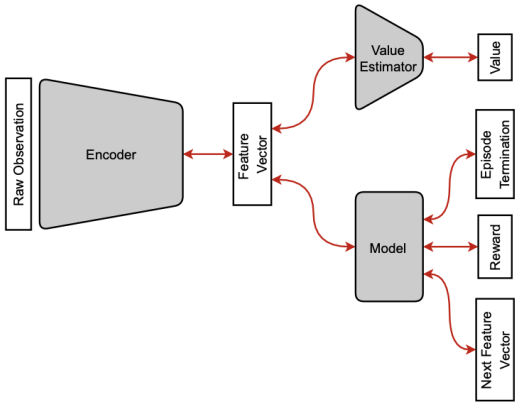


Figure 4.3: Illustration of an encoder $\boldsymbol{z}_t = E(\boldsymbol{o}_t)$, which is passed to a value estimator $v_t = V(\boldsymbol{z}_t)$, and a world model, which predicts the next latent state $\hat{\boldsymbol{z}}_{t+1} = M(\boldsymbol{z}_t, a_t)$, the reward $r_t = R(\boldsymbol{z}_t, a_t)$, and the termination (done) flag, $d_t = done(\boldsymbol{z}_t)$. From Figure C.2 of [AP23]. Used with kind permission of Doina Precup.

that two states s_1 and s_2 are bisimiliar if $P(s'|s_1, a) \approx P(s'|s_2, a)$ and $R(s_1, a) = R(s_2, a)$. From this, we can derive a continuous measure called the **bisimulation metric** [FPP04]. This has the advantage (compared to value equivalence) of being policy independent, but the disadvantage that it can be harder to compute [Cas20; Zha+21], although there has been recent progress on computaitonally efficient methods such as MICo [Cas+21] and KSMe [Cas+23].

4.3.2.2 Self prediction

Unfortunately, in problems with sparse reward, predicting the value may not provide enough of a feedback signal to learn quickly. Consequently it is common to augment the training with a **self-prediction** loss where we train ϕ to ensure the following condition hold:

$$\exists M \quad \text{s.t.} \quad \mathbb{E}_{M^*}\left[\boldsymbol{z}'|\mathcal{D}, a\right] = \mathbb{E}_{M}\left[\boldsymbol{z}'|\phi(\mathcal{D}), a)\right] \quad \forall \mathcal{D}, a \tag{4.23}$$

where the LHS is the predicted mean of the next latent state under the true model, and the RHS is the predicted mean under the learned dynamics model. We call this the **EZP**, which stands for expected $\boldsymbol{z}$ prediction.[4]

A trivial way to minimize the (E)ZP loss is for the embedding to map everything to a constant vector, say $E(\mathcal{D}) = \mathbf{0}$, in which case $\boldsymbol{z}_{t+1}$ will be trivial for the dynamics model M to predict. However this is not a useful representation. This problem is **representational collapse** [Jin+22]. Fortunately, we can provably prevent collapse (at least for linear encoders) by using a frozen target network [Tan+23; Ni+24]. That is, we use the following auxiliary loss

$$\mathcal{L}_{\text{EZP}}(\boldsymbol{\phi}, \boldsymbol{\theta}; \mathcal{D}, a, \mathcal{D}') = ||M_{\boldsymbol{\theta}}(E_{\boldsymbol{\phi}}(\mathcal{D}, a)) - E_{\overline{\boldsymbol{\phi}}}(\mathcal{D}')||_2^2 \tag{4.24}$$

where

$$\overline{\boldsymbol{\phi}} = \rho\overline{\boldsymbol{\phi}} + (1-\rho)\text{sg}(\boldsymbol{\phi}) \tag{4.25}$$

is the (stop-gradient version of) the EMA of the encoder weights. (If we set $\rho = 0$, this is called a detached network.)

[4] In [Ni+24], they also describe the ZP loss, which requires predicting the full distribution over $\boldsymbol{z}'$ using a stochastic transition model. This is strictly more powerful, but somewhat more complicated, so we omit it for simplicity.

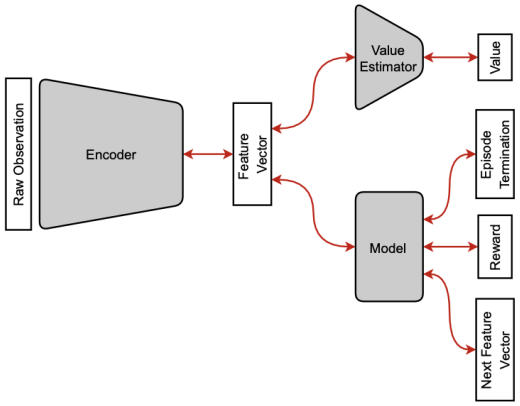


图 4.3：编码器 $\boldsymbol{z}_t = E(\boldsymbol{o}_t)$ 的示意图，该编码器传递给值估计器 $v_t = V(\boldsymbol{z}_t)$ 和一个世界模型，该模型预测下一个潜在状态 $\hat{\boldsymbol{z}}_{t+1} = M(\boldsymbol{z}_t, a_t)$，奖励 $r_t = R(\boldsymbol{z}_t, a_t)$ 和终止（完成）标志 $d_t = \text{完成}(\boldsymbol{z}_t)$。来自 *[AP23]* 的图 C.2。经 Doina Precup 许可使用。

两个状态 s_1 和 s_2 是双相似当且仅当 $P(s'|s_1, a) \approx P(s'|s_2, a)$ 和 $R(s_1, a) = R(s_2, a)$。由此，我们可以导出一个连续度量，称为 **双相似度** [FPP04]。与值等价相比，它具有政策无关的优点，但缺点是计算起来可能更困难 [Cas20； Zha+21]，尽管最近在计算效率高的方法上取得了进展，例如 MICo[Cas+21] 和 KSMe [Cas+23]。

4.3.2.2 自我预测

不幸的是，在稀疏奖励的问题中，预测价值可能无法提供足够的反馈信号以快速学习。因此，通常会在训练中增加一个 **自我预测**损失，我们训练 ϕ 以确保以下条件成立：

$$\exists M \quad \text{s.t.} \quad \mathbb{E}_{M^*}\left[\boldsymbol{z}'|\mathcal{D}, a\right] = \mathbb{E}_{M}\left[\boldsymbol{z}'|\phi(\mathcal{D}), a)\right] \quad \forall \mathcal{D}, a \tag{4.23}$$

LHS 是真实模型下下一个潜在状态的预测均值，RHS 是学习到的动力学模型下的预测均值。我们称这为 **EZP**，代表期望 $\boldsymbol{z}$ 预测 [4]。

将嵌入映射到常量向量是一种最小化（E） ZP 损失的平凡方法，例如 $E(\mathcal{D}) = \mathbf{0}$，在这种情况下，$\boldsymbol{z}_{t+1}$ 对于动态模型 M 的预测将是平凡的。然而，这并不是一个有用的表示。这个问题是**表示崩溃** [金 +22]。幸运的是，我们可以通过使用冻结的目标网络 [谭 +23； 倪 +24] 来证明地防止崩溃（至少对于线性编码器）。也就是说，我们使用以下辅助损失

$$\mathcal{L}_{\text{EZP}}(\boldsymbol{\phi}, \boldsymbol{\theta}; \mathcal{D}, a, \mathcal{D}') = ||M_{\boldsymbol{\theta}}(E_{\boldsymbol{\phi}}(\mathcal{D}, a)) - E_{\boldsymbol{\phi}}(\mathcal{D}')||_2^2 \tag{4.24}$$

在哪里

$$\boldsymbol{\phi} = \rho\boldsymbol{\phi} + (1 - \rho)\text{sg}(\boldsymbol{\phi}) \tag{4.25}$$

是编码器权重的 EMA （指数移动平均）的（停止梯度版本）。（如果我们设置 $\rho = 0$，这被称为分离网络。）

[4]在 [Ni+24]，他们也描述了ZP损失，这需要使用随机转换模型预测 $\boldsymbol{z}'$ 的全分布。这更强大，但稍微复杂一些，所以我们为了简单起见省略了它。

We can also train the latent encoder to predict the reward. Formally, we want to ensure we can satisfy the following condition, which we call **RP** for "reward prediction":

$$\exists R \;\; \text{s.t.} \;\; \mathbb{E}_{R^*}\left[r|\mathcal{D}, a\right] = \mathbb{E}_R\left[r|\phi(\mathcal{D}), a)\right] \;\; \forall \mathcal{D}, a \tag{4.26}$$

See Figure 4.3 for an illustration. In [Ni+24], they prove that a representation that satisfies ZP and RP is enough to satisfy value equivalence (sufficiency for Q^*).

Methods that optimize ZP and VP loss have been used in many papers, such as **Predictron** [Sil+17b], **Value Prediction Networks** [OSL17], **Self Predictive Representations** (SPR) [Sch+21], **Efficient Zero** (Section 4.1.3.3), **BYOL-Explore** (Section 4.3.2.6), etc.

4.3.2.3 Policy prediction

The value function and reward losses may be too sparse to learn efficiently. Although self-prediction loss can help somewhat, it does not use any extra information from the environment as feedback. Consequently it is natural to consider other kinds of prediction targets for learning the latent encoder (and dynamics). When using MCTS, it is possible compute what the policy should be for a given state, and this can be used as a prediction target for the reactive policy $a_t = \pi(\boldsymbol{z}_t)$, which in turn can be used as a feedback signal for the latent state. This method is used by MuZero (Section 4.1.3.2) and EfficientZero (Section 4.1.3.3).

4.3.2.4 Observation prediction

Another natural target to use for learning the encoder and dynamics is the next observation, using a one-step version of Equation (4.14). Indeed, [Ni+24] say that a representation ϕ satsifies the **OP** (observation prediction) criterion if it satisfies the following condition:

$$\exists D \;\; \text{s.t.} \;\; p^*(\boldsymbol{o}'|\mathcal{D}, a) = D(\boldsymbol{o}'|\phi(\mathcal{D}), a) \;\; \forall \mathcal{D}, a \tag{4.27}$$

where D is the decoder. In order to repeatedly apply this, we need to be able to update the encoding $\boldsymbol{z} = \phi(\mathcal{D})$ in a recursive or online way. Thus we must also satisfify the following recurrent encoder condition, which [Ni+24] call **Rec**:

$$\exists U \;\; \text{s.t.} \;\; \phi(\mathcal{D}') = U(\phi(\mathcal{D}), a, \boldsymbol{o}') \;\; \forall \mathcal{D}, a, \boldsymbol{o}' \tag{4.28}$$

where U is the update operator. Note that belief state updates (as in a POMDP) satisfy this property. Furthermore, belief states are a sufficient statistic to satisfy the OP condition. See Section 4.3.1.3 for a discussion of generative models of this form. However, there are other approaches to partial observability which work directly in prediction space (see Section 4.4.2).

4.3.2.5 Partial observation prediction

We have argued that predicting all the observations is problematic, but not predicting them is also problematic. A natural compromise is to predict some of the observations, or at least sone function of them. This is known as a **partial world model** (see e.g., [AP23]).

The best way to do this is an open research problem. A simple approach would be to predict all the observations, but put a penalty on the resulting OP loss term. A more sophisticated approach would be to structure the latent space so that we distinguish latent variables that are useful for learning Q^* (i.e., which affect the reward and which are affected by the agent's actions) from other latent variables that are needed to explain parts of the observation but otherwise are not useful. We can then impose an information bottleneck penalty on the latter, to prevent the agent focusing on irrelevant observational details. (See e.g., the **denoised MDP** method of [Wan+22].)

我们还可以训练潜在编码器来预测奖励。正式来说，我们希望确保能够满足以下条件，我们称之为 “ 奖励预测 ” 条件：**RP**

$$\exists R \text{ s.t. } \mathbb{E}_{R^*}\left[r|\mathcal{D},a\right] = \mathbb{E}_{R}\left[r|\phi(\mathcal{D}),a\right)] \ \ \forall \mathcal{D},a \tag{4.26}$$

参见图 4.3 以了解说明。在 [Ni+24] 中，他们证明满足 ZP 和 RP 的表示足以满足值等价性（对于 Q^* 的充分性）。

优化 ZP 和 VP 损失的方法已在许多论文中使用，例如 **Predictron** [Sil+17b],**Value Prediction Networks** [OSL17], **Self Predictive Representations**(SPR) [Sch+21], **EfficientZero** (第 4.1.3.3) 节，**BYOL-Explore** (第 4.3.2.6) 节等。

4.3.2.3 政策预测

值函数和奖励损失可能过于稀疏，难以高效学习。尽管自预测损失有所帮助，但它没有使用环境中的任何额外信息作为反馈。因此，考虑其他类型的预测目标来学习潜在编码器（和动力学）是自然的。在使用 MCTS 时，可以计算给定状态的政策应该是什么，这可以用作反应性策略 $a_t = \pi(\boldsymbol{z}_t)$ 的预测目标，反过来，这可以用来作为潜在状态的反馈信号。这种方法被 MuZero （第 4.1.3.2 节）和 EfficientZero （第 4.1.3.3 节）所采用。

4.3.2.4 观测预测

另一个用于学习编码器和动态的自然目标是下一个观测值，使用方程（ 4.14 ）的一步版本。事实上，[Ni+24] 表示，如果一个表示 ϕ 满足以下条件，则它满足 **OP** （观测预测）标准：

$$\exists D \text{ s.t. } p^*(\boldsymbol{o}'|\mathcal{D},a) = D(\boldsymbol{o}'|\phi(\mathcal{D}),a) \ \ \forall \mathcal{D},a \tag{4.27}$$

其中 D 是解码器。为了重复应用，我们需要能够以递归或在线方式更新编码 $\boldsymbol{z} = \phi(\mathcal{D})$。因此，我们还必须满足以下循环编码条件，我们称之为 [Ni+24]Rec：

$$\exists U \text{ s.t. } \phi(\mathcal{D}') = U(\phi(\mathcal{D}),a,\boldsymbol{o}') \ \ \forall \mathcal{D},a,\boldsymbol{o}' \tag{4.28}$$

其中 U 是更新算子。请注意，信念状态更新（如 POMDP 中的）满足此性质。此外，信念状态是满足 OP 条件的充分统计量。参见第 4.3.1.3 节 ，讨论此类生成模型。然而，还有其他处理部分可观测性的方法，这些方法直接在预测空间中工作（参见第 4.4.2 节）。

4.3.2.5 部分观测预测

我们曾提出，预测所有观测值是有问题的，但完全不预测它们也是有问题的。一个自然的折衷方案是预测一些观测值，或者至少是它们的一些函数。这被称为部分世界模型（例如，参见 AP ）。

这是最好的开放研究问题。一个简单的方法是预测所有观察结果，但对结果中的 OP 损失项进行惩罚。一个更复杂的方法是结构化潜在空间，以便我们区分对学习有用的潜在变量（即影响奖励并被代理的动作影响的变量）与其他潜在变量，后者需要解释观察结果的一部分，但除此之外没有其他用途。然后我们可以对后者施加信息瓶颈惩罚，以防止代理关注无关的观察细节。（例如，参见 Wan+22] 的去噪 MDP 方法。）

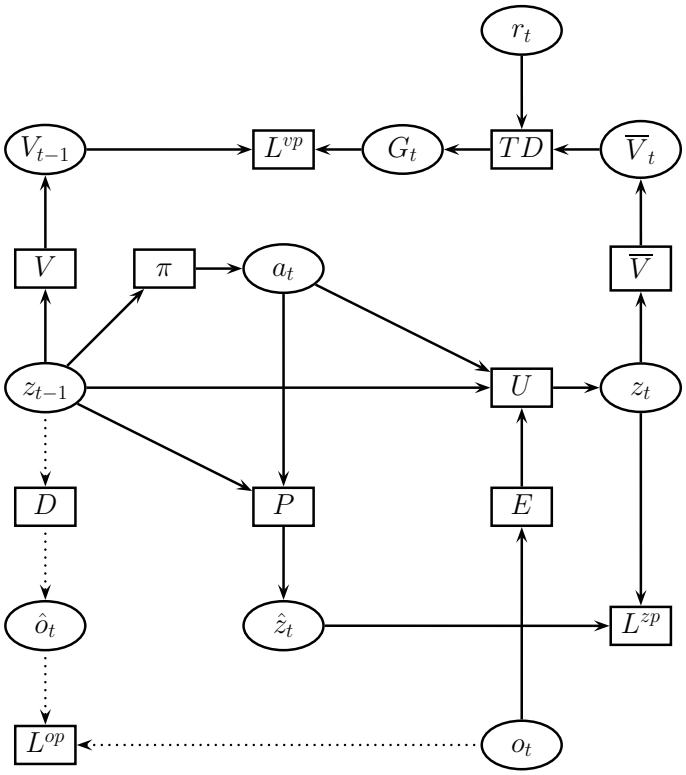


Figure 4.4: Illustration of (a simplified version of) the BYOL-Explore architecture, represented as a factor graph (so squares are functions, circles are variables). The dotted lines represent an optional observation prediction loss. The map from notation in this figure to the paper is as follows: $U \to h^c$ (closed-loop RNN update), $P \to h^o$ (open-loop RNN update), $D \to g$ (decoder), $E \to f$ (encoder). We have unrolled the forwards prediction for only 1 step. Also, we have omitted the reward prediction loss. The $\overline{V}$ node is the EMA version of the value function. The TD node is the TD operator.

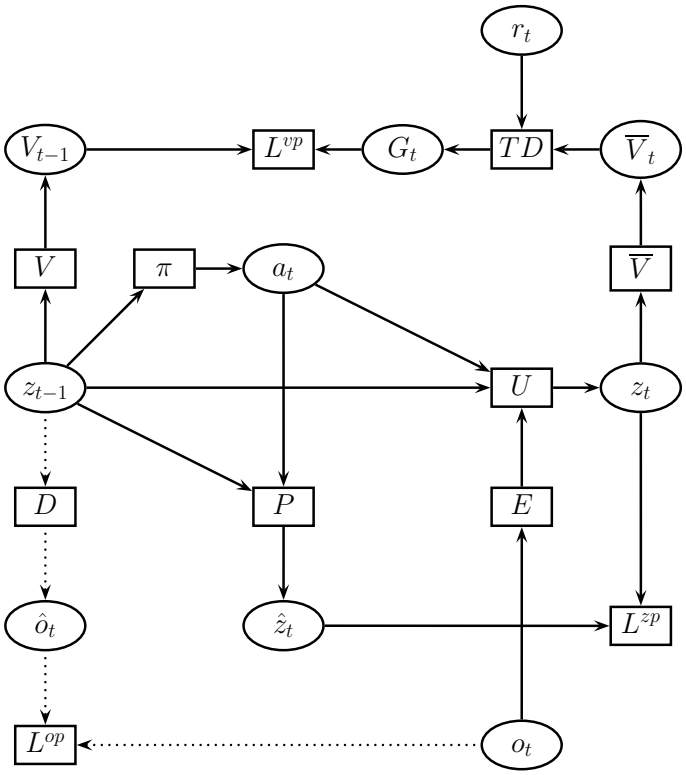


图 4.4：（简化版） BYOL-Explore 架构的示意图，表示为因子图（因此正方形是函数，圆形是变量）。虚线表示可选的观察预测损失。本图中的符号与论文中的对应关系如下：$U \to h^c$ （闭环 RNN 更新），$P \to h^o$ （开环 RNN 更新），$D \to g$ （解码器），$E \to f$ （编码器）。我们只展开前向预测了 1 步。此外，我们还省略了奖励预测损失。V 节点是值函数的 EMA 版本。TD 节点是 TD 算子。

4.3.2.6 BYOL-Explore

As an example of the above framework, consider the **BYOL-Explore** paper [Guo+22a], which uses a non-generative world model trained with ZP and VP loss. (BYOL stands for "build your own latent".) See Figure 4.4 for the computation graph, which we see is slightly simpler than the Dreamer computation graph in Figure 4.2 due to the lack of stochastic latents. In addition to using self-prediction loss to help train the latent representation, the error in this loss can be used to define an intrinsic reward, to encourage the agent to explore states where the model is uncertain. See Section 5.2.4 for further discussion of this topic.

4.4 Beyond one-step models: predictive representations

The "world models" we described in Section 4.3 are **one-step models** of the form $p(s'|s,a)$, or $p(z'|z,a)$ for $z = \phi(s)$, where ϕ is a state-abstraction function. However, such models are problematic when it comes to predicting many kinds of future events, such as "will a car pull in front of me?" or "when will it start raining?", since it is hard to predict exactly when these events will occur, and these events may correspond to many different "ground states". In principle we can roll out many possible long term futures, and apply some abstraction function to the resulting generated trajectories to extract features of interest, and thus derive a predictive model of the form $p(t', \phi(s_{t+1:t'})|s_t, \pi)$, where t' is the random duration of the sampled trajectory. However, it would be more efficient if we could directly predict this distribution without having to know the value of t', and without having to predict all the details of all the intermediate future states, many of which will be irrelevant given the abstraction function ϕ. This motivates the study of multi-step world models, that predict multiple steps into the future, either at the state level, or at the feature level. These are called **predictive representations**, and are a compromise between standard model-based RL and model-free RL, as we will see. Our presentation on this topic is based on [Car+24]. (See also Section 5.3, where we discuss the related topic of temporal abstraction from a model-free perspective.)

4.4.1 General value functions

The value function is based on predicting the sum of expected discounted future rewards. But the reward is just one possible signal of interest we can extract from the environment. We can generalize this by considering a **cumulant** $C_t \in \mathbb{R}$, which is some scalar of interest derived from the state or observation (e.g., did a loud bang just occur? is there a tree visible in the image?). We then define the **general value function** or **GVF** as follows [Sut95]:

$$V^{\pi,C,\gamma}(s) = \mathbb{E}\left[\sum_{t=0}^{\infty} \gamma^t C(s_{t+1}) | s_0 = s, a_{0:\infty} \sim \pi\right] \tag{4.29}$$

If $C(s_{t+1}) = R_{t+1}$, this reduces to the value function.[5] However, we can also define the GVF to predict components of the observation vector; this is called **nexting** [MWS14], since it refers to next state prediction at different timescales.

4.4.2 Successor representations

In this section we consider a variant of GVF where the cumulant corresponds to a state occupancy vector $C(s_{t+1}) = \mathbb{I}(s_{t+1} = \tilde{s})$, which provides a dense feedback signal. This give us the **successor representation** or **SR** [Day93]:

$$M^{\pi}(s, \tilde{s}) = \mathbb{E}\left[\sum_{t=0}^{\infty} \gamma^t \mathbb{I}(s_{t+1} = \tilde{s}) | S_0 = s\right] \tag{4.30}$$

[5] This follows the convention of [SB18], where we write $(s_t, a_t, r_{t+1}, s_{t+1})$ to represent the transitions, since r_{t+1} and s_{t+1} are both generated by applying a_t in state s_t.

4.3.2.6 BYOL-Explore 无法翻译

作为一个上述框架的例子，考虑 **BYOL-Explore** 论文 [郭 +22 亚], ，该论文使用与 ZP 和 VP 损失训练的非生成式世界模型。（ BYOL 代表 “ 构建自己的潜在 ” ）。参见图 4.4 ，该计算图比图 4.2 中的 Dreamer 计算图略简单，因为缺乏随机潜在。除了使用自预测损失来帮助训练潜在表示外，该损失的误差还可以用来定义一个内在奖励，以鼓励智能体探索模型不确定的状态。参见第 5.2.4 节，以进一步讨论此主题。

4.4 超越单步模型：预测表示

我们在第 4.3 节中描述的 “ 世界模型 ” 是 **一种一步模型** ，形式为 $p(s'|s,a)$，或者 $p(z'|z,a)$，用于 $z=\phi(s)$，其中 ϕ 是一个状态抽象函数。然而，当预测许多类型的未来事件时，例如 “ 一辆车是否会出现在我前面？ ” 或 “ 何时开始下雨？ ”，这些模型存在问题，因为这些事件的确切发生时间难以预测，并且这些事件可能对应于许多不同的 “ 基态 ”。原则上，我们可以模拟许多可能的长远未来，并应用一些抽象函数来提取生成的轨迹的特征，从而推导出形式为 $p(t',\phi(s_{t+1:t'})|s_t,\pi)$ 的预测模型，其中 t' 是采样轨迹的随机持续时间。然而，如果我们能直接预测这个分布，而无需知道 t' 的值，也无需预测所有中间未来状态的所有细节（其中许多在抽象函数 ϕ 下可能是不相关的），这将更有效率。这促使我们研究多步世界模型，这些模型可以预测未来多个步骤，无论是在状态层面还是在特征层面。这些被称为**预测表示**，它们在标准基于模型的强化学习和无模型强化学习之间提供了一个折衷方案，正如我们将看到的。我们关于这个主题的介绍基于 [Car+24]。(另见第 5.3 节，其中我们从无模型的角度讨论了相关的时间抽象问题。)

4.4.1 通用值函数

价值函数基于预测期望折现未来奖励的总和。但奖励只是我们可以从环境中提取的许多感兴趣的可能信号之一。我们可以通过考虑一个**累积量** $C_t \in \mathbb{R}$ 来推广这一点，它是从状态或观察中得到的某个感兴趣标量（例如，是否刚刚发生了巨大的爆炸声？图像中是否可见树木？）。然后我们定义**广义价值函数**或 **GVF** 如下 [Sut95]:

$$\left[V^{\pi,C,\gamma}(s) \right] = \mathbb{E}\left[\sum_{t=0}^{\infty} \gamma^t C(s_{t+1}) | s_0 = s, a_{0:\infty} \sim \pi \right] \tag{4.29}$$

如果 $C(s_{t+1}) = R_{t+1}$，这就简化为值函数。[5] 然而，我们还可以定义 GVF 来预测观测向量组件；这被称为 **下一时刻** [MWS14], ，因为它涉及不同时间尺度的下一状态预测。

4.4.2 后继表示

在本节中，我们考虑 GVF 的一种变体，其中矩量对应于状态占用向量 $C(s_{t+1}) = \mathbb{I}(s_{t+1} = \tilde{s})$，这提供了一个密集的反馈信号。这为我们提供了**后继表示**或 **SR** [Day93]:

$$M^{\pi}(s,\tilde{s}) = \mathbb{E}\left[\sum_{t=0}^{\infty} \gamma^t \mathbb{I}\left(s_{t+1} = \tilde{s}\right) | S_0 = s \right] \tag{4.30}$$

[5] 遵循了[SB18], 的惯例，其中我们写作$(s_t,a_t,r_{t+1},s_{t+1})$ 来表示转换，因为r_{t+1} 和s_{t+1}都是由在状态a_t 下应用s_t生成的。

If we define the policy-dependent state-transition matrix by

$$T^\pi(s,s') = \sum_a \pi(a|s) T(s'|s,a) \tag{4.31}$$

then the SR matrix can be rewritten as

$$\mathbf{M}^\pi = \sum_{t=0}^{\infty} \gamma^t [\mathbf{T}^\pi]^{t+1} = \mathbf{T}^\pi (\mathbf{I} - \gamma \mathbf{T}^\pi)^{-1} \tag{4.32}$$

Thus we see that the SR replaces information about individual transitions with their cumulants, just as the value function replaces individual rewards with the reward-to-go.

Like the value function, the SR obeys a Bellman equation

$$M^\pi(s,\tilde{s}) = \sum_a \pi(a|s) \sum_{s'} T(s'|s,a) \left(\mathbb{I}(s'=\tilde{s}) + \gamma M^\pi(s',\tilde{s}) \right) \tag{4.33}$$

$$= \mathbb{E}\left[\mathbb{I}(s'=\tilde{s}) + \gamma M^\pi(s',\tilde{s}) \right] \tag{4.34}$$

Hence we can learn an SR using a TD update of the form

$$M^\pi(s,\tilde{s}) \leftarrow M^\pi(s,\tilde{s}) + \eta \underbrace{\left(\mathbb{I}(s'=\tilde{s}) + \gamma M^\pi(s',\tilde{s}) - M^\pi(s,\tilde{s}) \right)}_{\delta} \tag{4.35}$$

where s' is the next state sampled from $T(s'|s,a)$. Compare this to the value-function TD update in Equation (2.16):

$$V^\pi(s) \leftarrow V^\pi(s) + \eta \underbrace{\left(R(s') + \gamma V^\pi(s') - V^\pi(s) \right)}_{\delta} \tag{4.36}$$

However, with an SR, we can easily compute the value function for any reward function (as approximated by a given policy) as follows:

$$V^{R,\pi} = \sum_{\tilde{s}} M^\pi(s,\tilde{s}) R(\tilde{s}) \tag{4.37}$$

See Figure 4.5 for an example.

We can also make a version of SR that depends on the action as well as the state to get

$$M^\pi(s,a,\tilde{s}) = \mathbb{E}\left[\sum_{t=0}^{\infty} \gamma^t \mathbb{I}(s_{t+1}=\tilde{s}) \,|\, s_0 = s, a_0 = a, a_{1:\infty} \sim \pi \right] \tag{4.38}$$

$$= \mathbb{E}\left[\mathbb{I}(s'=\tilde{s}) + \gamma M^\pi(s',a,\tilde{s}) | s_0 = s, a_0 = a, a_{1:\infty} \sim \pi \right] \tag{4.39}$$

This gives rise to a TD update of the form

$$M^\pi(s,a,\tilde{s}) \leftarrow M^\pi(s,a,\tilde{s}) + \eta \underbrace{\left(\mathbb{I}(s'=\tilde{s}) + \gamma M^\pi(s',a',\tilde{s}) - M^\pi(s,a,\tilde{s}) \right)}_{\delta} \tag{4.40}$$

where s' is the next state sampled from $T(s'|s,a)$ and a' is the next action sampled from $\pi(s')$. Compare this to the (on-policy) SARSA update from Equation (2.28):

$$Q^\pi(s,a) \leftarrow Q^\pi(s,a) + \eta \underbrace{\left(R(s') + \gamma Q^\pi(s',a') - Q^\pi(s,a) \right)}_{\delta} \tag{4.41}$$

However, from an SR, we can compute the state-action value function for any reward function:

$$Q^{R,\pi}(s,a) = \sum_{\tilde{s}} M^\pi(s,a,\tilde{s}) R(\tilde{s}) \tag{4.42}$$

如果我们定义策略相关的状态转移矩阵为

$$T^{\pi}(s,s') = \sum_{a} \pi(a|s) T(s'|s,a) \tag{4.31}$$

然后，SR 矩阵可以重写为

$$\mathbf{M}^{\pi} = \sum_{t=0}^{\infty} \gamma^{t} [\mathbf{T}^{\pi}]^{t+1} = \mathbf{T}^{\pi} (\mathbf{I} - \gamma \mathbf{T}^{\pi})^{-1} \tag{4.32}$$

因此，我们看到 SR 用累积量替换了关于个别转换的信息，就像价值函数用奖励到下一次奖励替换个别奖励一样。

类似于值函数，SR 遵循贝尔曼方程 n

$$M^{\pi}(s,\tilde{s}) = \sum_{a} \pi(a|s) \sum_{s'} T(s'|s,a) \left(\mathbb{I}(s'=\tilde{s}) + \gamma M^{\pi}(s',\tilde{s}) \right) \tag{4.33}$$

$$= \mathbb{E}\left[\mathbb{I}(s'=\tilde{s}) + \gamma M^{\pi}(s',\tilde{s}) \right] \tag{4.34}$$

因此，我们可以通过以下形式的 TD 更新学习 SR：

$$M^{\pi}(s,\tilde{s}) \leftarrow M^{\pi}(s,\tilde{s}) + \eta \underbrace{\left(\mathbb{I}(s'=\tilde{s}) + \gamma M^{\pi}(s',\tilde{s}) - M^{\pi}(s,\tilde{s}) \right)}_{\delta} \tag{4.35}$$

其中 s' 是从 $T(s'|s,a)$ 中采样的下一个状态。将其与方程（ 2.16 ）中的值函数 TD 更新进行比较：

$$V^{\pi}(s) \leftarrow V^{\pi}(s) + \eta \underbrace{\left(R(s') + \gamma V^{\pi}(s') - V^{\pi}(s) \right)}_{\delta} \tag{4.36}$$

然而，有了 SR，我们可以轻松地计算任何奖励函数（由给定的策略近似）的价值函数，如下所示：

$$V^{R,\pi} = \sum_{\tilde{s}} M^{\pi}(s,\tilde{s}) R(\tilde{s}) \tag{4.37}$$

参见图 4.5 以获取示例。

我们也可以制作一个 SR 版本，它不仅依赖于状态，还依赖于动作来获取 [ ]

$$M^{\pi}(s,a,\tilde{s}) = \mathbb{E}\left[\sum_{t=0}^{\infty} \gamma^{t} \mathbb{I}(s_{t+1}=\tilde{s}) \,|\, s_0 = s, a_0 = a, a_{1:\infty} \sim \pi \right] \tag{4.38}$$

$$= \mathbb{E}\left[\mathbb{I}(s'=\tilde{s}) + \gamma M^{\pi}(s',a,\tilde{s}) | s_0 = s, a_0 = a, a_{1:\infty} \sim \pi \right] \tag{4.39}$$

这导致了一种形式的 TD 更新

$$M^{\pi}(s,a,\tilde{s}) \leftarrow M^{\pi}(s,a,\tilde{s}) + \eta \underbrace{\left(\mathbb{I}(s'=\tilde{s}) + \gamma M^{\pi}(s',a',\tilde{s}) - M^{\pi}(s,a,\tilde{s}) \right)}_{\delta} \tag{4.40}$$

其中 s' 是从 $T(s'|s,a)$ 中采样的下一个状态，a' 是从 $\pi(s')$ 中采样的下一个动作。将其与方程（ 2.28 ）中的（策略） SARSA 更新进行比较：

$$Q^{\pi}(s,a) \leftarrow Q^{\pi}(s,a) + \eta \underbrace{\left(R(s') + \gamma Q^{\pi}(s',a') - Q^{\pi}(s,a) \right)}_{\delta} \tag{4.41}$$

然而，从 SR 中，我们可以为任何奖励函数计算状态 - 动作值函数：

$$Q^{R,\pi}(s,a) = \sum_{\tilde{s}} M^{\pi}(s,a,\tilde{s}) R(\tilde{s}) \tag{4.42}$$

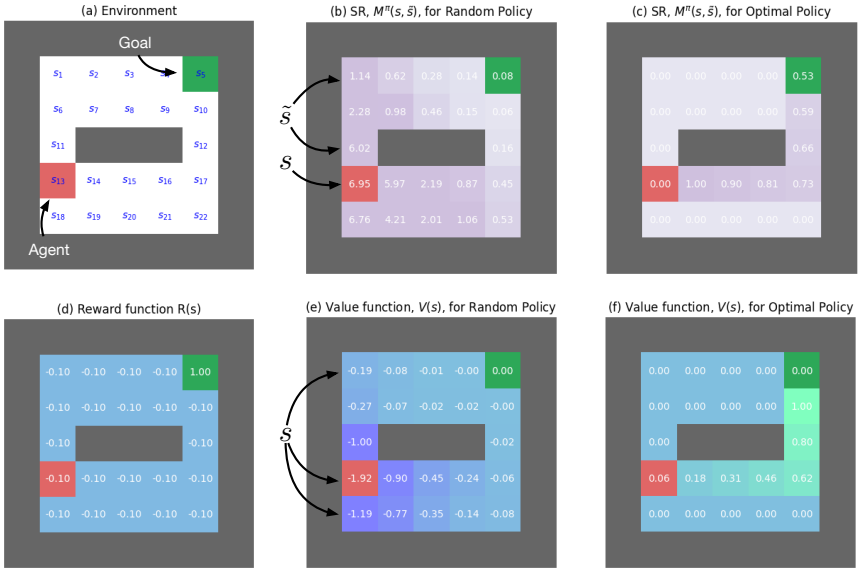


Figure 4.5: Illustration of successor representation for the 2d maze environment shown in (a) with reward shown in (d), which assings all states a reward of -0.1 except for the goal state which has a reward of 1.0. In (b-c) we show the SRs for a random policy and the optimal policy. In (e-f) we show the corresponding value functons. In (b), we see that the SR under the random policy assigns high state occupancy values to states which are close (in Manhattan distance) to the current state s_{13} (e.g., $M^\pi(s_{13}, s_{14}) = 5.97$) and low values to states that are further away (e.g., $M^\pi(s_{13}, s_{12}) = 0.16$). In (c), we see that the SR under the optimal policy assigns high state occupancy values to states which are close to the optimal path to the goal (e.g., $M^\pi(s_{13}, s_{14}) = 1.0$) and which fade with distance from the current state along that path (e.g., $M^\pi(s_{13}, s_{12}) = 0.66$). From Figure 3 of [Car+24]. Used with kind permission of Wilka Carvalho. Generated by `https://github.com/wcarvalho/jaxneurorl/blob/main/successor_representation.ipynb` .

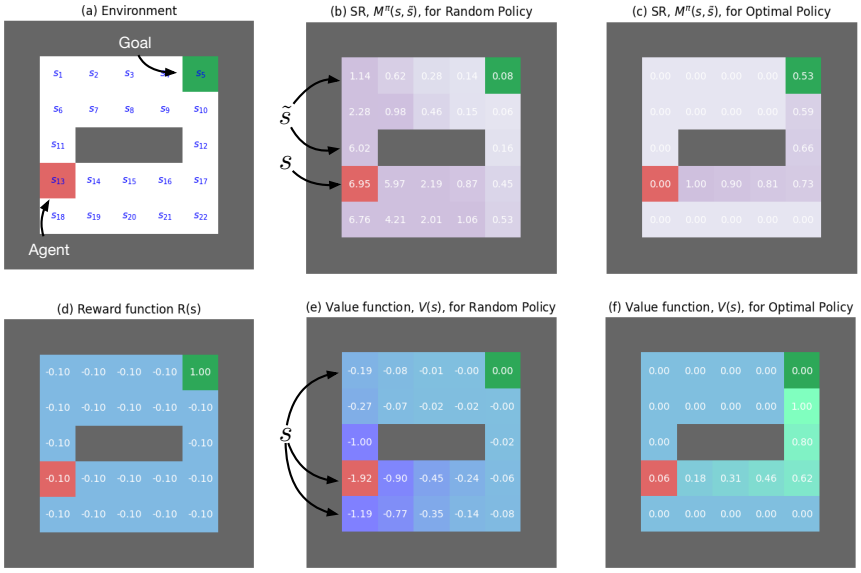


图 4.5：展示在（a）中所示的 2D 迷宫环境的后继表示，其中（d）显示了奖励，除了目标状态具有 1.0 的奖励外，所有状态都分配了 -0.1 的奖励。在（b-c）中，我们展示了随机策略和最优策略的 SR。在（e-f）中，我们展示了相应的值函数。在（b）中，我们看到随机策略下的 SR 将高状态占用值分配给接近当前状态（例如，曼哈顿距离）的状态（例如，$M^{\pi}(s_{13}, s_{14}) = 5.97$）并将低值分配给更远的（例如，$M^{\pi}(s_{13}, s_{12}) = 0.16$）状态。在（c）中，我们看到最优策略下的 SR 将高状态占用值分配给接近目标最优路径的状态（例如，$M^{\pi}(s_{13}, s_{14}) = 1.0$），并且这些值随着从当前状态沿该路径的距离而衰减（例如，$M^{\pi}(s_{13}, s_{12}) = 0.66$）。来自 [*Car+24*] 的第 3 图。经 Wilka Carvalho 许可使用。由 *https://github.com/wcarvalho/jaxneurorl/blob/main/successor_representation.ipynb* 生成。

This can be used to improve the policy as we discuss in Section 4.4.4.1.

We see that the SR representation has the computational advantages of model-free RL (no need to do explicit planning or rollouts in order to compute the optimal action), but also the flexibility of model-based RL (we can easily change the reward function without having to learn a new value function). This latter property makes SR particularly well suited to problems that use intrinsic reward (see Section 5.2.4), which often changes depending on the information state of the agent.

Unfortunately, the SR is limited in several ways: (1) it assumes a finite, discrete state space; (2) it depends on a given policy. We discuss ways to overcome limitation 1 in Section 4.4.3, and limitation 2 in Section 4.4.4.1.

4.4.3 Successor models

In this section, we discuss the **successor model** (also called a γ-model), which is a probabilistic extension of SR [JML20; Eys+21]. This allows us to generalize SR to work with continuous states and actions, and to simulate future state trajectories. The approach is to define the cumulant as the k-step conditional distribution $C(s_{k+1}) = P(s_{k+1} = \tilde{s}|s_0 = s, \pi)$, which is the probability of being in state $\tilde{s}$ after following π for k steps starting from state s. (Compare this to the SR cumulant, which is $C(s_{k+1}) = \mathbb{I}(s_{k+1} = \tilde{s})$.) The SM is then defined as

$$\boldsymbol{\mu}^\pi(\tilde{s}|s) = (1-\gamma)\sum_{t=0}^{\infty}\gamma^t P(s_{t+1} = \tilde{s}|s_0 = s) \tag{4.43}$$

where the $1-\gamma$ term ensures that $\boldsymbol{\mu}^\pi$ integrates to 1. (Recall that $\sum_{t=0}^{\infty}\gamma^t = \frac{1}{1-\gamma}$ for $\gamma < 1$.) In the tabular setting, the SM is just the normalized SR, since

$$\boldsymbol{\mu}^\pi(\tilde{s}|s) = (1-\gamma)M^\pi(s,\tilde{s}) \tag{4.44}$$

$$= (1-\gamma)\mathbb{E}\left[\sum_{t=0}^{\infty}\gamma^t\mathbb{I}(s_{t+1} = \tilde{s})\,|s_0 = s, a_{0:\infty} \sim \pi\right] \tag{4.45}$$

$$= (1-\gamma)\sum_{t=0}^{\infty}\gamma^t P(s_{t+1} = \tilde{s}|s_0 = s, \pi) \tag{4.46}$$

Thus $\boldsymbol{\mu}^\pi(\tilde{s}|s)$ tells us the probability that $\tilde{s}$ can be reached from s within a horizon determined by γ when following π, even though we don't know exactly when we will reach $\tilde{s}$.

SMs obey a Bellman-like recursion

$$\boldsymbol{\mu}^\pi(\tilde{s}|s) = \mathbb{E}\left[(1-\gamma)T(\tilde{s}|s,a) + \gamma\boldsymbol{\mu}^\pi(\tilde{s}|s')\right] \tag{4.47}$$

We can use this to perform policy evaluation by computing

$$V^\pi(s) = \frac{1}{1-\gamma}\mathbb{E}_{\boldsymbol{\mu}^\pi(\tilde{s}|s)}\left[R(\tilde{s})\right] \tag{4.48}$$

We can also define an action-conditioned SM

$$\boldsymbol{\mu}^\pi(\tilde{s}|s,a) = (1-\gamma)\sum_{t=0}^{\infty}\gamma^t P(s_{t+1} = \tilde{s}|s_0 = s, a_0 = a) \tag{4.49}$$

$$= (1-\gamma)T(\tilde{s}|s,a) + \gamma\mathbb{E}\left[\boldsymbol{\mu}^\pi(\tilde{s}|s',a',\pi)\right] \tag{4.50}$$

Hence we can learn an SM using a TD update of the form

$$\boldsymbol{\mu}^\pi(\tilde{s}|s,a) \leftarrow \boldsymbol{\mu}^\pi(\tilde{s}|s,a) + \eta\underbrace{\left((1-\gamma)T(s'|s,a) + \gamma\boldsymbol{\mu}^\pi(\tilde{s}|s',a') - \boldsymbol{\mu}^\pi(\tilde{s}|s,a)\right)}_{\delta} \tag{4.51}$$

这可以用来改进我们在第 4.4.4.1 节中讨论的政策。

我们发现 SR 表示具有无模型 RL 的计算优势（无需进行显式规划或回滚即可计算最优动作），同时也具有基于模型的 RL 的灵活性（我们可以轻松更改奖励函数，而无需学习新的值函数）。这种后一种特性使 SR 特别适合使用内在奖励的问题（参见第 5.2.4 节），这通常取决于代理的信息状态。

很遗憾，SR 在几个方面受到限制：（1）它假设一个有限、离散的状态空间；（2）它依赖于一个给定的策略。我们将在第 4.4.3 节讨论克服限制 1 的方法，在第 4.4.4.1 节讨论克服限制 2 的方法。

4.4.3 后继模型

在本节中，我们讨论**继任模型**（也称为 γ- 模型），它是对 SR[JML20； Eys+21] 的概率扩展； k 步条件分布 $C\left(s_{k+1}\right)=P\left(s_{k+1}=\tilde{s} | s_0=s, \pi\right)$，这是在从状态 $\tilde{s}$ 开始跟随 π 进行 k 步后处于状态的概率。（与 SR 累积量进行比较，SR 累积量是 $C\left(s_{k+1}\right)=\mathbb{I}\left(s_{k+1}=\tilde{s}\right)$。）然后定义 SM 为

$$\boldsymbol{\mu}^\pi(\tilde{s}|s) = (1-\gamma)\sum_{t=0}^{\infty}\gamma^t P(s_{t+1}=\tilde{s}|s_0=s) \tag{4.43}$$

$1-\gamma$ 项确保 $\boldsymbol{\mu}^\pi$ 积分等于 1。（回忆 $\sum_{t=0}^{\infty}\gamma^t = \frac{1}{1-\gamma}$ 对于 $\gamma<1$。）在表格设置中，SM 只是归一化的 SR，因为

$$\begin{aligned}
\boldsymbol{\mu}^\pi(\tilde{s}|s) &= (1-\gamma)M^\pi(s,\tilde{s}) && (4.44)\\
&= (1-\gamma)\mathbb{E}\left[\sum_{t=0}^{\infty}\gamma^t\mathbb{I}\left(s_{t+1}=\tilde{s}\right)|s_0=s, a_{0:\infty}\sim\pi\right] && (4.45)\\
&= (1-\gamma)\sum_{t=0}^{\infty}\gamma^t P(s_{t+1}=\tilde{s}|s_0=s,\pi) && (4.46)
\end{aligned}$$

因此，$\boldsymbol{\mu}^\pi(\tilde{s}|s)$ 告诉我们，在遵循 π 的情况下，在由 γ 确定的视域内，从 s 到达 $\tilde{s}$ 的概率，即使我们不知道确切何时会到达 $\tilde{s}$。

短信遵循类似贝尔曼的递归

$$\boldsymbol{\mu}^\pi(\tilde{s}|s) = \mathbb{E}\left[(1-\gamma)T(\tilde{s}|s,a) + \gamma\boldsymbol{\mu}^\pi(\tilde{s}|s')\right] \tag{4.47}$$

我们可以用它通过计算来执行策略评估

$$V^\pi(s) = \frac{1}{1-\gamma}\mathbb{E}_{\boldsymbol{\mu}^\pi(\tilde{s}|s)}\left[R(\tilde{s})\right] \tag{4.48}$$

我们还可以定义一个动作条件 SM

$$\begin{aligned}
\boldsymbol{\mu}^\pi(\tilde{s}|s,a) &= (1-\gamma)\sum_{t=0}^{\infty}\gamma^t P(s_{t+1}=\tilde{s}|s_0=s,a_0=a) && (4.49)\\
&= (1-\gamma)T(\tilde{s}|s,a) + \gamma\mathbb{E}\left[\boldsymbol{\mu}^\pi(\tilde{s}|s',a',\pi)\right] && (4.50)
\end{aligned}$$

因此，我们可以使用以下形式的 TD 更新来学习一个 SM

$$\boldsymbol{\mu}^\pi(\tilde{s}|s,a) \leftarrow \boldsymbol{\mu}^\pi(\tilde{s}|s,a) + \eta\underbrace{\left((1-\gamma)T(s'|s,a) + \gamma\boldsymbol{\mu}^\pi(\tilde{s}|s',a') - \boldsymbol{\mu}^\pi(\tilde{s}|s,a)\right)}_{\delta} \tag{4.51}$$

where s' is the next state sampled from $T(s'|s,a)$ and a' is the next action sampled from $\pi(s')$. With an SM, we can compute the state-action value for any reward:

$$Q^{R,\pi}(s,a) = \frac{1}{1-\gamma}\mathbb{E}_{\boldsymbol{\mu}^\pi(\tilde{s}|s,a)}\left[R(\tilde{s})\right] \tag{4.52}$$

This can be used to improve the policy as we discuss in Section 4.4.4.1.

4.4.3.1 Learning SMs

Although we can learn SMs using the TD update in Equation (4.51), this requires evaluating $T(s'|s,a)$ to compute the target update δ, and this one-step transition model is typically unknown. Instead, since $\boldsymbol{\mu}^\pi$ is a conditional density model, we will optimize the cross-entropy TD loss [JML20], defined as follows

$$\mathcal{L}_\mu = \mathbb{E}_{(s,a)\sim p(s,a),\tilde{s}\sim(T^\pi\boldsymbol{\mu}^\pi)(\cdot|s,a)}\left[\log \boldsymbol{\mu}_{\boldsymbol{\theta}}(\tilde{s}|s,a)\right] \tag{4.53}$$

where $(T^\pi\boldsymbol{\mu}^\pi)(\cdot|s,a)$ is the Bellman operator applied to $\boldsymbol{\mu}^\pi$ and then evaluated at (s,a), i.e.,

$$(T^\pi\boldsymbol{\mu}^\pi)(\tilde{s}|s,a) = (1-\gamma)T(s'|s,a) + \gamma\sum_{s'}T(s'|s,a)\sum_{a'}\pi(a'|s'(\boldsymbol{\mu}^\pi(\tilde{s}|s',a') \tag{4.54}$$

We can sample from this as follows: first sample $s' \sim T(s'|s,a)$ from the environment and then with probability $1-\gamma$ set $\tilde{s} = s'$ and terminate. Otherwise sample $a' \sim \pi(a'|s')$ and then create a bootstrap sample from the model using $\tilde{s} \sim \boldsymbol{\mu}^\pi(\tilde{s}|s',a')$.

There are many possible density models we can use for $\boldsymbol{\mu}^\pi$. In [Tha+22], they use a VAE. In [Tom+24], they use an autoregressive transformer applied to a set of discrete latent tokens, which are learned using VQ-VAE or a non-reconstructive self-supervised loss. They call their method **Video Occcupancy Models**.

An alternative approach to learning SMs, that avoids fitting a normalized density model over states, is to use contrastive learning to estimate how likely $\tilde{s}$ is to occur after some number of steps, given (s,a), compared to some randomly sampled negative state [ESL21; ZSE24]. Although we can't sample from the resulting learned model (we can only use it for evaluation), we can use it to improve a policy that achieves a target state (an approach known as goal-conditioned policy learning, discussed in Section 5.3.1).

4.4.3.2 Jumpy models using geometric policy composition

In [Tha+22], they propose **geometric policy composition** or GPC as a way to learn a new policy by sequencing together a set of N policies, as opposed to taking N primitive actions in a row. This can be thought of as a **jumpy model**, since it predicts multiple steps into the future, instead of one step at a time (c.f., [Zha+23a]).

In more detail, in GPC, the agent picks a sequence of n policies π_i for $i = 1:n$, and then samples states according to their corresponding SMs: starting with (s_0,a_0), we sample $s_1 \sim \boldsymbol{\mu}_\gamma^{\pi_1}(\cdot|s_0,a_0)$, then $a_1 \sim \pi_1(\cdot|s_1)$, then $s_2 \sim \boldsymbol{\mu}_\gamma^{\pi_2}(\cdot|s_1,a_1)$, etc. This continues for $n-1$ steps. Finally we sample $s_n \sim \boldsymbol{\mu}_{\gamma'}^{\pi_n}(\cdot|s_{n-1},a_{n-1})$, where $\gamma' > \gamma$ represents a longer horizon SM. The reward estimates computed along this sampled path can then be combined to compute the value of each candidate policy sequence.

4.4.4 Successor features

Both SRs and SMs require defining expectations or distributions over the entire future state vector, which can be problematic in high dimensional spaces. In [Bar+17] they introduced **successor features**, that generalize SRs by working with features $\boldsymbol{\phi}(s)$ instead of primitive states. In particular, if we define the cumulant to be $C(s_{t+1}) = \boldsymbol{\phi}(s_{t+1})$, we get the following definition of SF:

$$\boldsymbol{\psi}^{\pi,\boldsymbol{\phi}}(s) = \mathbb{E}\left[\sum_{t=0}^{\infty}\gamma^t\boldsymbol{\phi}(s_{t+1})|s_0 = s, a_{0:\infty}\sim\pi\right] \tag{4.55}$$

s' 是从 $T(s'|s,a)$ 中采样的下一个状态，a' 是从 $\pi(s')$ 中采样的下一个动作。使用 SM，我们可以计算任何奖励的状态 - 动作值：

$$Q^{R,\pi}(s,a) = \frac{1}{1-\gamma}\mathbb{E}_{\boldsymbol{\mu}^\pi(\tilde{s}|s,a)}\left[R(\tilde{s})\right] \tag{4.52}$$

这可以用来改进我们在第 4.4.4.1 节中讨论的政策。

4.4.3.1 学习 SMs

尽管我们可以使用方程（4.51）中的 TD 更新来学习 SMs，但这需要评估 $T(s'|s,a)$ 以计算目标更新 δ，而这个一步过渡模型通常是未知的。相反，由于 $\boldsymbol{\mu}^\pi$ 是一个条件密度模型，我们将优化定义如下 [JML20]，的交叉熵 TD 损失

$$\mathcal{L}_\mu = \mathbb{E}_{(s,a)\sim p(s,a),\tilde{s}\sim(T^\pi\boldsymbol{\mu}^\pi)(\cdot|s,a)}\left[\log \boldsymbol{\mu}_{\boldsymbol{\theta}}(\tilde{s}|s,a)\right] \tag{4.53}$$

其中（$T^\pi\boldsymbol{\mu}^\pi$）（$\cdot|s,\ a$）是应用于 $\boldsymbol{\mu}^\pi$ 的 Bellman 算子，然后在该（$s,\ a$）处进行评估，即

$$(T^\pi\boldsymbol{\mu}^\pi)(\tilde{s}|s,a) = (1-\gamma)T(s'|s,a) + \gamma\sum_{s'}T(s'|s,a)\sum_{a'}\pi(a'|s'(\boldsymbol{\mu}^\pi(\tilde{s}|s',a') \tag{4.54}$$

我们可以这样采样：首先从环境中采样 $s'\sim T(s'|s,a)$，然后以概率 $1-\gamma$ 设置 $\tilde{s}=s'$ 并终止。否则，采样 $a'\sim\pi(a'|s')$，然后使用 $\tilde{s}\sim\boldsymbol{\mu}^\pi(\tilde{s}|s',a')$ 从模型中创建一个自举样本。

我们可以使用许多可能的密度模型来用于 $\boldsymbol{\mu}^\pi$。在 [Tha+22]，他们使用了一个 VAE。在 [Tom+24]，他们使用了一个应用于一组离散潜在标记的自回归变换器，这些标记是通过 VQ-VAE 或非重建的自监督损失来学习的。他们将他们的方法称为 **视频占用模型**。

学习 SMs 的另一种方法，该方法避免在状态上拟合归一化密度模型，是使用对比学习来估计在经过一定步骤后，给定 (,)，$\tilde{s}$ 发生的可能性，与随机采样的负状态 [ESL21； ZSE24] 相比。尽管我们不能从学习到的模型中采样（我们只能用它进行评估），但我们可以用它来改进实现目标状态的政策（一种称为目标条件政策学习的方法，在第 5.3.1 节中讨论），。

4.4.3.2 使用几何策略组合的 Jumpymodels

在 [Tha+22] 中，他们提出 **几何策略组合** 或 GPC 作为通过将一组 N 策略按顺序排列来学习新策略的方法，而不是连续采取 N 原始动作。这可以被视为一个 **跳跃模型**，因为它预测多个步骤到未来，而不是一次预测一步（参见图 [Zha+23a]）。

更详细地说，在 GPC 中，智能体选择一系列 n 策略 π_i 用于 $i=1:\ n$，然后根据相应的 SMs 采样状态：从 $(s_0,\ a_0)$ 开始，我们采样 $s_1\sim\boldsymbol{\mu}_\gamma^{\pi_1}(\cdot|s_0,\ a_0)$，然后 $a_1\sim\pi_1(\cdot|s_1)$，然后 $s_2\sim\boldsymbol{\mu}_\gamma^{\pi_2}(\cdot|s_1,\ a_1)$，等等。这会持续 $n-1$ 步。最后，我们采样 $s_n\sim\boldsymbol{\mu}_{\gamma'}^{\pi_n}(\cdot|s_{n-1},\ a_{n-1})$，其中 $\gamma'>\gamma$ 代表一个更长的视野 SM。沿着这个采样路径计算出的奖励估计可以组合起来计算每个候选策略序列的价值。

4.4.4 后继特性

SRs 和 SMs 都需要在整个未来状态向量上定义期望或分布，这在高维空间中可能存在问题。在 [Bar+17] 中，他们引入了**后继特征**，通过使用特征 $\boldsymbol{\phi}(s)$ 而不是原始状态来泛化 SRs。特别是，如果我们定义累积量是 $C(s_{t+1})=\boldsymbol{\phi}(s_{t+1})$，我们得到以下 SF 的定义：

$$\boldsymbol{\psi}^{\pi,\boldsymbol{\phi}}(s) = \mathbb{E}\left[\sum_{t=0}^{\infty}\gamma^t\boldsymbol{\phi}(s_{t+1})|s_0=s,a_{0:\infty}\sim\pi\right] \tag{4.55}$$

We will henceforth drop the $\boldsymbol{\phi}$ superscript from the notation, for brevity. SFs obey a Bellman equation

$$\boldsymbol{\psi}(s) = \mathbb{E}\left[\boldsymbol{\phi}(s') + \gamma \boldsymbol{\psi}(s')\right] \tag{4.56}$$

If we assume the reward function can be written as

$$R(s, \boldsymbol{w}) = \boldsymbol{\phi}(s)^{\mathsf{T}} \boldsymbol{w} \tag{4.57}$$

then we can derive the value function for any reward as follows:

$$\begin{aligned} V^{\pi,\boldsymbol{w}}(s) &= \mathbb{E}\left[R(s_1) + \gamma R(s_2) + \cdots | s_0 = s\right] && (4.58)\\ &= \mathbb{E}\left[\boldsymbol{\phi}(s_1)^{\mathsf{T}} \boldsymbol{w} + \gamma \boldsymbol{\phi}(s_2)^{\mathsf{T}} \boldsymbol{w} + \cdots | s_0 = s\right] && (4.59)\\ &= \mathbb{E}\left[\sum_{t=0}^{\infty} \gamma^t \boldsymbol{\phi}(s_{t+1}) | s_0 = s\right]^{\mathsf{T}} \boldsymbol{w} = \boldsymbol{\psi}^{\pi}(s)^{\mathsf{T}} \boldsymbol{w} && (4.60) \end{aligned}$$

Similarly we can define an action-conditioned version of SF as

$$\begin{aligned} \boldsymbol{\psi}^{\pi,\boldsymbol{\phi}}(s,a) &= \mathbb{E}\left[\sum_{t=0}^{\infty} \boldsymbol{\phi}(s_{t+1}) | s_0 = s, a_0 = a, a_{1:\infty} \sim \pi\right] && (4.61)\\ &= \mathbb{E}\left[\boldsymbol{\phi}(s') + \gamma \boldsymbol{\psi}(s', a')\right] && (4.62) \end{aligned}$$

We can learn this using a TD rule

$$\boldsymbol{\psi}^{\pi}(s,a) \leftarrow \boldsymbol{\psi}^{\pi}(s,a) + \eta \underbrace{\left(\boldsymbol{\phi}(s') + \gamma \boldsymbol{\psi}^{\pi}(s', a') - \boldsymbol{\psi}^{\pi}(s,a)\right)}_{\delta} \tag{4.63}$$

And we can use it to derive a state-action value function:

$$Q^{\pi,\boldsymbol{w}}(s) = \boldsymbol{\psi}^{\pi}(s,a)^{\mathsf{T}} \boldsymbol{w} \tag{4.64}$$

This allows us to define multiple Q functions (and hence policies) just by changing the weight vector $\boldsymbol{w}$, as we discuss in Section 4.4.4.1.

4.4.4.1 Generalized policy improvement

So far, we have discussed how to compute the value function for a new reward function but using the SFs from an existing known policy. In this section we discuss how to create a new policy that is better than an existing set of policies, by using **Generalized Policy Improvement** or **GPI** [Bar+17; Bar+20].

Suppose we have learned a set of N (potentially optimal) policies π_i and their corresponding SFs $\boldsymbol{\psi}^{\pi_i}$ for maximizing rewards defined by $\boldsymbol{w}_i$. When presented with a new task $\boldsymbol{w}_{\text{new}}$, we can compute a new policy using GPI as follows:

$$a^*(s; \boldsymbol{w}_{\text{new}}) = \operatorname*{argmax}_a \max_i Q^{\pi_i}(s, a, \boldsymbol{w}_{\text{new}}) = \operatorname*{argmax}_a \max_i \boldsymbol{\psi}^{\pi_i}(s,a)^{\mathsf{T}} \boldsymbol{w}_{\text{new}} \tag{4.65}$$

If $\boldsymbol{w}_{\text{new}}$ is in the span of the training tasks (i.e., there exist weights α_i such that $\boldsymbol{w}_{\text{new}} \sum_i \alpha_i \boldsymbol{w}_i$), then the GPI theorem states that $\pi(a|s) = \mathbb{I}\left(a = a^*(s, \boldsymbol{w}_{\text{new}})\right)$ will perform at least as well as any of the existing policies, i.e., $Q^{\pi}(s,a) \geq \max_i Q^{\pi_i}(s,a)$ (c.f., policy improvement in Section 3.4). See Figure 4.6 for an illustration.

Note that GPI is a model-free approach to computing a new policy, based on an existing library of policies. In [Ale+23], they propose an extension that can also leverage a (possibly approximate) world model to learn better policies that can outperform the library of existing policies by performing more decision-time search.

我们将从此省略符号中的 $\boldsymbol{\phi}$ 上标，以简化表示。SFs 遵循 Bellman 方程

$$\boldsymbol{\psi}(s) = \mathbb{E}\left[\boldsymbol{\phi}(s') + \gamma\boldsymbol{\psi}(s')\right] \tag{4.56}$$

如果我们假设奖励函数可以写成

$$R(s, \boldsymbol{w}) = \boldsymbol{\phi}(s)^{\mathsf{T}}\boldsymbol{w} \tag{4.57}$$

然后我们可以推导出任何奖励的价值函数如下：

$$V^{\pi,\boldsymbol{w}}(s) = \mathbb{E}\left[R(s_1) + \gamma R(s_2) + \cdots | s_0 = s\right] \tag{4.58}$$

$$= \mathbb{E}\left[\boldsymbol{\phi}(s_1)^{\mathsf{T}}\boldsymbol{w} + \gamma\boldsymbol{\phi}(s_2)^{\mathsf{T}}\boldsymbol{w} + \cdots | s_0 = s\right] \tag{4.59}$$

$$= \mathbb{E}\left[\sum_{t=0}^{\infty}\gamma^t\boldsymbol{\phi}(s_{t+1})|s_0 = s\right]^{\mathsf{T}}\boldsymbol{w} = \boldsymbol{\psi}^{\pi}(s)^{\mathsf{T}}\boldsymbol{w} \tag{4.60}$$

类似地，我们可以定义一个基于动作的 SF 版本

$$\boldsymbol{\psi}^{\pi,\boldsymbol{\phi}}(s, a) = \mathbb{E}\left[\sum_{t=0}^{\infty}\boldsymbol{\phi}(s_{t+1})|s_0 = s, a_0 = a, a_{1:\infty} \sim \pi\right] \tag{4.61}$$

$$= \mathbb{E}\left[\boldsymbol{\phi}(s') + \gamma\boldsymbol{\psi}(s', a')\right] \tag{4.62}$$

我们可以通过 TD 规则来学习这个

$$\boldsymbol{\psi}^{\pi}(s, a) \leftarrow \boldsymbol{\psi}^{\pi}(s, a) + \eta\underbrace{\left(\boldsymbol{\phi}(s') + \gamma\boldsymbol{\psi}^{\pi}(s', a') - \boldsymbol{\psi}^{\pi}(s, a)\right)}_{\delta} \tag{4.63}$$

我们可以用它推导出状态 - 动作值函数：

$$Q^{\pi,\boldsymbol{w}}(s) = \boldsymbol{\psi}^{\pi}(s, a)^{\mathsf{T}}\boldsymbol{w} \tag{4.64}$$

这使我们能够通过改变权重向量 来定义多个 Q 函数（以及相应的策略），正如我们在第 4.4.4.1 节中讨论的那样。

4.4.4.1 广义策略改进

到目前为止，我们已经讨论了如何使用现有已知策略的 SFs 来计算新奖励函数的价值函数。在本节中，我们将讨论如何通过使用**广义策略改进**或 **GPI** [Bar+17； Bar+20]。

假设我们已经学习了一组 N （可能最优）策略 π_i 及其对应的 SFs$\boldsymbol{\psi}^{\pi_i}$，以最大化由 $\boldsymbol{w}_i$ 定义的奖励。当面对一个新任务 $\boldsymbol{w}_{\text{new}}$ 时，我们可以使用 GPI 计算一个新的策略，如下所示：

$$a^*(s; \boldsymbol{w}_{\text{new}}) = \operatorname*{argmax}_a \max_i Q^{\pi_i}(s, a, \boldsymbol{w}_{\text{new}}) = \operatorname*{argmax}_a \max_i \boldsymbol{\psi}^{\pi_i}(s, a)^{\mathsf{T}}\boldsymbol{w}_{\text{new}} \tag{4.65}$$

如果 $\boldsymbol{w}_{\text{new}}$ 位于训练任务的范围内（即存在权重 α_i 使得 $\boldsymbol{w}_{\text{new}} \quad _i \alpha_i \boldsymbol{w}_i$），那么 GPI 定理表明 $\pi(a|s) = \mathbb{I}\overline{\sum a} = a^*(s,\boldsymbol{w}_{\text{new}}))$ 将至少与任何现有策略一样表现良好，即 $Q^{\pi}(s,a) \geq \max_i Q^{\pi_i}(s,a)$ （参见第 3.4 节中的策略改进）。见图 4.6 以了解说明。

请注意，GPI 是一种基于现有策略库计算新策略的无模型方法。在 [Ale+23] 中，他们提出了一种扩展，可以利用（可能是近似的）世界模型来学习更好的策略，通过在决策时进行更多搜索，从而超越现有策略库。

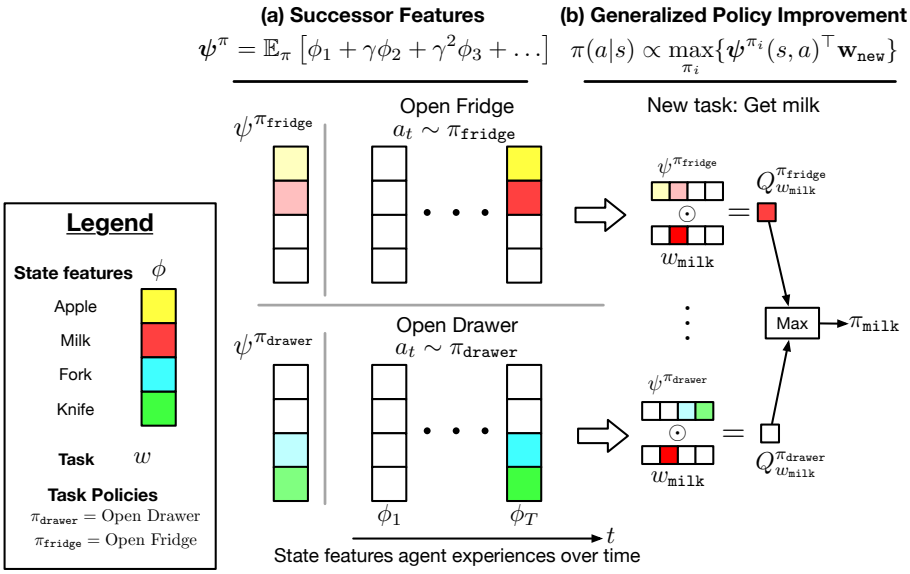


Figure 4.6: Illustration of successor features representation. (a) Here $\boldsymbol{\phi}_t = \boldsymbol{\phi}(s_t)$ is the vector of features for the state at time t, and $\boldsymbol{\psi}^\pi$ is the corresponding SF representation, which depends on the policy π. (b) Given a set of existing policies and their SFs, we can create a new one by specifying a desired weight vector $\boldsymbol{w}_{new}$ and taking a weighted combination of the existing SFs. From Figure 5 of [Car+24]. Used with kind permission of Wilka Carvalho.

4.4.4.2 Option keyboard

One limitation of GPI is that it requires that the reward function, and the resulting policy, be defined in terms of a fixed weight vector $\boldsymbol{w}_{\text{new}}$, where the preference over features is constant over time. However, for some tasks we might want to initially avoid a feature or state and then later move towards it. To solve this, [Bar+19; Bar+20] introduced the **option keyboard**, in which the weight vector for a task can be computed dynamically in a state-dependent way, using $\boldsymbol{w}_s = g(s, \boldsymbol{w}_{\text{new}})$. (Options are discussed in Section 5.3.2.) Actions can then be chosen as follows:

$$a^*(s; \boldsymbol{w}_{\text{new}}) = \operatorname*{argmax}_a \max_i \boldsymbol{\psi}^{\pi_i}(s,a)^\mathsf{T} \boldsymbol{w}_s \tag{4.66}$$

Thus the policy π_i that is chosen depends in the current state. Thus $\boldsymbol{w}_s$ induces a set of policies that are active for a period of time, similar to playing a chord on a piano.

4.4.4.3 Learning SFs

A key question when using SFs is how to learn the cumulants or state-features $\boldsymbol{\phi}(s)$. Various approaches have been suggested, including leveraging meta-gradients [Vee+19], image reconstruction [Mac+18b], and maximizing the mutual information between task encodings and the cumulants that an agent experiences when pursuing that task [Han+19]. The cumulants are encouraged to satisfies the linear reward constraint by minimizing

$$\mathcal{L}_r = ||r - \boldsymbol{\phi}_{\boldsymbol{\theta}}(s)^\mathsf{T} \boldsymbol{w}||_2^2 \tag{4.67}$$

Once the cumulant function is known, we have to learn the corresponding SF. The standard approach learns a different SF for every policy, which is limiting. In [Bor+19] they introduced **Universal Successor Feature Approximators** which takes an input a policy encoding $\boldsymbol{z}_{\boldsymbol{w}}$, representing a policy $\pi_{\boldsymbol{w}}$ (typically we set $\boldsymbol{z}_{\boldsymbol{w}} = \boldsymbol{w}$). We then define

$$\boldsymbol{\psi}^{\pi_w}(s,a) = \boldsymbol{\psi}_{\boldsymbol{\theta}}(s,a,\boldsymbol{z}_{\boldsymbol{w}}) \tag{4.68}$$

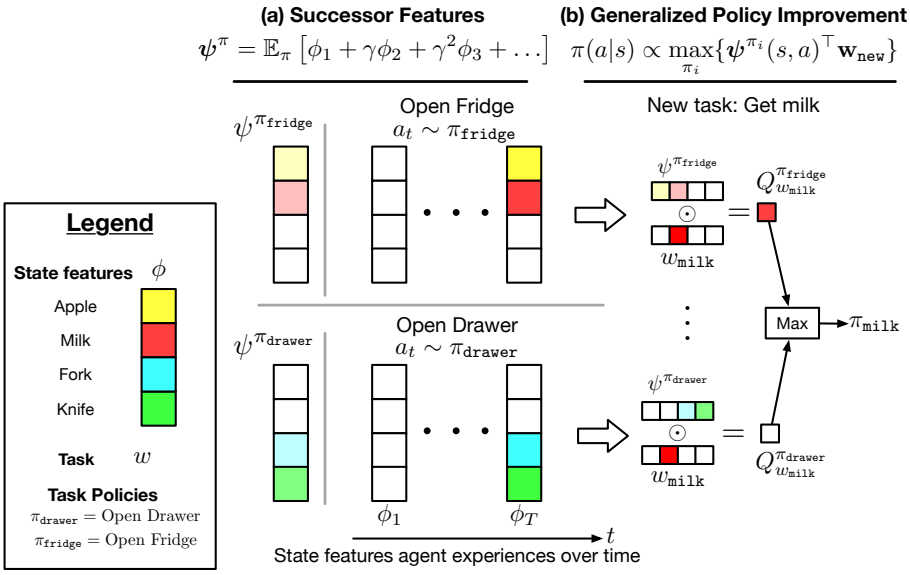


图 4.6：后继特征表示的示意图。（a）这里 $\boldsymbol{\phi}_t = \boldsymbol{\phi}(s_t)$ 是时间 t 状态下特征向量，$\boldsymbol{\psi}^{\pi}$ 是对应的 SF 表示，它依赖于策略 π。（b）给定一组现有策略及其 SF，我们可以通过指定所需的权重向量 $\boldsymbol{w}_{new}$ 并取现有 SF 的加权组合来创建一个新的策略。参见 *[Car+24]* 的第 5 图。经 Wilka Carvalho 许可使用。

4.4.4.2 选项键盘

GPI 的一个限制是，它要求奖励函数和由此产生的策略以固定权重向量 $_{\text{new}}$ 的形式定义，其中特征偏好随时间保持不变。然而，对于某些任务，我们可能希望最初避免某个特征或状态，然后后来逐渐接近它。为了解决这个问题，[Bar+19； Bar+20] 引入了**选项键盘**，其中任务的权重向量可以以状态相关的方式动态计算，使用 $_{\text{new}}$。（选项在 5.3.2 节中讨论。）然后可以选择以下操作：

$$^{\text{new}} \qquad ^{\top} \tag{4.66}$$

因此，所选择的政策 取决于当前状态。因此 诱导出一组在一定时间内有效的政策，类似于在钢琴上弹奏和弦。

4.4.4.3 学习 SFs

在使用 SFs 时，一个关键问题是如何学习累积量或状态特征 。已经提出了各种方法，包括利用元梯度 [Vee+19]、图像重建 [Mac+18b]，以及最大化任务编码与代理在追求该任务时经历的累积量之间的互信息 [Han+19]。累积量通过最小化来鼓励满足线性奖励约束

$$_{\boldsymbol{\theta}} \qquad ^{\top} \tag{4.67}$$

一旦知道了累积量函数，我们就必须学习相应的状态函数（SF）。标准方法为每个策略学习不同的状态函数，这很有限。在 [Bor+19] 他们引入了 **通用后继特征逼近器**，它接受一个策略编码 $_{\boldsymbol{w}}$ 作为输入，代表一个策略 $_{\boldsymbol{w}}$（通常我们设置 $_{\boldsymbol{w}}$ ）。然后我们定义

$$^{\boldsymbol{w}} \qquad _{\boldsymbol{\theta}} \qquad \boldsymbol{w} \tag{4.68}$$

The GPI update then becomes

$$a^*(s; \boldsymbol{w}_{\text{new}}) = \operatorname*{argmax}_a \max_{\boldsymbol{z}_{\boldsymbol{w}}} \boldsymbol{\psi}_{\boldsymbol{\theta}}(s, a, \boldsymbol{z}_{\boldsymbol{w}})^{\mathsf{T}} \boldsymbol{w}_{\text{new}} \tag{4.69}$$

so we replace the discrete max over a finite number of policies with a continuous optimization problem (to be solved per state).

If we want to learn the policies and SFs at the same time, we can optimize the following losses in parallel:

$$\mathcal{L}_Q = ||\boldsymbol{\psi}_{\boldsymbol{\theta}}(s, a, \boldsymbol{z}_w)^{\mathsf{T}} \boldsymbol{w} - \boldsymbol{y}_Q||, \; \boldsymbol{y}_Q = R(s'; \boldsymbol{w}) + \gamma \boldsymbol{\psi}_{\boldsymbol{\theta}}(s', a^*, \boldsymbol{z}_w)^{\mathsf{T}} \boldsymbol{w} \tag{4.70}$$

$$\mathcal{L}_{\boldsymbol{\psi}} = ||\boldsymbol{\psi}_{\boldsymbol{\theta}}(s, a, \boldsymbol{z}_{\boldsymbol{w}}) - \boldsymbol{y}_{\boldsymbol{\psi}}||, \; \boldsymbol{y}_{\boldsymbol{\psi}} = \boldsymbol{\phi}(s') + \gamma \boldsymbol{\psi}_{\boldsymbol{\theta}}(s', a^*, \boldsymbol{z}_{\boldsymbol{w}}) \tag{4.71}$$

where $a^* = \text{argmax}_{a'} \boldsymbol{\psi}_{\boldsymbol{\theta}}(s', a', \boldsymbol{z}_{\boldsymbol{w}})^{\mathsf{T}} \boldsymbol{w}$. The first equation is standard Q learning loss, and the second is the TD update rule in Equation (4.63) for the SF. In [Car+23], they present the **Successor Features Keyboard**, that can learn the policy, the SFs and the task encoding $\boldsymbol{z}_{\boldsymbol{w}}$, all simultaneously. They also suggest replacing the squared error regression loss in Equation (4.70) with a cross-entropy loss, where each dimension of the SF is now a discrete probability distribution over M possible values of the corresponding feature. (c.f. Section 5.1.2).

4.4.4.4 Choosing the tasks

A key advantage of SFs is that they provide a way to compute a value function and policy for any given reward, as specified by a task-specific weight vector $\boldsymbol{w}$. But how do we choose these tasks? In [Han+19] they sample $\boldsymbol{w}$ from a distribution at the start of each task, to encourage the agent to learn to explore different parts of the state space (as specified by the feature function $\boldsymbol{\phi}$). In [LA21] they extend this by adding an intrinsic reward that favors exploring parts of the state space that are surprising (i.e., which induce high entropy), c.f., Section 5.2.4. In [Far+23], they introduce **proto-value networks**, which is a way to define auxiliary tasks based on successor measures.

GPI 更新后变为

$$a^*(s; \boldsymbol{w}_{\text{new}}) = \underset{a}{\text{argmax}} \max_{\boldsymbol{z}_{\boldsymbol{w}}} \boldsymbol{\psi}_{\boldsymbol{\theta}}(s, a, \boldsymbol{z}_{\boldsymbol{w}})^{\mathsf{T}} \boldsymbol{w}_{\text{new}} \tag{4.69}$$

因此，我们将有限数量策略上的离散最大值替换为连续优化问题（按状态解决）。

如果我们想同时学习策略和 SFs，我们可以并行优化以下损失：

$$\mathcal{L}_Q = ||\boldsymbol{\psi}_{\boldsymbol{\theta}}(s, a, \boldsymbol{z}_w)^{\mathsf{T}} \boldsymbol{w} - \boldsymbol{y}_Q||, \; \boldsymbol{y}_Q = R(s'; \boldsymbol{w}) + \gamma \boldsymbol{\psi}_{\boldsymbol{\theta}}(s', a^*, \boldsymbol{z}_w)^{\mathsf{T}} \boldsymbol{w} \tag{4.70}$$

$$\mathcal{L}_{\boldsymbol{\psi}} = ||\boldsymbol{\psi}_{\boldsymbol{\theta}}(s, a, \boldsymbol{z}_{\boldsymbol{w}}) - \boldsymbol{y}_{\boldsymbol{\psi}}||, \; \boldsymbol{y}_{\boldsymbol{\psi}} = \boldsymbol{\phi}(s') + \gamma \boldsymbol{\psi}_{\boldsymbol{\theta}}(s', a^*, \boldsymbol{z}_{\boldsymbol{w}}) \tag{4.71}$$

在 $a^* =$ 中，使用 $\text{argmax}_{a'} \boldsymbol{\psi}_{\boldsymbol{\theta}}(s', a', \boldsymbol{z}_{\boldsymbol{w}})^{\mathsf{T}} \boldsymbol{w}$。第一个方程是标准的 Q 学习损失，第二个是 SF 中的 TD 更新规则（方程 4.63 ）。在 [Car+23] 中，他们提出了 **Successor FeaturesKeyboard**，可以同时学习策略、SF 和任务编码 $\boldsymbol{z}_{\boldsymbol{w}}$。他们还建议将方程（ 4.70 ）中的平方误差回归损失替换为交叉熵损失，其中 SF 的每个维度现在是对应特征 M 可能值的离散概率分布。（参见章节 5.1.2 ）

4.4.4.4 选择任务

强化学习的关键优势在于，它们提供了一种计算任何给定奖励的价值函数和策略的方法，该方法由特定任务的权重向量指定 $\boldsymbol{w}$。但我们是如何选择这些任务的？在 [Han+19] 中，他们在每个任务的开始时从分布中采样 $\boldsymbol{w}$ ，以鼓励智能体学习探索状态空间的不同部分（如特征函数 $\boldsymbol{\phi}$ 指定）。在 [LA21] 中，他们通过添加一个内在奖励来扩展这一方法，该奖励有利于探索状态空间中令人惊讶的部分（即，引起高熵的部分），参见图 5.2.4。在 [Far+23] 中，他们引入了**原型价值网络**，这是一种基于后继度量定义辅助任务的方法。

Chapter 5

Other topics in RL

In this section, we briefly mention some other important topics in RL.

5.1 Distributional RL

The **distributional RL** approach of [BDM17; BDR23], predicts the distribution of (discounted) returns, not just the expected return. More precisely, let $Z^\pi = \sum_{t=0}^{T} \gamma^t r_t$ be a random variable representing the reward-to-go. The standard value function is defined to compute the expectation of this variable: $V^\pi(s) = \mathbb{E}\left[Z^\pi | s_0 = s\right]$. In DRL, we instead attempt to learn the full distribution, $p(Z^\pi | s_0 = s)$. For a general review of distributional regression, see [KSS23]. Below we briefly mention a few algorithms in this class that have been explored in the context of RL.

5.1.1 Quantile regression methods

An alternative to predicting a full distribution is to predict a fixed set of quantiles. This is called quantile regression, and has been used with DQN in [Dab+17] to get **QR-DQN**, and with SAC in [Wur+22] to get **QR-SAC**. (The latter was used in Sony's **GTSophy** Gran Turismo AI racing agent.)

5.1.2 Replacing regression with classification

An alternative to quantile regression is to approximate the distribution over returns using a histogram, and then fit it using cross entropy loss (see Figure 5.1). This approach was first suggested in [BDM17], who called it **categorical DQN**. (In their paper, they use 51 discrete categories (atoms), giving rise to the name **C51**.)

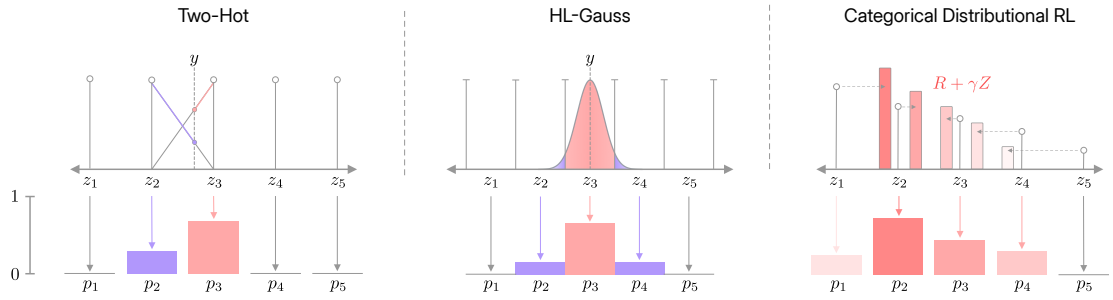


Figure 5.1: Illustration of how to encode a scalar target y or distributional target Z using a categorical distribution. From Figure 1 of [Far+24]. Used with kind permission of Jesse Farebrother.

第五章

其他强化学习中的主题

在这个部分，我们简要介绍了 RL 中的一些其他重要主题。

5.1 分布式强化学习

BDM17 的分布式强化学习**方法；** BDR 预测（折现）回报的分布，而不仅仅是期望回报。更准确地说，设为表示即时奖励的随机变量。标准价值函数被定义为计算这个变量的期望：[() 。在 DRL 中，我们试图学习完整的分布，() 。关于分布回归的综述，请参阅 KSS23], 。以下我们简要介绍了一些在这个类别中已被探索的算法在 RL 中的应用。

5.1.1 分位数回归方法

预测整个分布的替代方法是预测一组固定的分位数。这被称为分位数回归，并且已经在 [Dab+17] 中与 DQN 结合使用，以获得 **QR-DQN**，以及在 [Wur+22] 中与 SAC 结合使用，以获得 **QR-SAC**。（后者被用于索尼的 **GTSophy** Gran Turismo AI 赛车代理。）

5.1.2 将回归替换为分类

使用直方图对收益分布进行近似，然后使用交叉熵损失进行拟合，这是量回归的一种替代方法（见图 5.1 ）。这种方法最初由 [BDM 17], 提出，他们将其称为**分类 DQN**。（在他们的论文中，他们使用了 51 个离散类别（原子），从而产生了 **C51** 这个名称。）

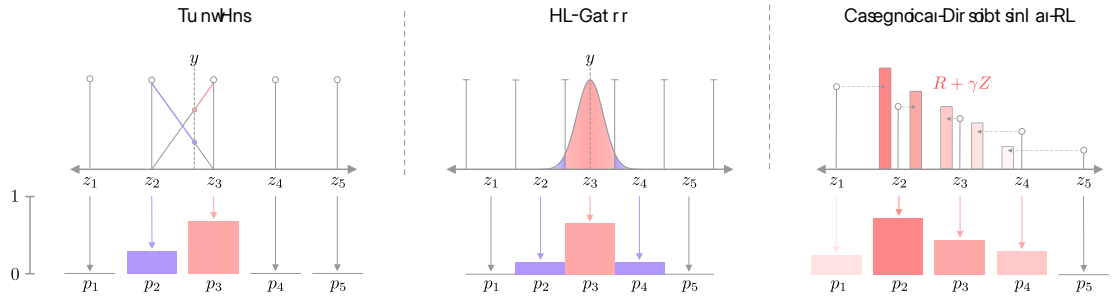


图 5.1：如何使用分类分布对标量目标 y 或分布性目标 Z 进行编码的说明。来自 [*Far+24*] 的第 1 图。经 Jesse Farebrother 许可使用。

An even simpler approach is to replace the distributional target with the standard scalar target (representing the mean), and then discretize this target and use cross entropy loss instead of squared error.[1] Unfortunately, this encoding is lossy. In [Sch+20], they proposed the **two-hot** transform, that is a lossless encoding of the target based on putting appropriate weight on the nearest two bins (see Figure 5.1). In [IW18], they proposed the **HL-Gauss** histogram loss, that convolves the target value y with a Gaussian, and then discretizes the resulting continuous distribution. This is more symetric than two-hot encoding, as shown in Figure 5.1. Regardless of how the discrete target is chosen, predictions are made using $\hat{y}(s;\boldsymbol{\theta}) = \sum_k p_k(s) b_k$, where $p_k(s)$ is the probability of bin k, and b_k is the bin center.

In [Far+24], they show that the HL-Gauss trick works much better than MSE, two-hot and C51 across a variety of problems (both offline and online), especially when they scale to large networks. They conjecture that the reason it beats MSE is that cross entropy is more robust to noisy targets (e.g., due to stochasticity) and nonstationary targets. They also conjecture that the reason HL works better than two-hot is that HL is closer to ordinal regression, and reduces overfitting by having a softer (more entropic) target distribution (similiar to label smoothing in classification problems).

5.2 Reward functions

Sequential decision making relies on the user to define the reward function in order to encourage the agent to exhibit some desired behavior. In this section, we discuss this crucial aspect of the problem.

5.2.1 Reward hacking

In some cases, the reward function may be misspecified, so even though the agent may maximize the reward, this might turn out not to be what the user desired. For example, suppose the user rewards the agent for making as many paper clips as possible. An optimal agent may convert the whole world into a paper clip factory, because the user forgot to specify various constraints, such as not killing people or not destroying the environment. In the **AI alignment** community, this example is known as the **paperclip maximizer problem**, and is due to Nick Bostrom [Bos16]. (See e.g., https://openai.com/index/faulty-reward-functions/ for some examples that have occurred in practice.) This is an example of a more general problem known as **reward hacking** [Ska+22]. For a potential solution, based on the assistance game paradigm, see Section 5.6.1.2.

5.2.2 Sparse reward

Even if the reward function is correct, optimizing it is not always easy. In particular, many problems suffer from **sparse reward**, in which $R(s,a) = 0$ for almost all states and actions, so the agent only every gets feedback (either positive or negative) on the rare occasions when it achieves some unknown goal. This requires **deep exploration** [Osb+19] to find the rewarding states. One approach to this is use to use PSRL (Section 1.4.4.2). However, various other heuristics have been developed, some of which we discuss below.

5.2.3 Reward shaping

In **reward shaping**, we add prior knowledge about what we believe good states should look like, as a way to combat the difficulties of learning from sparse reward. That is, we define a new reward function $r' = r + F$, where F is called the shaping function. In general, this can affect the optimal policy. For example, if a soccer playing agent is "artificially" rewarded for making contact with the ball, it might learn to repeatedly touch and untouch the ball (toggling between s and s'), rather than trying to win the original game. But in [NHR99], the prove that if the shaping function has the form

$$F(s,a,s') = \gamma \Phi(s') - \Phi(s) \tag{5.1}$$

[1] Technically speaking, this is no longer a distributional RL method, since the prediction target is the mean, but the mechanism for predicting the mean leverages a distribution, for robustness and ease of optimization.

一种更简单的方法是将分布目标替换为标准标量目标（表示均值），然后对此目标进行离散化，并使用交叉熵损失代替平方误差。[1] 遗憾的是，这种编码是有损的。在 [Sch+20] 中，他们提出了 **两种热**转换，这是一种基于对最近的两个箱进行适当加权的无损失目标编码（见图 5.1 ）。在 [IW18]，中，他们提出了 **HL-Gauss** 直方图损失，该损失将目标值 y 与高斯卷积，然后对得到的连续分布进行离散化。这比两种热编码更对称，如图 5.1 所示。无论如何选择离散目标，都使用 $\hat{y}(s;\boldsymbol{\theta}) = \sum_k p_k(s) b_k$ 进行预测，其中 $p_k(s)$ 是箱 k 的概率，而 b_k 是箱中心。

在 [Far+24]，他们表明 HL-Gauss 技巧在各种问题（离线和在线）上比 MSE 、二进制热编码和 C51 表现得更好，尤其是在它们扩展到大型网络时。他们推测，它之所以能打败 MSE，是因为交叉熵对噪声目标（例如，由于随机性）和非平稳目标更鲁棒。他们还推测，HL 之所以比二进制热编码表现得更好，是因为 HL 更接近序数回归，并通过具有更软（更熵）的目标分布来减少过拟合（类似于分类问题中的标签平滑）。

5.2 奖励函数

顺序决策依赖于用户定义奖励函数，以鼓励智能体表现出某些期望的行为。在本节中，我们讨论了该问题的这一关键方面。

5.2.1 奖励黑客行为

在某些情况下，奖励函数可能被错误指定，因此即使智能体可能最大化奖励，这最终可能并非用户所期望的。例如，假设用户奖励智能体尽可能多地制造回形针。一个最优的智能体可能会将整个世界变成回形针工厂，因为用户忘记指定各种约束，例如不杀人或不破坏环境。在 AI 对齐社区中，这个例子被称为 “ 回形针最大化问题 ”，归功于尼克 · 博斯特罗姆。这是更一般问题的一个例子，称为 “ 奖励黑客 ”。关于一个基于辅助游戏范式的潜在解决方案，请参阅第 5.6.1.2 节。

5.2.2 稀疏奖励

即使奖励函数是正确的，优化它也并不总是容易。特别是，许多问题都受到稀疏奖励的影响，其中（对于几乎所有的状态和动作） s （ a ） $= 0$ ，因此智能体只有在实现一些未知目标时才会得到反馈（无论是积极的还是消极的）。这需要深入探索以找到有奖励的状态。一种方法是使用 PSRL （第 1.4.4.2 节）。然而，已经开发出各种其他启发式方法，其中一些我们将在下面讨论。

5.2.3 奖励塑造

在 **奖励塑造**中，我们添加了我们认为良好状态应该是什么样的先验知识，作为对抗从稀疏奖励中学习的困难的一种方式。也就是说，我们定义了一个新的奖励函数 $r' = r + F$，其中 F 被称为塑造函数。一般来说，这可以影响最优策略。例如，如果一个踢足球的智能体因为 “ 人为地 ” 奖励与球接触而得到奖励，它可能会学会反复触摸和接触球（在 s 和 s' 之间切换），而不是尝试赢得原始游戏。但在 [NHR99]，中证明，如果塑造函数具有形式

$$F(s, a, s') = \gamma \Phi(s') - \Phi(s) \tag{5.1}$$

从技术上讲，这不再是一种分布式强化学习方法，因为预测目标是均值，但预测均值的机制利用了分布，以提高鲁棒性和优化简便性。

where $\Phi : \mathcal{S} \to \mathbb{R}$ is a **potential function**, then we can guarantee that the sum of shaped rewards will match the sum of original rewards plus a constant. This is called **Potential-Based Reward Shaping**.

In [Wie03], they prove that (in the tabular case) this approach is equivalent to initializing the value function to $V(s) = \Phi(s)$. In [TMM19], they propose an extension called potential-based advice, where they show that a potential of the form $F(s, a, s', a') = \gamma\Phi(s', a') - \Phi(s, a)$ is also valid (and more expressive). In [Hu+20], they introduce a reward shaping function z which can be used to down-weight or up-weight the shaping function:

$$r'(s, a) = r(s, a) + z_\phi(s, a)F(s, a) \tag{5.2}$$

They use bilevel optimization to optimize ϕ wrt the original task performance.

5.2.4 Intrinsic reward

When the extrinsic reward is sparse, it can be useful to (also) reward the agent for solving "generally useful" tasks, such as learning about the world. This is called **intrinsically motivated RL** [AMH23; Lin+19; Ami+21; Yua22; Yua+24; Col+22]. It can be thought of as a special case of reward shaping, where the shaping function is dynamically computed.

We can classify these methods into two main types: **knowledge-based intrinsic motivation**, or **artificial curiosity**, where the agent is rewarded for learning about its environment; and **competence-based intrinsic motivation**, where the agent is rewarded for achieving novel goals or mastering new skills.

5.2.4.1 Knowledge-based intrinsic motivation

One simple approach to knowledge-based intrinsic motivation is to add to the extrinsic reward an intrinsic **exploration bonus** $R^i_t(s_t)$, which is high when the agent visits novel states. For tabular environments, we can just count the number of visits to each state, $N_t(s)$, and define $R^i_t(s) = 1/N_t(s)$ or $R^i_t(s) = 1/\sqrt{N_t(s)}$, which is similar to the UCB heuristic used in bandits (see Section 1.4.3). We can extend exploration bonuses to high dimensional states (e.g. images) using density models [Bel+16]. Alternatively, [MBB20] propose to use the ℓ_1 norm of the successor feature (Section 4.4.4) representation as an alternative to the visitation count, giving rise to an intrinsic reward of the form $R^i(s) = 1/||\boldsymbol{\psi}^\pi(s)||_1$. Recently [Yu+23] extended this to combine SFs with *predecessor* representations, which encode retrospective information about the previous state (c.f., inverse dynamics models, mentioned below). This encourages exploration towards bottleneck states.

Another approach is the **Random Network Distillation** or **RND** method of [Bur+18]. This uses a fixed random neural network feature extractor $\boldsymbol{z}_t = f(\boldsymbol{s}_t; \boldsymbol{\theta}^*)$ to define a target, and then trains a predictor $\hat{\boldsymbol{z}}_t = f(\boldsymbol{s}_t; \hat{\boldsymbol{\theta}}_t)$ to predict these targets. If $\boldsymbol{s}_t$ is similar to previously seen states, then the trained model will have low prediction error. We can thus define the intrinsic reward as proportional to the squared error $||\hat{\boldsymbol{z}}_t - \boldsymbol{z}_t||_2^2$. The **BYOL-Explore** method of [Guo+22b] goes beyond RND by learning the target representation (for the next state), rather than using a fixed random projection, but is still based on prediction error.

We can also define an intrinsic reward in terms of the information theoretic **surprise** of the next state given the current one:

$$R(\boldsymbol{s}, \boldsymbol{a}, \boldsymbol{s}') = -\log q(\boldsymbol{s}'|\boldsymbol{s}, \boldsymbol{a}) \tag{5.3}$$

This is the same as methods based on rewarding states for prediction error. Unfortunately such methods can suffer from the **noisy TV problem** (also called a **stochastic trap**), in which an agent is attracted to states which are intrinsically to predict. To see this, note that by averaging over future states we see that the above reward reduces to

$$R(\boldsymbol{s}, \boldsymbol{a}) = -\mathbb{E}_{p^*(\boldsymbol{s}'|\boldsymbol{s},\boldsymbol{a})}\left[\log q(\boldsymbol{s}'|\boldsymbol{s}, \boldsymbol{a})\right] = \mathbb{H}_{ce}(p^*, q) \tag{5.4}$$

where p^* is the true model and q is the learned dynamics model, and $\mathbb{H}_{ce}$ is the cros -entropy. As we learn the optimal model, $q = p^*$, this reduces to the conditional entropy of the predictive distribution, which can be non-zero for inherently unpredictable states.

Φ: $\mathcal{S} \to \mathbb{R}$ 是一个**潜在函数**，那么我们可以保证形状奖励的总和将等于原始奖励的总和加上一个常数。这被称为**基于潜在函数的奖励塑造**。

在 [Wie03]，中，他们证明（在表格情况下）这种方法与将值函数初始化为 $V(s) = \Phi(s)$ 等价。在 [TMM19]，中，他们提出了一种称为基于势能的建议的扩展，其中他们表明形式为 $F(s, a, s', a') = \gamma\Phi(s', a') - \Phi(s, a)$ 的势能也是有效的（并且更具表现力）。在 [Hu+20] 中，他们介绍了一种奖励塑造函数 z，该函数可用于调整塑造函数的权重：

$$r'(s, a) = r(s, a) + z_\phi(s, a)F(s, a) \tag{5.2}$$

他们使用双级优化来优化与原始任务性能相关的 ϕ。

5.2.4 内在奖励

当外部奖励稀疏时，奖励智能体解决“通常有用”的任务，如了解世界，可能是有用的。这被称为内在动机强化学习；Lin； Ami； Yua22； Yua； Col。它可以被视为奖励塑造的特殊情况，其中塑造函数是动态计算的。

我们可以将这些方法分为两大类：**基于知识的内在动机**，或**人工好奇心**，其中智能体因了解其环境而获得奖励；以及**基于能力的内在动机**，其中智能体因实现新目标或掌握新技能而获得奖励。

5.2.4.1 基于知识的内在动机

基于知识的内在动机的一个简单方法是在外在奖励中添加一个内在**探索奖励** R_t^i（s_t），当智能体访问新状态时，该奖励值较高。对于表格环境，我们只需计算每个状态的访问次数 N_t（s），并定义 R_t^i（s）$= 1/N_t$（s）或 R_t^i（s）$= 1/\sqrt{N_t}$（s）），这与在多臂老虎机中使用的 UCB 启发式方法类似（见第 1.4.3 节）。我们可以通过密度模型 [Bel+16] 将探索奖励扩展到高维状态（例如图像）。或者，[MBB20] 提出使用后继特征（第 4.4.4 节）表示的 ℓ_1 范数作为访问次数的替代，从而产生一种形式的内在奖励 R^i（s）$= 1/||\boldsymbol{\psi}^\pi$（$s$）$||_1$。最近 [Yu+23] 将此扩展到结合 SFs 与前驱表示，这些表示编码了关于先前状态的反向信息（例如，逆动力学模型，如下所述）。这鼓励探索瓶颈状态。

另一种方法是**随机网络蒸馏**或 **RND** 方法 [Bur+18]。这种方法使用一个固定的随机神经网络特征提取器 $\boldsymbol{z}_t = f(\boldsymbol{s}_t; \boldsymbol{\theta}^*)$ 来定义目标，然后训练一个预测器 $\hat{\boldsymbol{z}}_t = f(\boldsymbol{s}_t; \hat{\boldsymbol{\theta}}_t)$ 来预测这些目标。如果 $\boldsymbol{s}_t$ 与之前看到的州相似，那么训练好的模型将具有较低的预测误差。因此，我们可以将内在奖励定义为与平方误差 $||\hat{\boldsymbol{z}}_t - \boldsymbol{z}_t||_2^2$ 成比例。**BYOL-Explore** 方法 [Guo+22b] 通过学习目标表示（对于下一个状态），而不是使用固定的随机投影，超越了 RND，但仍基于预测误差。

我们也可以用当前状态给定下个状态的信息论**惊喜**来定义内在奖励：

$$R(\boldsymbol{s}, \boldsymbol{a}, \boldsymbol{s}') = -\log q(\boldsymbol{s}'|\boldsymbol{s}, \boldsymbol{a}) \tag{5.3}$$

这是与基于奖励状态预测误差的方法相同。不幸的是，这种方法可能会受到**噪声 TV 问题**（也称为**随机陷阱**）的影响，其中智能体会被吸引到本质上难以预测的状态。为了理解这一点，请注意，通过对未来状态的平均，我们发现上述奖励简化为

$$R(\boldsymbol{s}, \boldsymbol{a}) = -\mathbb{E}_{p^*(\boldsymbol{s}'|\boldsymbol{s},\boldsymbol{a})}\left[\log q(\boldsymbol{s}'|\boldsymbol{s}, \boldsymbol{a})\right] = \mathbb{H}_{ce}(p^*, q) \tag{5.4}$$

p^* 是真实模型，q 是学习到的动力学模型，$\mathbb{H}_{ce}$ 是交叉熵。当我们学习最优模型 $q = p^*$ 时，这降低到预测分布的条件熵，对于本质上不可预测的状态，这可以是非零的。

To help filter out such random noise, [Pat+17] proposes an **Intrinsic Curiosity Module**. This first learns an **inverse dynamics model** of the form $a = f(\boldsymbol{s}, \boldsymbol{s}')$, which tries to predict which action was used, given that the agent was in $\boldsymbol{s}$ and is now in $\boldsymbol{s}'$. The classifier has the form $\text{softmax}(g(\phi(\boldsymbol{s}), \phi(s'), a))$, where $\boldsymbol{z} = \phi(\boldsymbol{s})$ is a representation function that focuses on parts of the state that the agent can control. Then the agent learns a forwards dynamics model in $\boldsymbol{z}$-space. Finally it defines the intrinsic reward as

$$R(\boldsymbol{s}, \boldsymbol{a}, \boldsymbol{s}') = -\log q(\phi(\boldsymbol{s}')|\phi(\boldsymbol{s}), a) \tag{5.5}$$

Thus the agent is rewarded for visiting states that lead to unpredictable consequences, where the difference in outcomes is measured in a (hopefully more meaningful) latent space.

Another solution is to replace the cross entropy with the KL divergence, $R(\boldsymbol{s}, \boldsymbol{a}) = \mathrm{D}_{\mathbb{KL}}(p||q) = \mathbb{H}_{ce}(p, q) - \mathbb{H}(p)$, which goes to zero once the learned model matches the true model, even for unpredictable states. This has the desired effect of encouraging exploration towards states which have epistemic uncertainty (reducible noise) but not aleatoric uncertainty (irreducible noise) [MP+22]. The **BYOL-Hindsight** method of [Jar+23] is one recent approach that attempts to use the $R(\boldsymbol{s}, \boldsymbol{a}) = \mathrm{D}_{\mathbb{KL}}(p||q)$ objective. Unfortunately, computing the $\mathrm{D}_{\mathbb{KL}}(p||q)$ term is much harder than the usual variational objective of $\mathrm{D}_{\mathbb{KL}}(q||p)$. A related idea, proposed in the RL context by [Sch10], is to use the **information gain** as a reward. This is defined as $R_t(\boldsymbol{s}_t, \boldsymbol{a}_t) = \mathrm{D}_{\mathbb{KL}}(q(\boldsymbol{s}_t|\boldsymbol{h}_t, \boldsymbol{a}_t, \boldsymbol{\theta}_t)||q(\boldsymbol{s}_t|\boldsymbol{h}_t, \boldsymbol{a}_t, \boldsymbol{\theta}_{t-1})$, where $\boldsymbol{h}_t$ is the history of past observations, and $\boldsymbol{\theta}_t = \text{update}(\boldsymbol{\theta}_{t-1}, \boldsymbol{h}_t, \boldsymbol{a}_t, \boldsymbol{s}_t)$ are the new model parameters. This is closely related to the BALD (Bayesian Active Learning by Disagreement) criterion [Hou+11; KAG19], and has the advantage of being easier to compute, since it is does not reference the true distribution p.

5.2.4.2 Goal-based intrinsic motivation

We will discuss goal-conditioned RL in Section 5.3.1. If the agent creates its own goals, then it provides a way to explore the environment. The question of when and how an agent to switch to pursuing a new goal is studied in [Pis+22] (see also [BS23]). Some other key work in this space includes the **scheduled auxiliary control** method of [Rie+18], and the **Go Explore** algorithm in [Eco+19; Eco+21] and its recent LLM extension [LHC24].

5.3 Hierarchical RL

So far we have focused on MDPs that work at a single time scale. However, this is very limiting. For example, imagine planning a trip from San Francisco to New York: we need to choose high level actions first, such as which airline to fly, and then medium level actions, such as how to get to the airport, followed by low level actions, such as motor commands. Thus we need to consider actions that operate multiple levels of **temporal abstraction**. This is called **hierarchical RL** or **HRL**. This is a big and important topic, and we only brief mention a few key ideas and methods. Our summary is based in part on [Pat+22]. (See also Section 4.4 where we discuss multi-step predictive models; by contrast, in this section we focus on model-free methods.)

5.3.1 Feudal (goal-conditioned) HRL

In this section, we discuss an approach to HRL known as **feudal RL** [DH92]. Here the action space of the higher level policy consists of **subgoals** that are passed down to the lower level policy. See Figure 5.2 for an illustration. The lower level policy learns a **universal policy** $\pi(a|s, g)$, where g is the goal passed into it [Sch+15a]. This policy optimizes an MDP in which the reward is define as $R(s, a|g) = 1$ iff the goal state is achieved, i.e., $R(s, a|s) = \mathbb{I}(s = g)$. (We can also define a dense reward signal using some state abstraction function ϕ, by defininng $R(s, a|g) = \text{sim}(\phi(s), \phi(g))$ for some similarity metric.) This approach to RL is known as **goal-conditioned RL** [LZZ22].

为了帮助过滤掉这种随机噪声，[Pat+17] 提出了一种**内在好奇心模块**。它首先学习一个**逆动力学模型**，形式为 $a = f(\boldsymbol{s}, \boldsymbol{s}')$，试图预测在代理处于 $\boldsymbol{s}$ 时使用了哪种动作，现在处于 $\boldsymbol{s}'$。分类器形式为 $\text{softmax}(g(\phi(\boldsymbol{s})\phi(s')a))$，其中 $\boldsymbol{z} = \phi(\boldsymbol{s})$ 是一个关注代理可以控制的状态部分的表示函数。然后代理在 $\boldsymbol{z}$- 空间中学习一个正向动力学模型。最后它定义内在奖励为

$$R(\boldsymbol{s}, \boldsymbol{a}, \boldsymbol{s}') = -\log q(\phi(\boldsymbol{s}')|\phi(\boldsymbol{s}), a) \tag{5.5}$$

因此，代理被奖励访问导致不可预测后果的状态，其中结果差异在（希望更有意义的）潜在空间中衡量。

另一种解决方案是用 KL 散度替换交叉熵，$R(\boldsymbol{s}, \boldsymbol{a}) = \mathrm{D}_{\mathbb{KL}}(p||q) = \mathbb{H}_{ce}(p, q) - \mathbb{H}(p)$，一旦学习到的模型与真实模型匹配，即使对于不可预测的状态，它也会趋近于零。这具有鼓励探索具有认知不确定性（可减少噪声）但不是随机不确定性（不可减少噪声）的状态的预期效果 [MP+22]。**BYOL-Hindsight** 方法 [Jar+23] 是最近的一种尝试使用 $R(\boldsymbol{s},\boldsymbol{a}) = \mathrm{D}_{\mathbb{KL}}(p||q)$ 目标的方法。不幸的是，计算 $\mathrm{D}_{\mathbb{KL}}(p||q)$ 项比通常的变分目标 $\mathrm{D}_{\mathbb{KL}}(q||p)$要困难得多。一个相关想法，由[Sch10], 在RL环境中提出，是使用**信息增益**作为奖励。这被定义为 $R_t(\boldsymbol{s}_t, \boldsymbol{a}_t) = \mathrm{D}_{\mathbb{KL}}(q(\boldsymbol{s}_t|\boldsymbol{h}_t, \boldsymbol{a}_t, \boldsymbol{\theta}_t)||q(\boldsymbol{s}_t|\boldsymbol{h}_t, \boldsymbol{a}_t, \boldsymbol{\theta}_{t-1})$，其中 $\boldsymbol{h}_t$ 是过去观察的历史，而 $\boldsymbol{\theta}_t = \text{update}(\boldsymbol{\theta}_{t-1}, \boldsymbol{h}_t, \boldsymbol{a}_t, \boldsymbol{s}_t)$ 是新的模型参数。这与 BALD （基于差异的贝叶斯主动学习）标准 [Hou+11； KAG19]，密切相关，并且具有计算更简单的优点，因为它不参考真实分布 p。

5.2.4.2 基于目标的内在动机

我们将在第 5.3.1 节讨论目标条件 RL。如果智能体创建自己的目标，那么它提供了一种探索环境的方法。关于智能体何时以及如何切换到追求新目标的问题在 [Pis+22] （参见 [BS23] ）中进行了研究。这个领域的一些其他关键工作包括 **R ie** 的调度辅助控制 [方法，以及 +18]Eco **中的** Go Explore [算法； +19Eco 及其最近的 LLM 扩展 +21] LHC[LHC24]。

5.3 层次化强化学习

到目前为止，我们主要关注在单一时间尺度上工作的 MDPs。然而，这非常有限。例如，想象一下从旧金山到纽约的旅行计划：我们首先需要选择高级动作，比如选择哪家航空公司，然后是中级动作，比如如何到达机场，最后是低级动作，比如电机命令。因此，我们需要考虑操作多个时间抽象层次的动作。这被称为**分层强化学习**或 **HRL**。这是一个大而重要的主题，我们只简要介绍了一些关键思想和方法。我们的总结部分基于 [Pat+22]。（另见第 4.4 节，其中我们讨论了多步预测模型；相比之下，在本节中，我们专注于无模型方法。）

5.3.1 封建（目标条件）强化学习

在本节中，我们讨论了一种称为**封建 RL** [DH92] 的 HRL 方法。在这里，高级策略的动作空间由**子目标**组成，这些目标被传递给低级策略。见图 5.2 ，以了解说明。低级策略学习一个**通用策略** $\pi(a|s,g)$，其中 g 是传递给它的目标 [Sch+15a]。此策略优化一个 MDP，其中奖励定义为 $R(s,a|g) = 1$ ，如果达到目标状态，即 $R(s, a|s) = \mathbb{I}(s = g)$。(我们还可以使用某种状态抽象函数 ϕ 定义密集奖励信号，通过定义 $R(s,a|g) = \text{sim}(\phi(s)\ \phi(g))$ ，用于某些相似度度量。) 这种 RL 方法被称为**目标条件 RL**[LZZ22]。

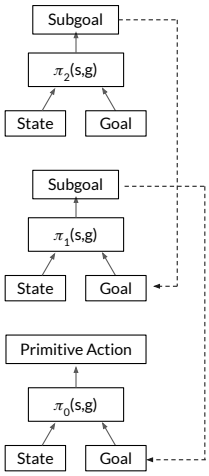


Figure 5.2: Illustration of a 3 level hierarchical goal-conditioned controller. From http://bigai.cs.brown.edu/2019/09/03/hac.html. Used with kind permission of Andrew Levy.

5.3.1.1 Hindsight Experience Relabeling (HER)

In this section, we discuss an approach to efficiently learning goal-conditioned policies, in the special case where the set of goal states $\mathcal{G}$ is the same as the set of original states $\mathcal{S}$. We will extend this to the hierarchical case below.

The basic idea is as follows. We collect various trajectores in the environment, from a starting state s_0 to some terminal state s_T, and then define the goal of each trajectory as being $g = s_T$; this trajectory then serves as a demonstration of how to achieve this goal. This is called **hindsight experience relabeling** or **HER** [And+17]. This can be used to relabel the trajectories stored in the replay buffer. That is, if we have $(s, a, R(s|g), s', g)$ tuples, we replace them with $(s, a, R(s|g'), g')$ where $g' = s_T$. We can then use any off-policy RL method to learn $\pi(a|s, g)$. In [Eys+20], they show that HER can be viewed as a special case of maximum-entropy inverse RL, since it is estimating the reward for which the corresponding trajectory was optimal.

5.3.1.2 Hierarchical HER

We can leverage HER to learn a hierarchical controller in several ways. In [Nac+18] they propose **HIRO** (Hierarchical Reinforcement Learning with Off-policy Correction) as a way to train a two-level controller. (For a two-level controller, the top level is often called the **manager**, and the low level the **worker**.) The data for the manager are transition tuples of the form $(s_t, g_t, \sum r_{t:t+c}, s_{t+c})$, where c is the time taken for the worker to reach the goal (or some maximum time), and r_t is the main task reward function at step t. The data for the worker are transition tuples of the form $(s_{t+i}, g_t, a_{t+i}, r^{g_t}_{t+i}, s_{t+i+1})$ for $i = 0 : c$, where r^g_t is the reward wrt reaching goal g. This data can be used to train the two policies. However, if the worker fails to achieve the goal in the given time limit, all the rewards will be 0, and no learning will take place. To combat this, if the worker does not achieve g_t after c timesteps, the subgoal is relabeled in the transition data with another subgoal g'_t which is sampled from $p(g|\tau)$, where τ is the observed trajectory. Thus both policies treat g'_t as the goal in hindsight, so they can use the actually collected data for training

The **hierarchical actor critic** (HAC) method of [Lev+18] is a simpler version of HIRO that can be extended to multiple levels of hierarchy, where the lowest level corresponds to primitive actions (see Figure 5.2). In the HAC approach, the output subgoal in the higher level data, and the input subgoal in the lower-level data, are replaced with the actual state that was achieved in hindsight. This allows the training of each level of the hierarchy independently of the lower levels, by assuming the lower level policies are already optimal (since they achieved the specified goal). As a result, the distribution of (s, a, s') tuples experienced by a higher level

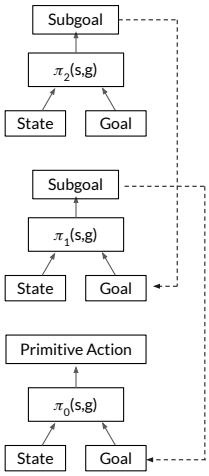


图 5.2：3 级层次目标条件控制器的示意图。来自 *http://bigai.cs.brown.edu/2019/09/03/hac.html*。经 Andrew Levy 许可使用。

5.3.1.1 事后经验重标记（HER）

在这一节中，我们讨论了一种在目标状态集 $\mathcal{G}$ 与原始状态集 $\mathcal{S}$ 相同的情况下，高效学习目标条件策略的方法。我们还将将其扩展到下面的分层情况。

基本思想如下。我们在环境中收集各种轨迹，从起始状态 s_0 到某个终止状态 s_T，然后定义每条轨迹的目标为 $g = s_T$；这条轨迹随后作为实现该目标的示范。这被称为 **事后经验重标记**或 **HER** [以及 +17]。这可以用来重新标记存储在重放缓冲区中的轨迹。也就是说，如果我们有 $(s, a, R(s|g)\, s', g)$ 元组，我们将它们替换为 $(s, a, R(s|g')\, g')$，其中 $g' = s_T$。然后我们可以使用任何离线策略强化学习方法来学习 $\pi(a|s,g)$。在 [Eys+20] 中，他们表明 HER 可以被视为最大熵逆强化学习的一个特例，因为它是在估计对应轨迹最优的奖励。

5.3.1.2 层次化 HER

我们可以利用 HER 以多种方式学习分层控制器。在 [Nac+18] 中，他们提出了 **HIRO**（基于离线策略校正的分层强化学习）作为训练两级控制器的方法。（对于两级控制器，顶层通常被称为**经理**，底层被称为**工人**。）经理的数据是形式为 $(s_t, g_t, \sum r_{t:t+c}, s_{t+c})$ 的转换元组，其中 c 是工人达到目标所需的时间（或某些最大时间），而 r_t 是第 t 步的主要任务奖励函数。工人的数据是形式为 $(s_{t+i}, g_t, a_{t+i}, r^{g_t}_{t+i}, s_{t+i+1})$ 的转换元组，用于 $i = 0 : c$，其中 r^g_t是相对于达到目标的奖励 g。这些数据可以用来训练两个策略。然而，如果工人在给定的时间限制内未能达到目标，所有奖励都将为 0，并且不会发生学习。为了应对这种情况，如果工人在 g_t 后 c 个时间步内未能达到 g'_t，则将子目标重新标记为转换数据中的另一个子目标，该子目标从 $p(g|\tau)$ 中采样，其中 τ 是观察到的轨迹。因此，两个策略都将 g'_t 视为事后目标，因此它们可以使用实际收集到的数据进行训练

Lev+18] 的分层演员评论员（HAC）方法是一个 HIRO 的简化版本，可以扩展到多个层次，其中最低层对应原始动作（见图 5.2）。在 HAC 方法中，高级别数据中的输出子目标和低级别数据中的输入子目标被实际达到的状态所取代。这使得可以独立于低级别层次训练每个级别的层次，通过假设低级别策略已经是最优的（因为它们已经实现了指定的目标）。因此，高级别层次所经历的 (s, a, s') 元组

will be stable, providing a stationary learning target. By contrast, if all policies are learned simultaneously, the distribution becomes **non-stationary**, which makes learning harder. For more details, see the paper, or the corresponding blog post (with animations) at http://bigai.cs.brown.edu/2019/09/03/hac.html.

5.3.1.3 Learning the subgoal space

In the previous approaches, the subgoals are defined in terms of the states that were achieved at the end of each trajectory, $g' = s_T$. This can be generalized by using a state abstraction function to get $g' = \phi(s_T)$. The methods in Section 5.3.1.2) assumed that ϕ was manually specified. We now mention some ways to learn ϕ.

In [Vez+17], they present **Feudal Networks** for learning a two level hierarchy. The manager samples subgoals in a learned latent subgoal space. The worker uses distance to this subgoal as a reward, and is trained in the usual way. The manager uses the "transition gradient" as a reward, which is derived from the task reward as well as the distance between the subgoal and the actual state transition made by the worker. This reward signal is used to learn the manager policy and the latent subgoal space.

Feudal networks do not guarantee that the learned subgoal space will result in optimal behavior. In [Nac+19], they present a method to optimize the policy and ϕ function so as to minimize a bound on the suboptimality of the hierarchical policy. This approach is combined with HIRO (Section 5.3.1.2) to tackle the non-stationarity issue.

5.3.2 Options

The feudal approach to HRL is somewhat limited, since not all subroutines or skills can be defined in terms of reaching a goal state (even if it is a partially specified one, such as being in a desired location but without specifying the velocity). For example, consider the skill of "driving in a circle", or "finding food". The **options** framework is a more general framework for HRL first proposed in [SPS99]. We discuss this below.

5.3.2.1 Definitions

An option $\omega = (I, \pi, \beta)$ is a tuple consisting of: the **initiation set** $I_\omega \subset S$, which is a subset of states that this option can start from (also called the **affordances** of each state [Khe+20]); the **subpolicy** $\pi_\omega(a|s) \in [0, 1]$; and the **termination condition** $\beta_\omega(s) \in [0, 1]$, which gives the probability of finishing in state s. (This induces a geometric distribution over option durations, which we denote by $\tau \sim \beta_\omega$.) The set of all options is denoted Ω.

To execute an option at step t entails choosing an action using $a_t = \pi_\omega(s_t)$ and then deciding whether to terminate at step $t + 1$ with probability $1 - \beta_\omega(s_{t+1})$ or to continue following the option at step $t + 1$. (This is an example of a **semi-Markov decision process** [Put94].) If we define $\pi_\omega(s) = a$ and $\beta_\omega(s) = 0$ for all s, then this option corresponds to primitive action a that terminates in one step. But with options we can expand the repertoire of actions to include those that take many steps to finish.

To create an MDP with options, we need to define the reward function and dynamics model. The reward is defined as follows:

$$R(s, \omega) = \mathbb{E}\left[R_1 + \gamma R^2 + \cdots + \gamma^{\tau-1} R_\tau | S_0 = s, A_{0:\tau-1} \sim \pi_\omega, \tau \sim \beta_\omega\right] \tag{5.6}$$

The dynamics model is defined as follows:

$$p_\gamma(s'|s, \omega) = \sum_{k=1}^{\infty} \gamma^k \Pr\left(S_k = s', \tau = k | S_0 = s, A_{0:k-1} \sim \pi_\omega, \tau \sim \beta_\omega\right) \tag{5.7}$$

Note that $p_\gamma(s'|s, \omega)$ is not a conditional probability distribution, because of the γ^k term, but we can usually treat it like one. Note also that a dynamics model that can predict multiple steps ahead is sometimes called a **jumpy model** (see also Section 4.4.3.2).

将保持稳定，提供一个静态的学习目标。相比之下，如果所有策略同时学习，分布将变为**非静态**，这使得学习更加困难。更多细节请参阅论文，或查看相应的博客文章（带有动画） http://bigai.cs.brown.edu/2019/09/03/hac.html。

5.3.1.3 学习子目标空间

在先前的方法中，子目标是通过每个轨迹结束时达到的状态来定义的，$g' = s_T$。这可以通过使用状态抽象函数来泛化，以获得 $g' = \phi(s_T)$。第 5.3.1.2 节中的方法假设 ϕ 是手动指定的。我们现在介绍一些学习方法 ϕ。

在 [Vez+17]，他们提出了用于学习两级层次结构的 封建网络。经理在学习的潜在子目标空间中采样子目标。工人使用到这个子目标的距离作为奖励，并以通常的方式进行训练。经理使用 “ 转换梯度 ” 作为奖励，该奖励来自任务奖励以及子目标与工人实际状态转换之间的距离。这个奖励信号用于学习经理策略和潜在子目标空间。

封建网络不能保证学习到的子目标空间会导致最优行为。在 Nac 中，他们提出了一种优化策略和函数的方法，以最小化分层策略次优性的界限。这种方法与 HIRO （第 5.3.1.2 节）相结合，以解决非平稳性问题。

5.3.2 选项

对 HRL 的封建方法有些局限，因为并非所有子例程或技能都可以用达到目标状态来定义（即使这个状态是部分指定的，比如在期望的位置但没有指定速度）。例如，考虑 “ 圆形驾驶 ” 或 “ 寻找食物 ” 的技能。**选项**框架是 HRL 的一个更通用的框架，首次在 [SPS99] 中提出。我们将在下面讨论这个问题。

5.3.2.1 定义

一个选项 $\omega = (I, \pi, \beta)$ 是一个元组，由以下部分组成：**初始化集合** $I_\omega \subset S$，它是可以从该选项开始的状态的子集（也称为每个状态的 **能力** Khe+20] ）；**子策略** $\pi_\omega(a|s) \in [0,1]$；以及**终止条件** $\beta_\omega(s) \in [0,1]$，它给出了在状态 s 结束的概率。（这诱导了一个关于选项持续时间的几何分布，我们用 $\tau \sim \beta_\omega$ 表示。）所有选项的集合表示为 Ω。

在步骤 t 执行选项涉及使用 $a_t = \pi_\omega(s_t)$ 选择一个动作，然后决定是否在步骤 $t+1$ 以概率 $1 - \beta_\omega(s_{t+1})$ 终止，或者继续在步骤 $t+1$ 跟随该选项。（这是一个 **半马尔可夫决策过程的例子** [Put94]。）如果我们定义 $\pi_\omega(s) = a$ 和 $\beta_\omega(s) = 0$ 对所有 s，那么这个选项对应于在一步终止的基本动作 a。但是，通过选项，我们可以扩展动作库，包括那些需要多步才能完成的动作。

为了创建具有选项的 MDP，我们需要定义奖励函数和动态模型。奖励定义为以下内容：

$$R(s,\omega) = \mathbb{E}\left[R_1 + \gamma R^2 + \cdots + \gamma^{\tau-1} R_\tau | S_0 = s, A_{0:\tau-1} \sim \pi_\omega, \tau \sim \beta_\omega\right] \tag{5.6}$$

动力学模型定义为如下：

$$p_\gamma(s'|s,\omega) = \sum_{k=1}^{\infty} \gamma^k \Pr\left(S_k = s', \tau = k | S_0 = s, A_{0:k-1} \sim \pi_\omega, \tau \sim \beta_\omega\right) \tag{5.7}$$

请注意，$p_\gamma(s'|s,\omega)$ 不是一个条件概率分布，因为存在 γ^k 项，但我们通常可以将其视为一个。还要注意，能够预测多个步骤的动力学模型有时被称为**跳跃模型**（也可参见第 4.4.3.2 节）。

We can use these definitions to define the value function for a hierarchical policy using a generalized Bellman equation, as follows:

$$V_\pi(s) = \sum_{\omega \in \Omega(s)} \pi(\omega|s) \left[R(s,\omega) + \sum_{s'} p_\gamma(s'|s,\omega) V_\pi(s') \right] \tag{5.8}$$

We can compute this using value iteration. We can then learn a policy using policy iteration, or a policy gradient method. In other words, once we have defined the options, we can use all the standard RL machinery.

Note that GCRL can be considered a special case of options where each option corresponds to a different goal. Thus the reward function has the form $R(s,\omega) = \mathbb{I}(s = \omega)$, the termination function is $\beta_\omega(s) = \mathbb{I}(s = \omega)$, and the initiation set is the entire state space.

5.3.2.2 Learning options

The early work on options, including the **MAXQ** approach of [Die00], assumed that the set of options was manually specified. Since then, many methods for learning options have been proposed. We mention a few of these below.

The first set of methods for option learning rely on two stage training. In the first stage, exploration methods are used to collect trajectories. Then this data is analysed, either by inferring hidden segments using EM applied to a latent variable model [Dan+16], or by using the **skill chaining** method of [KB09], which uses classifiers to segment the trajectories. The labeled data can then be used to define a set of options, which can be trained using standard methods.

The second set of methods for option learning use end-to-end training, i.e., the options and their policies are jointly learned online. For example, [BHP17] propose the **option-critic** architecture. The number of options is manually specified, and all policies are randomly initialized. Then they are jointly trained using policy gradient methods designed for semi-MDPs. (See also [RLT18] for a hierarchical extension of option-critic to support options calling options.) However, since the learning signal is just the main task reward, the method can work poorly in problems with sparse reward compared to subgoal methods (see discussion in [Vez+17; Nac+19]).

Another problem with option-critic is that it requires specialized methods that are designed for optimizing semi-MDPs. In [ZW19], they propose **double actor critic**, which allows the use of standard policy gradient methods. This works by defining two parallel **augmented MDPs**, where the state space of each MDP is the cross-product of the original state space and the set of options. The manager learns a policy over options, and the worker learns a policy over states for each option. Both MDPs just use task rewards, without subgoals or subtask rewards.

It has been observed that option learning using option-critic or double actor-critic can fail, in the sense that the top level controller may learn to switch from one option to the next at almost every time step [ZW19; Har+18]. The reason is that the optimal policy does not require the use of temporally extended options, but instead can be defined in terms of primitive actions (as in standard RL). Therefore in [Har+18] they propose to add a regularizer called the **deliberation cost**, in which the higher level policy is penalized whenever it switches options. This can speed up learning, at the cost of a potentially suboptimal policy.

Another possible failure mode in option learning is if the higher level policy selects a single option for the entire task duration. To combat this, [KP19] propose the **Interest Option Critic**, which learns the initiation condition I_ω so that the option is selected only in certain states of interest, rather than the entire state space.

In [Mac+23], they discuss how the successor representation (discussed in Section 4.4) can be used to define options, using a method they call the **Representation-driven Option Discovery** (ROD) cycle.

In [Lin+24b] they propose to represent options as programs, which are learned using LLMs.

我们可以使用这些定义来使用广义贝尔曼方程定义层次策略的价值函数，如下所示：

$$V_\pi(s) = \sum_{\omega \in \Omega(s)} \pi(\omega|s) \left[R(s,\omega) + \sum_{s'} p_\gamma(s'|s,\omega) V_\pi(s') \right] \tag{5.8}$$

我们可以使用值迭代来计算这个。然后我们可以使用策略迭代或策略梯度方法来学习策略。换句话说，一旦我们定义了选项，我们就可以使用所有标准的强化学习工具。

请注意，GCRL 可以被视为一种特殊情况，其中每个选项对应不同的目标。因此，奖励函数的形式为 $R(s,\omega) = \mathbb{I}(s = \omega)$，终止函数是 $\beta_\omega(s) = \mathbb{I}(s = \omega)$，而初始化集是整个状态空间。

5.3.2.2 学习选项

早期对选项的研究，包括 **MAXQ** 方法，假设选项集是手动指定的。从那时起，已经提出了许多学习选项的方法。以下我们提到其中的一些。

选项学习方法的第一套方法依赖于两阶段训练。在第一阶段，使用探索方法收集轨迹。然后分析这些数据，要么通过将 EM 应用于潜在变量模型来推断隐藏段 [Dan+16]，要么使用**技能链**方法 [KB09]，，该方法使用分类器来分割轨迹。然后可以使用标记数据来定义一组选项，这些选项可以使用标准方法进行训练。

选项学习的第二种方法使用端到端训练，即在线联合学习选项及其策略。例如，[BHP17] 提出了**选项 - 评论家**架构。选项的数量是手动指定的，所有策略都是随机初始化的。然后它们使用为半 MDP 设计的策略梯度方法联合训练。（有关选项 - 评论家分层扩展以支持选项调用选项的讨论，请参阅 [RLT18] 。）然而，由于学习信号仅仅是主要任务的奖励，与子目标方法相比，在稀疏奖励的问题中，该方法可能表现不佳（参见 [Vez+17;Nac+19] ）的讨论。）

选项批评的另一个问题是它需要为优化半马尔可夫决策过程（ MDPs ）而设计的专用方法。在 [ZW 19]，他们提出了**双演员批评**，这使得可以使用标准的策略梯度方法。这是通过定义两个并行**增强的 MDPs** 来实现的，其中每个 MDP 的状态空间是原始状态空间与选项集合的笛卡尔积。管理者学习一个关于选项的策略，而工人学习每个选项的状态策略。这两个 MDP 仅使用任务奖励，没有子目标或子任务奖励。

观察到使用选项 - 评论家或双演员 - 评论家进行选项学习可能会失败，即在几乎每个时间步，顶级控制器都可能学会从一种选项切换到另一种选项。原因是最佳策略不需要使用时间扩展的选项，而可以用原始动作来定义（如标准强化学习）。因此，在 Har 他们提出添加一个称为 “ 深思熟虑成本 ” 的正则化器，其中每当高级策略切换选项时，就会对其进行惩罚。这可以加快学习速度，但可能会以策略次优为代价。

在选项学习中，另一种可能的故障模式是，高级策略在整个任务持续期间选择单个选项。为了解决这个问题，[KP 19] 提出了**兴趣选项评论家**，它学习启动条件 I_ω ，以便仅在感兴趣的某些状态下选择选项，而不是整个状态空间。

在 [Mac+23] 中，他们讨论了如何使用后继表示法（在第 4.4 节中讨论）来定义选项，使用他们称之为**表示法驱动的选项发现**（ ROD ）循环的方法。在 [Lin+24b] 他们提出将选项表示为程序，这些程序是可学习的 使用 LLM 的 Ned。

5.4 Imitation learning

In previous sections, an RL agent is to learn an optimal sequential decision making policy so that the total reward is maximized. **Imitation learning** (IL), also known as **apprenticeship learning** and **learning from demonstration** (LfD), is a different setting, in which the agent does not observe rewards, but has access to a collection $\mathcal{D}_{\text{exp}}$ of trajectories generated by an expert policy π_{exp}; that is, $\boldsymbol{\tau} = (s_0, a_0, s_1, a_1, \ldots, s_T)$ and $a_t \sim \pi_{\text{exp}}(s_t)$ for $\boldsymbol{\tau} \in \mathcal{D}_{\text{exp}}$. The goal is to learn a good policy by imitating the expert, in the absence of reward signals. IL finds many applications in scenarios where we have demonstrations of experts (often humans) but designing a good reward function is not easy, such as car driving and conversational systems. (See also Section 5.5, where we discuss the closely related topic of offline RL, where we also learn from a collection of trajectories, but no longer assume they are generated by an optimal policy.)

5.4.1 Imitation learning by behavior cloning

A natural method is **behavior cloning**, which reduces IL to supervised learning; see [Pom89] for an early application to autonomous driving. It interprets a policy as a classifier that maps states (inputs) to actions (labels), and finds a policy by minimizing the imitation error, such as

$$\min_{\pi} \mathbb{E}_{p^{\gamma}_{\pi_{\text{exp}}}(s)} \left[D_{\mathbb{KL}} \left(\pi_{\text{exp}}(s) \parallel \pi(s) \right) \right] \tag{5.9}$$

where the expectation wrt $p^{\gamma}_{\pi_{\text{exp}}}$ may be approximated by averaging over states in $\mathcal{D}_{\text{exp}}$. A challenge with this method is that the loss does not consider the sequential nature of IL: future state distribution is not fixed but instead depends on earlier actions. Therefore, if we learn a policy $\hat{\pi}$ that has a low imitation error under distribution $p^{\gamma}_{\pi_{\text{exp}}}$, as defined in Equation (5.9), it may still incur a large error under distribution $p^{\gamma}_{\hat{\pi}}$ (when the policy $\hat{\pi}$ is actually run). This problem has been tackled by the offline RL literature, which we discuss in Section 5.5.

5.4.2 Imitation learning by inverse reinforcement learning

An effective approach to IL is **inverse reinforcement learning** (IRL) or **inverse optimal control** (IOC). Here, we first infer a reward function that "explains" the observed expert trajectories, and then compute a (near-)optimal policy against this learned reward using any standard RL algorithms studied in earlier sections. The key step of reward learning (from expert trajectories) is the opposite of standard RL, thus called inverse RL [NR00].

It is clear that there are infinitely many reward functions for which the expert policy is optimal, for example by several optimality-preserving transformations [NHR99]. To address this challenge, we can follow the maximum entropy principle, and use an energy-based probability model to capture how expert trajectories are generated [Zie+08]:

$$p(\boldsymbol{\tau}) \propto \exp\Big(\sum_{t=0}^{T-1} R_{\boldsymbol{\theta}}(s_t, a_t) \Big) \tag{5.10}$$

where $R_{\boldsymbol{\theta}}$ is an unknown reward function with parameter $\boldsymbol{\theta}$. Abusing notation slightly, we denote by $R_{\boldsymbol{\theta}}(\boldsymbol{\tau}) = \sum_{t=0}^{T-1} R_{\boldsymbol{\theta}}(s_t, a_t))$ the cumulative reward along the trajectory $\boldsymbol{\tau}$. This model assigns exponentially small probabilities to trajectories with lower cumulative rewards. The partition function, $Z_{\boldsymbol{\theta}} \triangleq \int_{\boldsymbol{\tau}} \exp(R_{\boldsymbol{\theta}}(\boldsymbol{\tau}))$, is in general intractable to compute, and must be approximated. Here, we can take a sample-based approach. Let $\mathcal{D}_{\text{exp}}$ and $\mathcal{D}$ be the sets of trajectories generated by an expert, and by some known distribution q, respectively. We may infer $\boldsymbol{\theta}$ by maximizing the likelihood, $p(\mathcal{D}_{\text{exp}}|\boldsymbol{\theta})$, or equivalently, minimizing the negative log-likelihood loss

$$\mathcal{L}(\boldsymbol{\theta}) = -\frac{1}{|\mathcal{D}_{\text{exp}}|} \sum_{\boldsymbol{\tau} \in \mathcal{D}_{\text{exp}}} R_{\boldsymbol{\theta}}(\boldsymbol{\tau}) + \log \frac{1}{|\mathcal{D}|} \sum_{\boldsymbol{\tau} \in \mathcal{D}} \frac{\exp(R_{\boldsymbol{\theta}}(\boldsymbol{\tau}))}{q(\boldsymbol{\tau})} \tag{5.11}$$

5.4 模仿学习

在前面的章节中，RL 智能体需要学习一个最优的顺序决策策略，以使总奖励最大化。模仿学习（ IL ），也称为**学徒学习**和从演示中学习（ LfD ），是一个不同的设置，其中智能体不观察奖励，但可以访问由专家策略生成的轨迹集合 $\mathcal{D}_{\text{exp}}$ ；也就是说，$\boldsymbol{\tau} = （ s_0, a_0, s_1, a_1, \ldots, s_T ）$ 和 $a_t \sim \pi_{\text{exp}} （ s_t ）$ 用于 $\boldsymbol{\tau} \in \mathcal{D}_{\text{exp}}$。目标是通过对专家的模仿来学习一个好的策略，在没有奖励信号的情况下。IL 在许多场景中都有应用，在这些场景中，我们有专家（通常是人类）的演示，但设计一个好的奖励函数并不容易，例如汽车驾驶和对话系统。（另见第 5.5 节，其中我们讨论了与离线 RL 密切相关的主题，在离线 RL 中，我们也从轨迹集合中学习，但不再假设它们是由最优策略生成的。）

5.4.1 通过行为克隆的模仿学习

一种自然的方法是**行为克隆**，它将 IL 降低为监督学习；参见 [Pom89] ，了解其在自动驾驶中的早期应用。它将策略解释为将状态（输入）映射到动作（标签）的分类器，并通过最小化模仿误差来找到策略，例如

$$\min_{\pi} \mathbb{E}_{p^{\gamma}_{\pi_{\text{exp}}}(s)} \left[D_{\mathbb{KL}} \left(\pi_{\text{exp}}(s) \parallel \pi(s) \right) \right] \tag{5.9}$$

其中，关于 $p^{\gamma}_{\pi_{\text{exp}}}$ 的期望可以通过在 $\mathcal{D}_{\text{exp}}$ 中的状态上平均来近似。这种方法的一个挑战是损失函数没有考虑 IL 的顺序性：未来状态分布不是固定的，而是依赖于早期动作。因此，如果我们学习一个在分布 下具有低模仿误差的策略 $\hat{\pi}$ ，如方程（ 5.9 ）中定义的，那么在分布 $p^{\gamma}_{\hat{\pi}}$ （当策略 $\hat{\pi}$ 实际运行时）下仍可能产生较大的误差。这个问题已经被离线强化学习文献所解决，我们将在第 5.5 节中讨论。

5.4.2 通过逆强化学习进行模仿学习

IL 的有效方法有**逆强化学习**（ IRL ）或**逆最优控制**（ IOC ）。在这里，我们首先推断出一个 “ 解释 ” 观察到的专家轨迹的奖励函数，然后使用前面章节中研究过的任何标准强化学习算法来计算针对这个学习到的奖励的（近）最优策略。奖励学习（从专家轨迹中）的关键步骤与标准强化学习相反，因此被称为逆强化学习 [NR00]。

显然，存在无限多个奖励函数，专家策略对于这些函数都是最优的，例如通过几个保持最优性的变换 [NHR99]。为了应对这一挑战，我们可以遵循最大熵原理，并使用基于能量的概率模型来捕捉专家轨迹是如何生成的 [Zie+08]:

$$p(\boldsymbol{\tau}) \propto \exp\Big(\sum_{t=0}^{T-1} R_{\boldsymbol{\theta}}(s_t, a_t) \Big) \tag{5.10}$$

$R_{\boldsymbol{\theta}}$ 是一个具有参数 $\boldsymbol{\theta}$ 的未知奖励函数。略微滥用符号，我们用 $R_{\boldsymbol{\theta}}(\boldsymbol{\tau})$ 表示沿轨迹 $\boldsymbol{\tau}$ 的累积奖励 $= \sum^{T-1}_{t=0} R_{\boldsymbol{\theta}}(s_t, a_t))$。此模型将指数级小的概率分配给累积奖励较低的轨迹。配分函数 $Z_{\boldsymbol{\theta}} \triangleq \int_{\boldsymbol{\tau}} \exp(R_{\boldsymbol{\theta}}(\boldsymbol{\tau}))$ 通常难以计算，必须进行近似。在这里，我们可以采用基于样本的方法。设 $\mathcal{D}_{\text{exp}}$ 和 $\mathcal{D}$ 分别为专家生成和某些已知分布 q 生成的轨迹集合。我们可以通过最大化似然 $\boldsymbol{\theta}$ 或等价地，最小化负对数似然损失 $p(\mathcal{D}_{\text{exp}}|\boldsymbol{\theta})$ 来推断 $\boldsymbol{\theta}$ 。

$$\mathcal{L}(\boldsymbol{\theta}) = -\frac{1}{|\mathcal{D}_{\text{exp}}|} \sum_{\boldsymbol{\tau} \in \mathcal{D}_{\text{exp}}} R_{\boldsymbol{\theta}}(\boldsymbol{\tau}) + \log \frac{1}{|\mathcal{D}|} \sum_{\boldsymbol{\tau} \in \mathcal{D}} \frac{\exp(R_{\boldsymbol{\theta}}(\boldsymbol{\tau}))}{q(\boldsymbol{\tau})} \tag{5.11}$$

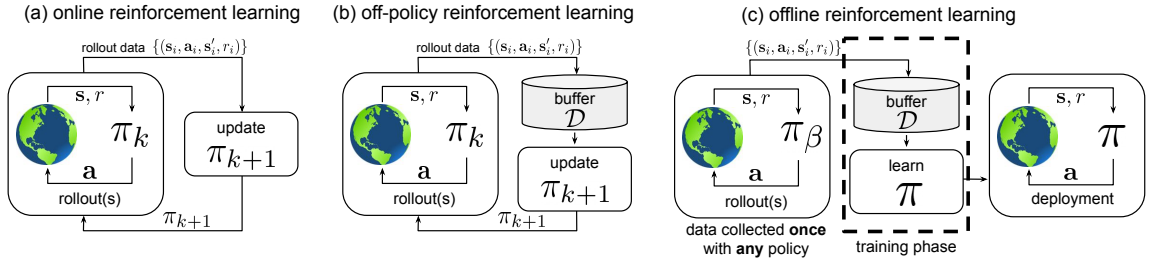


Figure 5.3: Comparison of online on-policy RL, online off-policy RL, and offline RL. From Figure 1 of [Lev+20a]. Used with kind permission of Sergey Levine.

The term inside the log of the loss is an importance sampling estimate of Z that is unbiased as long as $q(\boldsymbol{\tau}) > 0$ for all $\boldsymbol{\tau}$. However, in order to reduce the variance, we can choose q adaptively as $\boldsymbol{\theta}$ is being updated. The optimal sampling distribution, $q_*(\boldsymbol{\tau}) \propto \exp(R_{\boldsymbol{\theta}}(\boldsymbol{\tau}))$, is hard to obtain. Instead, we may find a policy $\hat{\pi}$ which induces a distribution that is close to q_*, for instance, using methods of maximum entropy RL discussed in Section 1.5.3. Interestingly, the process above produces the inferred reward $R_{\boldsymbol{\theta}}$ as well as an approximate optimal policy $\hat{\pi}$. This approach is used by **guided cost learning** [FLA16], and found effective in robotics applications.

5.4.3 Imitation learning by divergence minimization

We now discuss a different, but related, approach to IL. Recall that the reward function depends only on the state and action in an MDP. It implies that if we can find a policy π, so that $p^\gamma_\pi(s,a)$ and $p^\gamma_{\pi_{\text{exp}}}(s,a)$ are close, then π receives similar long-term reward as π_{exp}, and is a good imitation of π_{exp} in this regard. A number of IL algorithms find π by minimizing the divergence between p^γ_π and $p^\gamma_{\pi_{\text{exp}}}$. We will largely follow the exposition of [GZG19]; see [Ke+19] for a similar derivation.

Let f be a convex function, and D_f be the corresponding f-divergence [Mor63; AS66; Csi67; LV06; CS04]. From the above intuition, we want to minimize $\mathrm{D}_\mathrm{f}\left(p^\gamma_{\pi_{\text{exp}}} \middle\| p^\gamma_\pi\right)$. Then, using a variational approximation of D_f [NWJ10], we can solve the following optimization problem for π:

$$\min_\pi \max_{\boldsymbol{w}} \mathbb{E}_{p^\gamma_{\pi_{\text{exp}}}(s,a)}\left[T_{\boldsymbol{w}}(s,a)\right] - \mathbb{E}_{p^\gamma_\pi(s,a)}\left[f^*(T_{\boldsymbol{w}}(s,a))\right] \tag{5.12}$$

where f^* is the convex conjugate of f, and $T_{\boldsymbol{w}} : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$ is some function parameterized by $\boldsymbol{w}$. We can think of π as a generator (of actions) and $T_{\boldsymbol{w}}$ as an adversarial critic that is used to compare the generated (s,a) pairs to the real ones. Thus the first expectation can be estimated using $\mathcal{D}_{\text{exp}}$, as in behavior cloning, and the second can be estimated using trajectories generated by policy π. Furthermore, to implement this algorithm, we often use a parametric policy representation $\pi_{\boldsymbol{\theta}}$, and then perform stochastic gradient updates to find a saddle-point to Equation (5.12). With different choices of the convex function f, we can obtain many existing IL algorithms, such as **generative adversarial imitation learning** (**GAIL**) [HE16] and **adversarial inverse RL** (**AIRL**) [FLL18], etc.

5.5 Offline RL

Offline reinforcement learning (also called **batch reinforcement learning** [LGR12]) is concerned with learning a reward maximizing policy from a fixed, static dataset, collected by some existing policy, known as the **behavior policy**. Thus no interaction with the environment is allowed (see Figure 5.3). This makes policy learning harder than the online case, since we do not know the consequences of actions that were not taken in a given state, and cannot test any such "counterfactual" predictions by trying them. (This is the same problem as in off-policy RL, which we discussed in Section 3.5.) In addition, the policy will be deployed

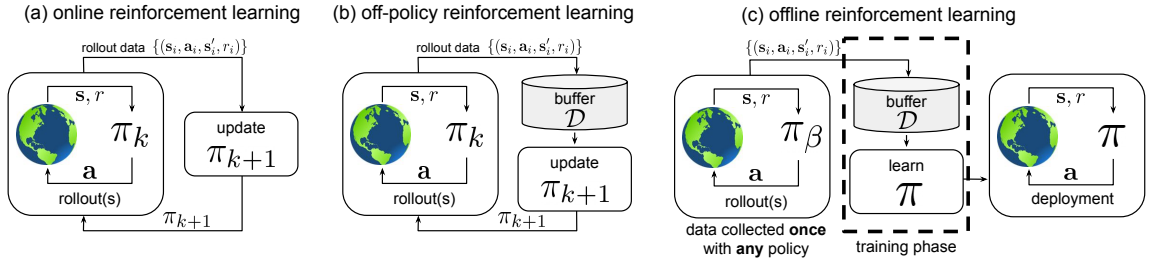


图 5.3：在线策略强化学习、在线非策略强化学习和离线强化学习的比较。来自 [Lev+20a] 的第 1 图。经 Sergey Levine 许可使用。

损失函数中的日志是一个重要性采样估计 Z，只要 $q(\boldsymbol{\tau}) > 0$ 对所有 $\boldsymbol{\tau}$ 都是无偏的。然而，为了减少方差，我们可以在 q 更新时自适应地选择 $\boldsymbol{\theta}$。最优采样分布 $q_*(\boldsymbol{\tau}) \propto \exp(R_{\boldsymbol{\theta}}(\boldsymbol{\tau}))$ 难以获得。相反，我们可能找到一个策略 $\hat{\pi}$，该策略诱导的分布接近 q_*，例如，使用第 1.5.3 节中讨论的最大熵 RL 方法。有趣的是，上述过程产生了推断奖励 $R_{\boldsymbol{\theta}}$ 以及一个近似最优策略 $\hat{\pi}$。这种方法被**引导成本学习** [FLA16]，使用，并在机器人应用中证明有效。

5.4.3 通过发散最小化进行模仿学习

我们现在讨论一种不同但相关的 IL 方法。回想一下，在 MDP 中，奖励函数只依赖于状态和动作。这意味着如果我们能找到一个策略 π，使得 $p^\gamma_\pi(s,a)$ 和 $p^\gamma_{\pi_{\text{exp}}}(s,a)$ 接近，那么 π 将获得与 π_{exp} 相似的长期奖励，并且在这方面是 π_{exp} 的良好模仿。许多 IL 算法通过最小化 π 之间的差异来找到 p^γ_π 和 $p^\gamma_{\pi_{\text{exp}}}$。我们将主要遵循 GZG[的阐述；参见 Ke[进行类似的推导。

设 f 为一个凸函数，且 D_f 为其相应的 f- 散度 [Mor63； AS66； Csi67； LV06； CS04]。从上述直觉出发，我们希望最小化 $\mathrm{D}_\mathrm{f}\left(p^\gamma_{\pi_{\text{exp}}} \middle\| p^\gamma_\pi\right)$。然后，利用 D_f [NWJ10]，的变分近似，我们可以为 π 求解以下优化问题：

$$\min_\pi \max_{\boldsymbol{w}} \mathbb{E}_{p^\gamma_{\pi_{\text{exp}}}(s,a)}\left[T_{\boldsymbol{w}}(s,a)\right] - \mathbb{E}_{p^\gamma_\pi(s,a)}\left[f^*(T_{\boldsymbol{w}}(s,a))\right] \tag{5.12}$$

其中 f^* 是 f 的凸共轭，而 $T_{\boldsymbol{w}}{:}\mathcal{S} \times \mathcal{A} \to \mathbb{R}$ 是由 $\boldsymbol{w}$ 参数化的某个函数。我们可以将 π 视为一个生成器（动作），而将 $T_{\boldsymbol{w}}$ 视为一个用于将生成的（s,a）与真实值进行比较的对抗性评论家。因此，第一个期望可以使用 $\mathcal{D}_{\text{exp}}$ 来估计，就像在行为克隆中一样，而第二个期望可以使用由策略 π 生成的轨迹来估计。此外，为了实现此算法，我们通常使用参数化策略表示 $\pi_{\boldsymbol{\theta}}$，然后执行随机梯度更新以找到方程（5.12）的鞍点。通过不同的凸函数 f 选择，我们可以获得许多现有的 IL 算法，例如 **生成对抗性模仿学习**（**GAIL**）[HE16] 和**对抗性逆强化学习**（**AIRL**）[FLL18]，等。

5.5 离线强化学习

离线强化学习（也称为 **批量强化学习** [LGR12]）涉及从固定的、静态的数据集中学习一个最大化奖励的策略，这些数据集是由某些现有的策略收集的，称为 **行为策略**。因此，不允许与环境进行交互（见图 5.3）。这使得策略学习比在线情况更困难，因为我们不知道在给定状态下未采取的动作的后果，并且无法通过尝试来测试任何这样的“反事实”预测。（这与我们在第 3.5 节中讨论的离策略 RL 中的问题相同。）此外，策略将被部署

on new states that it may not have seen, requiring that the policy generalize out-of-distribution, which is the main bottleneck for current offline RL methods [Par+24b].

A very simple and widely used offline RL method is known as behavior cloning or BC. This amounts to training a policy to predict the observed output action a_t associated with each observed state s_t, so we aim to ensure $\pi(s_t) \approx a_t$, as in supervised learning. This assumes the offline dataset was created by an expert, and so falls under the umbrella of imitation learning (see Section 5.4.1 for details). By contrast, offline RL methods can leverage suboptimal data. We give a brief summary of some of these methods below. For more details, see e.g., [Lev+20b; Che+24b; Cet+24]. For some offline RL benchmarks, see DR4L [Fu+20], RL Unplugged [Gul+20], OGBench (Offline Goal-Conditioned benchmark) [Par+24a], and D5RL [Raf+24].

5.5.1 Offline model-free RL

In principle, we can tackle offline RL using the off-policy methods that we discussed in Section 3.5. These use some form of importance sampling, based on $\pi(a|s)/\pi_b(a|s)$, to reweight the data in the replay buffer $\mathcal{D}$, which was collected by the behavior policy, towards the current policy (the one being evaluated/ learned). Unfortunately, such methods only work well if the behavior policy is is close to the new policy. In the online RL case, this can be ensured by gradually updating the new policy away from the behavior policy, and then sampling new data from the updated policy (which becomes the new behavior policy). Unfortunately, this is not an option in the offline case. Thus we need to use other strategies to control the discrepancy between the behavior policy and learned policy, as we discuss below. (Besides the algorithmic techniques we discuss, another reliable way to get better offline RL performance is to train on larger, more diverse datasets, as shown in [Kum+23].)

5.5.1.1 Policy constraint methods

In the **policy constraint** method, we use a modified form of actor-critic, which, at iteration k, uses an update of the form

$$Q^\pi_{k+1} \leftarrow \operatorname*{argmin}_Q \mathbb{E}_{(s,a,s')\sim\mathcal{D}} \left[\left(Q(s,a) - (R(s,a) + \gamma \mathbb{E}_{\pi_k(a'|s')} \left[Q^\pi_k(s',a') \right]) \right)^2 \right] \tag{5.13}$$

$$\pi_{k+1} \leftarrow \operatorname*{argmax}_\pi \mathbb{E}_{s\sim\mathcal{D}} \left[\mathbb{E}_{\pi(a|s)} \left[Q^\pi_{k+1}(s,a) \right] \right] \quad \text{s.t.} \quad D(\pi, \pi_b) \le \epsilon \tag{5.14}$$

where $D(\pi(\cdot|s), \pi_b(\cdot|s))$ is a divergence measure on distributions, such as KL divergence or another f-divergence. This ensures that we do not try to evaluate the Q function on actions a' that are too dissimilar from those seen in the data buffer (for each sampled state s), which might otherwise result in artefacts similar an adversarial attack.

As an alternative to adding a constraint, we can add a penalty of $\alpha D(\pi(\cdot|s), \pi_b(\cdot|s))$ to the target Q value and the actor objective, resulting in the following update:

$$Q^\pi_{k+1} \leftarrow \operatorname*{argmin}_Q \mathbb{E}_{(s,a,s')\sim\mathcal{D}} \left[\left(Q(s,a) - (R(s,a) + \gamma \mathbb{E}_{\pi_k(a'|s')} \left[Q^\pi_k(s',a') - \alpha\gamma D(\pi_k(\cdot|s'), \pi_b(\cdot|s')) \right]) \right)^2 \right] \tag{5.15}$$

$$\pi_{k+1} \leftarrow \operatorname*{argmax}_\pi \mathbb{E}_{s\sim\mathcal{D}} \left[\mathbb{E}_{\pi(a|s)} \left[Q^\pi_{k+1}(s,a) \right] - \alpha D(\pi(\cdot|s'), \pi_b(\cdot|s')) \right] \tag{5.16}$$

One problem with the above method is that we have to fit a parametric model to $\pi_b(a|s)$ in order to evaluate the divergence term. Fortunately, in the case of KL, the divergence can be enforced implicitly, as in the **advantage weighted regression** or **AWR** method of [Pen+19], the **reward weighted regression** method of [PS07], the **advantage weighted actor critic** or **AWAC** method of [Nai+20], the **advantage weighted behavior model** or **ABM** method of [Sie+20], In this approach, we first solve (nonparametrically) for the new policy under the KL divergence constraint to get $\overline{\pi}_{k+1}$, and then we project this into the required

在它可能未曾见过的状态下，要求政策泛化到分布之外，这是当前离线强化学习方法的主要瓶颈 [Par+24b]。

一个非常简单且广泛使用的离线强化学习方法被称为行为克隆或 BC。这相当于训练一个策略来预测与每个观察到的状态相关的观察到的输出动作 a_t，因此我们旨在确保 $\pi(s_t) \approx a_t$，就像监督学习一样。这假设离线数据集是由专家创建的，因此属于模仿学习范畴（见第 5.4.1 节，详情见）。相比之下，离线强化学习方法可以利用次优数据。以下简要介绍了一些这些方法。更多详情，请参阅例如 [Lev+20b； Che+24b； Cet+24]。有关一些离线强化学习基准，请参阅 DR4L[Fu+20]、RL Unplugged[Gul+20]、OGBench（离线目标条件基准）[Par+24a]，和 D5RL[Raf+24]。

5.5.1 离线模型无关的强化学习

原则上，我们可以使用第 3.5 节中讨论的离线策略方法来处理离线强化学习。这些方法使用某种形式的重要性采样，基于 $\pi(a|s)/\pi_b(a|s)$，来重新加权行为策略收集的回放缓冲区 $\mathcal{D}$ 中的数据，使其更接近当前策略（被评估 / 学习的策略）。不幸的是，这种方法只有在行为策略接近新策略时才有效。在在线强化学习的情况下，可以通过逐渐更新新策略，使其远离行为策略，然后从更新的策略中采样新数据（这成为新的行为策略）来确保这一点。不幸的是，在离线情况下，这不是一个可行的选项。因此，我们需要使用其他策略来控制行为策略和学习的策略之间的差异，如下所述。（除了我们讨论的算法技术外，提高离线强化学习性能的另一种可靠方法是训练更大的、更多样化的数据集，如 [Kum+23] 所示。）

5.5.1.1 政策约束方法

在**策略约束**方法中，我们使用了一种改进的 actor-critic 形式，该形式在迭代 k 时使用以下形式的更新：

$$Q^{\pi}_{k+1} \leftarrow \operatorname*{argmin}_{Q} \mathbb{E}_{(s,a,s')\sim\mathcal{D}}\left[\left(Q(s,a) - (R(s,a) + \gamma \mathbb{E}_{\pi_k(a'|s')}\left[Q^{\pi}_{k}(s',a')\right])\right)^2\right] \tag{5.13}$$

$$\pi_{k+1} \leftarrow \operatorname*{argmax}_{\pi} \mathbb{E}_{s\sim\mathcal{D}}\left[\mathbb{E}_{\pi(a|s)}\left[Q^{\pi}_{k+1}(s,a)\right]\right] \quad \text{s.t.} \quad D(\pi,\pi_b) \leq \epsilon \tag{5.14}$$

在分布上，如 KL 散度或另一个 f- 散度，是 $D(\pi(\cdot|s)\,\pi_b(\cdot|s))$ 的散度度量。这确保了我们不会尝试在数据缓冲区（对于每个采样的状态 s）中看到太不同的动作 a' 上评估 Q 函数，否则可能会导致类似于对抗攻击的伪影。

作为一种替代添加约束的方法，我们可以在目标值和演员目标中添加一个惩罚 $\alpha D(\pi(\cdot|s)\pi_b(\cdot|s))$，从而得到以下更新：

$$Q^{\pi}_{k+1} \leftarrow \operatorname*{argmin}_{Q} \mathbb{E}_{(s,a,s')\sim\mathcal{D}}\left[\left(Q(s,a) - (R(s,a) + \gamma \mathbb{E}_{\pi_k(a'|s')}\left[Q^{\pi}_{k}(s',a') - \alpha\gamma D(\pi_k(\cdot|s'),\pi_b(\cdot|s'))\right])\right)^2\right] \tag{5.15}$$

$$\pi_{k+1} \leftarrow \operatorname*{argmax}_{\pi} \mathbb{E}_{s\sim\mathcal{D}}\left[\mathbb{E}_{\pi(a|s)}\left[Q^{\pi}_{k+1}(s,a)\right] - \alpha D(\pi(\cdot|s'),\pi_b(\cdot|s'))\right] \tag{5.16}$$

上述方法的一个问题是，我们必须将参数模型拟合到 $\pi_b(a|s)$ 以评估发散项。幸运的是，在 KL 的情况下，发散可以隐式地强制执行，就像在 **优势加权回归**或 **AWR** 方法中，[Pen+19] 的 **奖励加权回归**方法，[PS07]，的 **优势加权演员评论**或 **AWAC** 方法，[Nai+20] 的 **优势加权行为模型**或 **ABM** 方法，[Sie+20] 中，这种方法，我们首先在 KL 发散约束下（非参数地）求解新策略，以获得 π_{k+1}，然后将其投影到所需的

policy function class via supervised regression, as follows:

$$\overline{\pi}_{k+1}(a|s) \leftarrow \frac{1}{Z}\pi_b(a|s)\exp\left(\frac{1}{\alpha}Q_k^{\pi}(s,a)\right) \tag{5.17}$$

$$\pi_{k+1} \leftarrow \operatorname*{argmin}_{\pi} D_{\mathbb{KL}}\left(\overline{\pi}_{k+1} \parallel \pi\right) \tag{5.18}$$

In practice the first step can be implemented by weighting samples from $\pi_b(a|s)$ (i.e., from the data buffer) using importance weights given by $\exp\left(\frac{1}{\alpha}Q_k^{\pi}(s,a)\right)$, and the second step can be implemented via supervised learning (i.e., maximum likelihood estimation) using these weights.

It is also possible to replace the KL divergence with an integral probability metric (IPM), such as the maximum mean discrepancy (MMD) distance, which can be computed from samples, without needing to fit a distribution $\pi_b(a|s)$. This approach is used in [Kum+19]. This has the advantage that it can constrain the support of the learned policy to be a subset of the behavior policy, rather than just remaining close to it. To see why this can be advantageous, consider the case where the behavior policy is uniform. In this case, constraining the learned policy to remain close (in KL divergence) to this distribution could result in suboptimal behavior, since the optimal policy may just want to put all its mass on a single action (for each state).

5.5.1.2 Behavior-constrained policy gradient methods

Recently a class of methods has been developed that is simple and effective: we first learn a baseline policy $\pi(a|s)$ (using BC) and a Q function (using Bellman minimization) on the offline data, and then update the policy parameters to pick actions that have high expected value according to Q and which are also likely under the BC prior. An early example of this is the $Q^{\dagger}$ algorithm of [Fuj+19]. In [FG21], they present the **DDPG+BC** method, which optimizes

$$\max_{\pi} J(\pi) = \mathbb{E}_{(s,a)\sim\mathcal{D}}\left[Q(s,\mu^{\pi}(s)) + \alpha \log \pi(a|s)\right] \tag{5.19}$$

where $\mu^{\pi}(s) = \mathbb{E}_{\pi(a|s)}\left[a\right]$ is the mean of the predicted action, and α is a hyper-parameter. As another example, the **DQL** method of [WHZ23] optimizes a diffusion policy using

$$\min_{\pi} \mathcal{L}(\pi) = \mathcal{L}_{\text{diffusion}}(\pi) + \mathcal{L}_q(\pi) = \mathcal{L}_{\text{diffusion}}(\pi) - \alpha\mathbb{E}_{s\sim\mathcal{D},a\sim\pi(\cdot|s)}\left[Q(s,a)\right] \tag{5.20}$$

Finally, [Aga+22b] discusses how to transfer the policy from a previous agent to a new agent by combining BC with Q learning.

5.5.1.3 Uncertainty penalties

An alternative way to avoid picking out-of-distribution actions, where the Q function might be unreliable, is to add a penalty term to the Q function based on the estimated epistemic uncertainty, given the dataset $\mathcal{D}$, which we denote by $\text{Unc}(P_D(Q^{\pi}))$, where $P_D(Q^{\pi})$ is the distribution over Q functions, and Unc is some metric on distributions. For example, we can use a deep ensemble to represent the distribution, and use the variance of $Q(s,a)$ across ensemble members as a measure of uncertainty. This gives rise to the following policy improvement update:

$$\pi_{k+1} \leftarrow \operatorname*{argmax}_{\pi} \mathbb{E}_{s\sim\mathcal{D}}\left[\mathbb{E}_{\pi(a|s)}\left[\mathbb{E}_{P_D(Q^{\pi}_{k+1})}\left[Q^{\pi}_{k+1}(s,a)\right]\right] - \alpha\text{Unc}(P_D(Q^{\pi}_{k+1}))\right] \tag{5.21}$$

For examples of this approach, see e.g., [An+21; Wu+21; GGN22].

5.5.1.4 Conservative Q-learning and pessimistic value functions

An alternative to explicitly estimating uncertainty is to add a **conservative penalty** directly to the Q-learning error term. That is, we minimize the following wrt $\boldsymbol{w}$ using each batch of data $\mathcal{B}$:

$$\overline{\mathcal{E}}(\mathcal{B},\boldsymbol{w}) = \alpha\mathcal{C}(\mathcal{B},\boldsymbol{w}) + \mathcal{E}(\mathcal{B},\boldsymbol{w}) \tag{5.22}$$

政策函数类通过监督回归，如下所示：

$$\pi_{k+1}(a|s) \leftarrow \frac{1}{Z}\pi_b(a|s)\exp\left(\frac{1}{\alpha}Q_k^{\pi}(s,a)\right) \tag{5.17}$$

$$\pi_{k+1} \leftarrow \operatorname*{argmin}_{\pi} D_{\mathbb{KL}}\left(\pi_{k+1} \parallel \pi\right) \tag{5.18}$$

在实践中，第一步可以通过使用由 $\exp(\frac{1}{\alpha}Q_k^{\pi}(s,a))$给出的重要性权重对 $\pi_b(a|s)$ （即数据缓冲区）中的样本进行加权来实现，第二步可以通过使用这些权重通过监督学习（即最大似然估计）来实现。

也可能用积分概率度量（ IPM ），如最大均值差异（ MMD ）距离，来替换 KL 散度，该距离可以从样本中计算出来，无需拟合分布 $\pi_b(a|s)$。这种方法在 [Kum+19] 中被使用。这种方法的优点在于，它可以约束学习到的策略的支持集成为行为策略的子集，而不仅仅是保持接近。为了理解这为什么是有利的，考虑行为策略是均匀分布的情况。在这种情况下，将学习到的策略约束在 KL 散度上保持接近这个分布可能会导致次优行为，因为最优策略可能只想将所有质量放在单个动作（对于每个状态）上。

5.5.1.2 行为约束策略梯度 方法

最近开发了一类简单有效的方法：我们首先在离线数据上学习一个基线策略 $\pi(a|s)$ （使用 BC ）和一个 Q 函数（使用 Bellman 最小化），然后根据 Q 更新策略参数，选择具有高期望值的动作，这些动作在 BC 先验下也很有可能。这类方法的早期例子是 $Q^{\dagger}$ 算法，由 [Fuj+19] 提出。在 [FG21], 中，他们提出了 **DDPG+BC** 方法，该方法优化了

$$\max_{\pi} J(\pi) = \mathbb{E}_{(s,a)\sim\mathcal{D}}\left[Q(s,\mu^{\pi}(s)) + \alpha\log\pi(a|s)\right] \tag{5.19}$$

其中 $\mu^{\pi}(s) = \mathbb{E}_{\pi(a|s)}\left[a\right]$ 是预测动作的均值，而 α 是一个超参数。作为另一个例子，**DQL** 方法通过 [WHZ23] 使用

$$\min_{\pi} \mathcal{L}(\pi) = \mathcal{L}_{\text{diffusion}}(\pi) + \mathcal{L}_q(\pi) = \mathcal{L}_{\text{diffusion}}(\pi) - \alpha\mathbb{E}_{s\sim\mathcal{D},a\sim\pi(\cdot|s)}\left[Q(s,a)\right] \tag{5.20}$$

最后，[Aga+22b] 讨论了如何通过结合 BC 和 Qlearning 将策略从先前的智能体转移到新的智能体。

5.5.1.3 不确定性惩罚

一种避免选择分布外动作的替代方法，其中 Q 函数可能不可靠，是在基于数据集 $\mathcal{D}$ 估计的先验不确定性上对 Q 函数添加惩罚项，我们将其表示为 $\text{Unc}(P_D(Q^{\pi}))$，其中 $P_D(Q^{\pi})$ 是 Q 函数的分布，Unc 是分布上的某个度量。例如，我们可以使用深度集成来表示分布，并使用集成成员间 $Q(s,a)$ 的方差作为不确定性的度量。这导致了以下策略改进更新：

$$\pi_{k+1} \leftarrow \operatorname*{argmax}_{\pi} \mathbb{E}_{s\sim\mathcal{D}}\left[\mathbb{E}_{\pi(a|s)}\left[\mathbb{E}_{P_D(Q_{k+1}^{\pi})}\left[Q_{k+1}^{\pi}(s,a)\right]\right] - \alpha\text{Unc}(P_D(Q_{k+1}^{\pi}))\right] \tag{5.21}$$

例如，请参阅此方法的示例，例如 [An+21 ； Wu+21 ； GGN22]。

5.5.1.4 保守 Q 学习与悲观值函数

一种替代显式估计不确定性的方法是直接将一个**保守的惩罚**添加到 Q- 学习误差项中。也就是说，我们使用每一批数据 $\mathcal{B}$ 来最小化以下内容相对于 $\boldsymbol{w}$：

$$\mathcal{E}(\mathcal{B},\boldsymbol{w}) = \alpha\mathcal{C}(\mathcal{B},\boldsymbol{w}) + \mathcal{E}(\mathcal{B},\boldsymbol{w}) \tag{5.22}$$

where $\mathcal{E}(\mathcal{B}, \boldsymbol{w}) = \mathbb{E}_{(s,a,s') \in \mathcal{B}} \left[(Q_{\boldsymbol{w}}(s,a) - (r + \gamma \max_{a'} Q_{\boldsymbol{w}}(s', a')))^2 \right]$ is the usual loss for Q-learning, and $\mathcal{C}(\mathcal{B}, \boldsymbol{w})$ is some conservative penalty. In the **conservative Q learning** or **CQL** method of [Kum+20], we use the following penalty term:

$$\mathcal{C}(\mathcal{B}, \boldsymbol{w}) = \mathbb{E}_{s \sim \mathcal{B}, a \sim \pi(\cdot|s)} \left[Q_{\boldsymbol{w}}(s,a) \right] - \mathbb{E}_{(s,a) \sim \mathcal{B}} \left[Q_{\boldsymbol{w}}(s,a) \right] \tag{5.23}$$

If π is the behavior policy, this penalty becomes 0.

5.5.2 Offline model-based RL

In Chapter 4, we discussed model-based RL, which can train a dynamics model given a fixed dataset, and then use this to generate synthetic data to evaluate and then optimize different possible policies. However, if the model is wrong, the method may learn a suboptimal policy, as we discussed in Section 4.2.3. This problem is particularly severe in the offline RL case, since we cannot recover from any errors by collecting more data. Therefore various conservative MBRL algorithms have been developed, to avoid exploiting model errors. For example, [Kid+20] present the **MOREL** algorithm, and [Yu+20] present the **MOPO** algorithm. Unlike the value function uncertainty method of Section 5.5.1.3, or the conservative value function method of Section 5.5.1.4, these model-based methods add a penalty for visiting states where the model is likely to be incorrect.

In more detail, let $u(s,a)$ be an estimate of the uncertainty of the model's predictions given input (s,a). In MOPO, they define a conservative reward using $\overline{R}(s,a) = R(s,a) - \lambda u(s,a)$, and in MOREL, they modify the MDP so that the agent enters an absorbing state with a low reward when $u(s,a)$ is sufficiently large. In both cases, it is possible to prove that the model-based estimate of the policy's performance under the modified reward or dynamics is a lower bound of the performance of the policy's true performance in the real MDP, provided that the uncertainty function u is an error oracle, which means that is satisfies $D(M_{\boldsymbol{\theta}}(s'|s,a), M^*(s'|s,a)) \leq u(s,a)$, where M^* is the true dynamics, and $M_{\boldsymbol{\theta}}$ is the estimated dynamics.

For more information on offline MBRL methods, see [Che+24c].

5.5.3 Offline RL using reward-conditioned sequence modeling

Recently an approach to offline RL based on sequence modeling has become very popular. The basic idea — known as **upside down RL** [Sch19] or **RvS** (RL via Supervised learning) [KPL19; Emm+21] — is to train a generative model over future states and/or actions conditioned on the observed reward, rather than predicting the reward given a state-action trajectory. At test time, the conditioning is changed to represent the desired reward, and futures are sampled from the model. The implementation of this idea then depends on what kind of generative model to use, as we discuss below.

The **trajectory transformer** method of [JLL21] learns a joint model of the form $p(\boldsymbol{s}_{1:T}, \boldsymbol{a}_{1:T}, \boldsymbol{r}_{1:T})$ using a transformer, and then samples from this using beam search, selecting the ones with high reward (similar to MPC, Section 4.1.1). The **decision transformer** [Che+21b] is related, but just generates action sequences, and conditions on the past observations and the future reward-to-go. That is, it fits

$$\operatorname*{argmax}_{\boldsymbol{\theta}} \mathbb{E}_{p_{\mathcal{D}}} \left[\log \pi_{\boldsymbol{\theta}}(a_t | s_{0:t}, a_{0:t-1}, \mathrm{RTG}_{0:t}) \right] \tag{5.24}$$

where $\mathrm{RTG}_t = \sum_{k=t}^{T} r_t$ is the return to go. (For a comparison of decision transformers to other offline RL methods, see [Bha+24].)

The **diffuser** method of [Jan+22] is a diffusion version of trajectory transformer, so it fits $p(\boldsymbol{s}_{1:T}, \boldsymbol{a}_{1:T}, \boldsymbol{r}_{1:T})$ using diffusion, where the action space is assumed to be continuous. They also replace beam search with classifier guidance. The **decision diffuser** method of [Aja+23] extends diffuser by using classifer-free guidance, where the conditioning signal is the reward-to-go, simlar to decision transformer. However, unlike diffuser, the decision diffuser just models the future state trajectories (rather than learning a joint distribution over states and actions), and infers the actions using an **inverse dynamics model** $a_t = \pi(s_t, s_{t+1})$, which is trained using supervised learning.

其中 $\mathcal{E}(\mathcal{B},\boldsymbol{w}) = \mathbb{E}_{(s,a,s')\in\mathcal{B}}\left[(Q_{\boldsymbol{w}}(s,a) - (r + \gamma \max_{a'} Q_{\boldsymbol{w}}(s',a')))_2\right]$ 是 Q- 学习的通常损失，而 $\mathcal{C}(\mathcal{B},\boldsymbol{w})$ 是某种保守惩罚。在 **保守的 Q 学习** 或 **CQL** 方法中，我们使用以下惩罚项：

$$\mathcal{C}(\mathcal{B},\boldsymbol{w}) = \mathbb{E}_{s\sim\mathcal{B},a\sim\pi(\cdot|s)}\left[Q_{\boldsymbol{w}}(s,a)\right] - \mathbb{E}_{(s,a)\sim\mathcal{B}}\left[Q_{\boldsymbol{w}}(s,a)\right] \tag{5.23}$$

如果 π 是行为策略，则此惩罚变为 0。

5.5.2 基于离线模型的强化学习

在第 4 章，我们讨论了基于模型的强化学习，它可以在给定固定数据集的情况下训练动力学模型，然后使用该模型生成合成数据以评估和优化不同的可能策略。然而，如果模型错误，该方法可能会学习到一个次优策略，正如我们在第 4.2.3 节中讨论的那样。在离线强化学习的情况下，这个问题尤为严重，因为我们无法通过收集更多数据来恢复任何错误。因此，已经开发了各种保守的 MBRL 算法，以避免利用模型错误。例如，[Kid+20] 提出了 **MOREL** 算法，而 [Yu+20] 提出了 **MOPO** 算法。与第 5.5.1.3 节中的值函数不确定性方法或第 5.5.1.4 节中的保守值函数方法不同，这些基于模型的方法为访问模型可能错误的州添加了惩罚。

更详细地说，令 $u(s,a)$ 为给定输入 (s,a) 时模型预测不确定性的估计。在 MOPO 中，他们使用 $R(s,a) = R(s,a) - \lambda u(s,a)$ 定义了一个保守的奖励，而在 MOREL 中，他们修改了 MDP，使得当 $u(s,a)$ 足够大时，智能体进入一个低奖励的吸收状态。在这两种情况下，如果不确定性函数 u 是一个错误预言机，即它满足 $D(M_{\boldsymbol{\theta}}(s'|s,a)\, M^*(s'|s,a)) \leq u(s,a)$，其中 M^* 是真实动力学，$M_{\boldsymbol{\theta}}$ 是估计动力学，则可以证明基于模型的策略性能估计在修改后的奖励或动力学下是策略真实性能在真实 MDP 中的下界。

对于 离线 MBRL 方法的更多信息，请参阅 [Che+2 4c].

5.5.3 离线强化学习使用奖励条件序列建模

最近，一种基于序列建模的离线强化学习方法变得非常流行。其基本思想 —— 被称为**倒置 RL** [Sch19] 或 **RvS** （通过监督学习进行 RL ） [KPL19； Emm+21] —— 是在观察到的奖励的条件下训练一个生成模型来预测未来的状态和 / 或动作，而不是根据状态 - 动作轨迹预测奖励。在测试时，条件被改变以表示所需的奖励，并从模型中采样未来状态。然后，这种想法的实现取决于我们下面讨论的生成模型类型。

轨迹变换器**方法**学习了一个形式为 JLL[的联合模型 (21])，使用变换器，然后通过束搜索从该模型中采样，选择具有高奖励的样本（类似于 MPC，第 4.1.1p 节）。决策变换器 $\boldsymbol{s}_{1:T}$Cheb 与 $\boldsymbol{a}_{1:T}$ 相关，但仅生成动作序列，并基于过去的观察和未来的奖励到到达条件。也就是说，它适合

$$\underset{\boldsymbol{\theta}}{\operatorname{argmax}}\, \mathbb{E}_{p_{\mathcal{D}}}\left[\log \pi_{\boldsymbol{\theta}}(a_t|s_{0:t}, a_{0:t-1}, \mathrm{RTG}_{0:t})\right] \tag{5.24}$$

其中 $\mathrm{RTG}_t = \sum_{k=t}^{T} r_t$ 表示返回到游戏。（关于决策转换器与其他离线强化学习方法进行比较，请参阅 [Bha+24]。）

The **diffuser** method of [Jan+22] is a diffusion version of trajectory transformer, so it fits $p(\boldsymbol{s}_{1:T}, \boldsymbol{a}_{1:T}, \boldsymbol{r}_{1:T})$using diffusion, where the action space is assumed tobe continuous. They also replace beam searchwithclassifier guidance. The **decision diffuser**method of [Aja+23] extends diffuser by using classifer-free guidance, where the conditioning signal is the reward-to-go, similar to decision transformer. However, unlike diffuser, the decision diffuser just models the future state trajectories (rather than learning a joint distribution over states and actions), and infers the actions using an **inverse dynamics model** $a_t = \pi(s_t,s_{t+1})$, which istrained using supervisedlearning.

One problem with the above approaches is that conditioning on a desired return and taking the predicted action can fail dramatically in stochastic environments, since trajectories that result in a return may have only achieved that return due to chance [PMB22; Yan+23; Bra+22; Vil+22]. (This is related to the optimism bias in the control-as-inference approach discussed in Section 1.5.)

5.5.4 Hybrid offline/online methods

Despite the progress in offline RL, it is fundamentally more limited in what it can learn compared to online RL [OCD21]. Therefore, various hybrids of offline and online RL have been proposed, such as [Bal+23] and [Nak+23].

For example, [Nak+23] suggest pre-training with offline RL (specifically CQL) followed by online finetuning. Naively this does not work that well, because CQL can be too conservative, requiring the online learning to waste some time at the beginning fixing the pessimism. So they propose a small modification to CQL, known as **calibrated Q learning**. This simply prevents CQL from being too conservative, by replacing the CQL regularizer with

$$\min_Q \max_\pi J(Q,\pi) + \alpha \mathbb{E}_{s\sim\mathcal{D},a\sim\pi(a|s)}\left[\max(Q(s,a), V^{\pi_\beta}(s)) - \alpha \mathbb{E}_{(s,a)\sim\mathcal{D}}\left[Q(s,a)\right]\right] \tag{5.25}$$

where the $Q(s,a)$ term inside the max ensures conservatism (so Q lower bounds the value of the learned policy), and the $V^{\pi_\beta}(s)$ term ensures "calibration" (so Q upper bounds the value of the behavior policy). Then online finetuning is performed in the usual way.

5.6 LLMs and RL

In this section, we discuss some connections between RL and "**foundation models**" (see e.g., [Cen21]). These are large pretrained generative models of text and/or images such as **large language models** (**LLMs**) and their multimodal extension, sometimes called **vision language models** (**VLMs**). Note that this is a very fast growing field, so we only briefly mention a few highlights. For more details, see e.g. https://github.com/WindyLab/LLM-RL-Papers.

5.6.1 RL for LLMs

We can think of LLMs as agents, where the state s_t is the entire sequence of previous words, $s_t = (w_1, \ldots, w_{t-1})$, the action is the next word w_t[2], the stochastic policy $\pi(a_t|s_t)$ is the LLM, and the transition model is the determistic function $p(s_{t+1}|s_t, a_t) = \delta(s_t = \text{concat}(s_t, a_t))$. We see that the size of the state grows linearly over time, which is a standard way to capture non-local dependencies in a Markov model.

We discuss how to train these models below. Once trained, they are used in a semi-MDP fashion, in which at round t the agent generates an answer $a_t = (a_{t,1}, \ldots, a_{t,N_t})$, which is a sequence of N_t tokens, in response to a prompt from the user, $p_t = (p_{t,1}, \ldots, p_{t,M_t})$, and the previous context (dialog history), $c_t = (p_{1,1:M_1}, a_{1,1:N_1}, p_{2,1:M_2}, \ldots, a_{t-1,1:N_{t-1}})$. We can now define the state as the sequence of tokens $s_t = (c_t, p_t)$. Similarly, the action sequence a_t can be flattened into a single atomic (string-valued) action, since there is no intermediate feedback from the environment after each token is produced.[3] Note that, if there is a single round of prompting and answering (as is often assumed during training), then this is a contextual bandit problem rather than a full MDP. In particular, the context is the string p_t and the action is the string a_t. However, in multi-turn dialog situations, the agent's actions will affect the environment (i.e., the user's mental state, and hence subsequent prompt p_{t+1}), turning it into a full MDP.

[2] When using VLMs, the "words" are a tokenized representation of the visual input and/or output. Even when using language, the elementary components w_t are sub-words (which allows for generalization), not words. So a more precise term would be "tokens" instead of "words".

[3] The fact that the action (token) sequence is generated by an autoregressive policy inside the agent's head is an implementation detail, and not part of the problem specification; for example, the agent could instead use discrete diffusion to generate $a_t = (a_{t,1}, \ldots, a_{t,N_t})$.

上述方法的一个问题是，在随机环境中，基于期望回报并采取预测行动可能会失败得很惨，因为导致回报的轨迹可能只是由于偶然才实现了那个回报。[PMB22；严 +23；布拉 +22；维尔 +22]。（这与第 1.5 节中讨论的控制作为推理方法中的乐观偏差有关。）

5.5.4 混合离线 / 在线方法

尽管离线强化学习取得了进展，但与在线强化学习相比，它在可以学习的内容上本质上更为有限。因此，已经提出了各种离线和在线强化学习的混合方法，例如 Bal 和 Nak。

例如，[Nak+23] 建议先进行离线强化学习（特别是 CQL ）预训练，然后进行在线微调。这种方法并不奏效，因为 CQL 可能过于保守，需要在线学习在开始时浪费一些时间来纠正悲观情绪。因此，他们提出对 CQL 进行微小修改，称为**校准 Q 学习**。这仅仅通过用替换 CQL 正则化器来防止 CQL 过于保守。

$$\min_Q \max_\pi J(Q,\pi) + \alpha \mathbb{E}_{s\sim\mathcal{D},a\sim\pi(a|s)}\left[\max(Q(s,a), V^{\pi_\beta}(s)) - \alpha \mathbb{E}_{(s,a)\sim\mathcal{D}}\left[Q(s,a)\right]\right] \tag{5.25}$$

在 max 函数内部，$Q(s,a)$ 项确保了保守性（因此 Q 限制了学习策略的值），而 $V^{\pi_\beta}(s)$ 项确保了 “ 校准 ” （因此 Q 限制了行为策略的值）。然后以通常的方式进行在线微调。

5.6 机器学习和强化学习

在本节中，我们讨论了强化学习与 “ **基础模型** ” 之间的一些联系（例如，参见 [Cen21] ）。这些是文本和 / 或图像的大规模预训练生成模型，例如**大型语言模型**（ **LLMs** ）及其多模态扩展，有时被称为**视觉语言模型**（ **VLMs** ）。请注意，这是一个发展非常迅速的领域，所以我们只简要提及一些亮点。更多详情，请参见例如 https://github.com/WindyLab/LLM-RL-Papers。

5.6.1 用于 LLMs 的 RL

我们可以将 LLM 视为代理，其中状态 s_t 是之前所有单词的整个序列，$s_t = (w_1, \ldots, w_{t-1})$，动作是下一个单词 w_t[2]，随机策略 $\pi(a_t|s_t)$ 是 LLM，而转换模型是确定性函数 $p(s_{t+1}|s_t, a_t) = \delta(s_t = \text{concat}(s_t, a_t))$。我们注意到状态的大小随时间线性增长，这是在马尔可夫模型中捕获非局部依赖的标准方法。

我们将在下面讨论如何训练这些模型。一旦训练完成，它们将以半 MDP 的方式使用，在 t 轮次中，代理生成一个答案 $a_t = (a_{t,1}, \ldots, a_{t,N_t})$，这是一个由 N_t 个标记组成的序列，作为对用户提示 $p_t = (p_{t,1}, \ldots, p_{t,M_t})$ 和先前上下文（对话历史） $c_t = (p_{1,1:M_1}, a_{1,1:N_1}, p_{2,1:M_2}, \ldots, a_{t-1,1:N_{t-1}})$ 的响应。现在我们可以将状态定义为标记序列 $s_t = (c_t, p_t)$。同样，动作序列 a_t 可以被展平成一个单一的原子（字符串值）动作，因为在每个标记生成后，环境没有中间反馈。注意，如果有单轮次的提示和回答（在训练期间通常假设如此），那么这是一个上下文赌博机问题，而不是完整的 MDP。特别是，上下文是字符串 p_t，动作是字符串 a_t。然而，在多轮对话情况下，代理的动作将影响环境（即用户的心理状态，以及随后的提示 p_{t+1}），使其成为一个完整的 MDP。

[2]在使用VLMs时，“单词”是视觉输入和/或输出的标记表示。即使在使用语言的情况下，基本组件w_t 是子词（这允许泛化），而不是单词。因此，更精确的术语应该是 “ 标记 ”，而不是 “ 单词 ”。3 动作（标记）序列是由代理内部的自回归策略生成的，这是一个实现细节，而不是问题规范的一部分；例如，代理可以使用离散扩散来生成 $a_t = (a_{t,1}, \ldots, a_{t,N_t})$。

5.6.1.1 RLHF

LLMs are usually trained with behavior cloning, i.e., MLE on a fixed dataset, such as a large text (and tokenized image) corpus scraped from the web. This is called "**pre-training**". We can then improve their performance using RL, as we describe below; this is called "**post-training**".

A common way to perform post-training is to use **reinforcement learning from human feedback** or **RLHF**. This technique, which was first introduced in the InstructGPT paper [Ouy+22], works as follows. First a large number of (context, answer0, answer1) tuples are generated, either by a human or an LLM. Then human raters are asked if they prefer answer 0 or answer 1. Let $y = 0$ denote the event that they prefer answer 0, and $y = 1$ the event that they prefer answer 1. We can then fit a model of the form

$$p(y = 0|a_0, a_1, c) = \frac{\exp(\phi(c, a_0))}{\exp(\phi(c, a_0)) + \exp(\phi(c, a_1))} \tag{5.26}$$

using binary cross entropy loss, where $\phi(c, a)$ is some function that maps text to a scalar (interpreted as logits). Typically $\phi(c, a)$ is a shallow MLP on top of the last layer of a pretrained LLM. Finally, we define the reward function as $R(s, a) = \phi(s, a)$, where s is the context (e.g., a prompt or previous dialog state), and a is the action (answer generated by LLM). We then use this reward to fine-tune the LLM using a policy gradient method such as PPO (Section 3.4.3), or a simpler method such as **RLOO** [Ahm+24], which is based on REINFORCE (Section 3.2).

Note that this form of training assumes the agent just interacts with a single action (answer) in response to a single prompt, so is learning the reward for a bandit problem, rather than the full MDP. Also, the learned reward function is a known parametric model (since it is fit to the human feedback data), whereas in RL, the reward is an unknown non-differentiable blackbox function. When viewed in this light, it becomes clear that one can also use non-RL algorithms to improve performance of LLMs, such as **DPO** [Raf+23] or the density estimation methods of [Dum+24]. For more details on RL for LLMs, see e.g., [Kau+23].

5.6.1.2 Assistance game

In general, any objective-maximizing agent may suffer from reward hacking (Section 5.2.1), even if the reward has been learned using lots of RLHF data. In [Rus19], Stuart Russell proposed a clever solution to this problem. Specifically, the human and machine are both treated as agents in a two-player cooperative game, called an **assistance game**, where the machine's goal is to maximize the user's utility (reward) function, which is inferred based on the human's behavior using inverse RL. That is, instead of trying to learn a point estimate of the reward function using RLHF, and then optimizing that, we treat the reward function as an unknown part of the environment. If we adopt a Bayesian perspective on this, we can maintain a posterior belief over the model parameters. This will incentivize the agent to perform information gathering actions. For example, if the machine is uncertain about whether something is a good idea or not, it will proceed cautiously (e.g., by asking the user for their preference), rather than blindly solving the wrong problem. For more details on this framework, see [Sha+20].

5.6.1.3 Run-time inference as MPC

Recently the LLM community has investigated ways to improve the "reasoning" performance of LLMs by using MCTS-like methods (see Section 4.1.3). The basic idea is to perform Monte Carlo rollouts of many possible action sequences (by generating different "chains of thought" in response to the context so far), and then applying a value function to the leaves of this search tree to decide on which trajectory to return as the final "decision". The value function is usually learned using policy gradient methods, such as REINFORCE (see e.g., [Zel+24]). It is believed that OpenAI's recently released **o1** (aka **Strawberry**) model[4]uses similar techniques, most likely pre-training on large numbers of human reasoning traces.

Note that the resulting policy is an instance of MPC (Section 4.1.1) or decision time planning. This means that, as in MPC, the agent must replan after every new state observation (which incorporates the

[4]See https://openai.com/index/learning-to-reason-with-llms/.

5.6.1.1 机器学习强化人类反馈

LLMs 通常使用行为克隆进行训练，即在固定数据集上进行最大似然估计，例如从网络爬取的大量文本（和分词图像）语料库。这被称为 “ **预训练** ”。然后我们可以使用 RL 来提高它们的性能，如我们下面所描述的；这被称为 “ **后训练** ”。

进行训练后的一种常见方法是使用**来自人类反馈的强化学习**或 **RLHF**。这种技术最初在 InstructGPT 论文 [Ouy+22] 中提出，其工作原理如下。首先生成大量（上下文，答案 0，答案 1 ）元组，由人类或 LLM 完成。然后请人类评分者判断他们更喜欢答案 0 还是答案 1。设 $y = 0$ 表示他们更喜欢答案 0 的事件，$y = 1$ 表示他们更喜欢答案 1 的事件。然后我们可以拟合一个形式为的模型。

$$p(y = 0|a_0, a_1, c) = \frac{\exp(\phi(c, a_0))}{\exp(\phi(c, a_0)) + \exp(\phi(c, a_1))} \tag{5.26}$$

使用二元交叉熵损失，其中 $\phi(c,a)$ 是某种将文本映射到标量（解释为 logits ）的函数。通常 $\phi(c,a)$ 是在预训练 LLM 的最后一层之上的浅层 MLP。最后，我们定义奖励函数为 $R(s,a) = \phi(s,a)$，其中 s 是上下文（例如，一个提示或先前的对话状态），而 a 是动作（由 LLM 生成的答案）。然后我们使用这个奖励来使用策略梯度方法如 PPO （第 3.4.3 ）或更简单的方法如 **RLOO** [Ahm+24] 微调 LLM，该方法基于 REINFORCE （第 3.2 ）。

请注意，这种训练形式假设智能体仅对单个提示做出单个动作（答案）的交互，因此它是在学习带枪问题的奖励，而不是完整的 MDP。此外，学习的奖励函数是一个已知的参数模型（因为它是根据人类反馈数据进行拟合的），而在强化学习中，奖励是一个未知的非可微黑盒函数。从这个角度来看，很明显，也可以使用非强化学习算法来提高 LLMs 的性能，例如 **DPO** [Raf+23] 或 [Dum+24] 的密度估计方法。有关 LLMs 的强化学习的更多详细信息，请参阅例如 [Kau+23]。

5.6.1.2 辅助游戏

一般来说，任何追求目标最大化的智能体都可能遭受奖励黑客攻击（第 5.2.1 节），即使奖励是通过大量 RLHF 数据学习得到的。在 Rus19], Stuart Russell 提出了一个巧妙的解决方案。具体来说，人和机器都被视为两人合作游戏中的一方，这种游戏被称为辅助游戏，其中机器的目标是最大化用户的效用（奖励）函数，该函数是基于人类行为通过逆强化学习推断出来的。也就是说，我们不是试图使用 RLHF 学习奖励函数的点估计，然后对其进行优化，而是将奖励函数视为环境中的未知部分。如果我们采用贝叶斯视角，我们可以保持对模型参数的后验信念。这将激励智能体执行信息收集行为。例如，如果机器不确定某件事是否是个好主意，它将谨慎行事（例如，询问用户的偏好），而不是盲目地解决错误的问题。有关此框架的更多详细信息，请参阅 Sha+20]。

5.6.1.3 运行时推理作为 MPC

最近，LLM 社区通过使用类似 MCTS 的方法（参见第 4.1.3 节）研究了提高 LLM “ 推理 ” 性能的方法。基本思想是执行许多可能动作序列的蒙特卡洛滚动（通过生成不同的 “ 思维链 ” 来响应迄今为止的上下文），然后应用价值函数到搜索树的叶子节点，以决定返回哪个轨迹作为最终的 “ 决策 ”。价值函数通常使用策略梯度方法学习，例如 REINFORCE （例如，参见 [Zel+24] ）。人们认为 OpenAI 最近发布的 **o1** （又名 **Strawberry** ）模型 [4]可能使用了类似的技术，很可能是通过大量人类推理轨迹进行预训练。

请注意，生成的策略是 MPC （第 4.1.1 节）的一个实例或决策时间规划。这意味着，与 MPC 一样，代理必须在每次新的状态观察后重新规划（这包括

[4]查看 https://openai.com/index/learning-to-reason-with-llms /.

response from the user), making the method much slower than "reactive" LLM policies, that does not use look-ahead search (but still conditions on the entire past context). However, once trained, it may be possible to distill this slower "system 2" policy into a faster reactive "system 1" policy.

5.6.2 LLMs for RL

There are many ways that (pretrained) LLMs can be used for RL, by leveraging their prior knowledge, their ability to generate code, their "reasoning" ability, and their ability to perform **in-context learning** (which can be viewed as a form of Bayesian inference [PAG24], which is a "gradient-free" way of optimally learning that is well suited to rapid learning from limited data). The survey in [Cao+24] groups the literature into four main categories: LLMs for pre-processing the inputs, LLMs for rewards, LLMs for world models, and LLMs for decision making or policies. In our brief presentation below, we follow this categorization. (See also [Spi+24] for a similar grouping.)

5.6.2.1 LLMs for pre-processing the input

If the input observations $\boldsymbol{o}_t$ sent to the agent are in natural language (or some other textual representation, such as JSON), it is natural to use an LLM to process them, in order to compute a more compact representation, $\boldsymbol{s}_t = \boldsymbol{\phi}(\boldsymbol{o}_t)$, where $\boldsymbol{\phi}$ can the hidden state of the last layer of an LLM. This encoder can either be frozen, or fine-tuned with the policy network. Note that we can also pass in the entire past observation history, $\boldsymbol{o}_{1:t}$, as well as static "side information", such as instruction manuals or human hints; these can all be concatenated to form the LLM prompt.

For example, the **AlphaProof** system[5] uses an LLM (called the "formalizer network") to translate an informal specification of a math problem into the formal Lean representation, which is then passed to an agent (called the "solver network") which is trained, using the AlphaZero method (see Section 4.1.3.1), to generate proofs inside the Lean theorem proving environment. In this environment, the reward is 0 or 1 (proof is correct or not), the state space is a structured set of previously proved facts and the current goal, and the action space is a set of proof tactics. The agent itself is a separate transformer policy network (distincy from the formalizer network) that is trained from scratch in an incremental way, based on the AlphaZero method.

If the observations are images, it it is traditional to use a CNN to proccess the input, so $\boldsymbol{s}_t \in \mathbb{R}^N$ would be an embedding vector. However, we could alternatively use a VLM to compute a structured representation, where $\boldsymbol{s}_t$ might be a set of tokens describing the scene at a high level. We then proceed as in the text case.

Note that the information that is extracted will heavily depend on the prompt that is used. Thus we should think of an LLM/VLM as an **active sensor** that we can control via prompts. Choosing how to control this sensor requires expanding the action space of the agent to include computational actions [Che+24d]. Note also that these kinds of "sensors" are very expensive to invoke, so an agent with some limits on its time and compute (which is all practical agents) will need to reason about the value of information and the cost of computation. This is called **metareasoning** [RW91]. Devising good ways to train agents to perform both computational actions (e.g., invoking an LLM or VLM) and environment actions (e.g., taking a step in the environment or calling a tool) is an open research problem.

5.6.2.2 LLMs for rewards

It is difficult to design a reward function to cause an agent to exhibit some desired behavior, as we discussed in Section 5.2. Fortunately LLMs can often help with this task. We discuss a few approaches below.

In [Kli+24], they present the **Motif** system, that uses an LLM in lieu of a human to provide preference judgements to an RLHF system. In more detail, a pre-trained policy is used to collect trajectories, from which pairs of states, $(\boldsymbol{o}, \boldsymbol{o}')$, are selected at random. The LLM is then asked which state is preferable, thus generating $(\boldsymbol{o}, \boldsymbol{o}', y)$ tuples, which can be used to train a binary classifier from which a reward model is extracted, as in Section 5.6.1.1. In [Kli+24], the observations $\boldsymbol{o}$ are text captions generated by the NetHack game, but the same method could be applied to images if we used a VLM instead of an LLM for learning the

[5] See https://deepmind.google/discover/blog/ai-solves-imo-problems-at-silver-medal-level/.

用户响应），使得该方法比不使用前瞻搜索的“反应性”LLM 策略慢得多（但仍然基于整个过去上下文）。然而，一旦训练完成，可能可以将这种较慢的“系统 2”策略提炼成更快的反应性“系统 1”策略。

5.6.2 用于强化学习的 LLMs

有许多方法可以使用（预训练）LLM 进行 RL，通过利用它们的先验知识、生成代码的能力、它们的“推理”能力以及它们在上下文中进行学习的能力**（这可以被视为一种贝叶斯推理**，即 PAG[，这是一种“无梯度”的优化学习方法，非常适合从有限数据中快速学习）。在 Cao24]，的综述中，将文献分为四大类：用于预处理输入的 LLM、用于奖励的 LLM、用于世界模型的 LLM 以及用于决策或策略的 LLM。在下面的简要介绍中，我们遵循这种分类。[（参见 Spi+24] 的类似分组。）[

5.6.2.1 用于预处理输入的 LLMs

如果输入给代理的观察结果 $\boldsymbol{o}_t$ 是自然语言（或某种其他文本表示，如 JSON），则自然地使用 LLM 来处理它们，以便计算一个更紧凑的表示 $\boldsymbol{s}_t = \boldsymbol{\phi}(\boldsymbol{o}_t)$，其中 $\boldsymbol{\phi}$ 可以是 LLM 最后一层的隐藏状态。此编码器可以是冻结的，也可以与策略网络微调。请注意，我们还可以传递整个过去观察历史 $\boldsymbol{o}_{1:t}$，以及静态“辅助信息”，例如说明书或人类提示；所有这些都可以连接起来形成 LLM 提示。

例如，**AlphaProof** 系统[5] 使用一个 LLM（称为“形式化网络”）将数学问题的非正式规范翻译成形式化的 Lean 表示，然后将其传递给一个代理（称为“求解网络”），该代理使用 AlphaZero 方法（见第 4.1.3.1 节）在 Lean 定理证明环境中生成证明。在这个环境中，奖励是 0 或 1（证明正确与否），状态空间是先前证明的事实和当前目标的有序集合，动作空间是一组证明策略。代理本身是一个独立的变换策略网络（与形式化网络不同），它根据 AlphaZero 方法从头开始以增量方式训练。

如果观察结果是图像，使用 CNN 处理输入是传统做法，因此 $\boldsymbol{s}_t \in \mathbb{R}^N$ 将是一个嵌入向量。然而，我们也可以使用 VLM 来计算一个结构化表示，其中 $\boldsymbol{s}_t$ 可能是一组描述场景的高级标记。然后我们按照文本情况继续处理。

请注意，提取的信息将严重依赖于所使用的提示。因此，我们应该将 LLM/VLM 视为一个**活跃的传感器**，我们可以通过提示来控制它。选择如何控制这个传感器需要扩展代理的动作空间，包括计算动作 [Che+24d]。还要注意的是，这类“传感器”的调用成本非常高，因此具有时间计算限制的代理（即所有实际代理）需要就信息的价值和计算成本进行推理。这被称为**元推理** [RW91]。设计出好的方法来训练代理执行计算动作（例如，调用 LLM 或 VLM）和环境动作（例如，在环境中迈一步或调用工具）是一个开放的研究问题。

5.6.2.2 用于奖励的 LLMs

设计一个奖励函数以使智能体表现出某些期望的行为是困难的，正如我们在第 5.2 节中讨论的那样。幸运的是，大型语言模型通常可以帮助完成这项任务。以下我们讨论几种方法。

在 [Kli+24] 中，他们提出了 **Motif** 系统，该系统使用一个 LLM 代替人来为 RLHF 系统提供偏好判断。更详细地说，使用预训练策略来收集轨迹，从中随机选择状态对，$(\boldsymbol{o}, \boldsymbol{o}')$。然后询问 LLM 哪个状态更可取，从而生成 $(\boldsymbol{o}, \boldsymbol{o}', y)$ 元组，这些元组可以用来训练一个二元分类器，从中提取奖励模型，如第 5.6.1.1 节所述。在 [Kli+24] 中，观察 $\boldsymbol{o}$ 是 NetHack 游戏生成的文本标题，但如果我们使用 VLM 而不是 LLM 来学习

[5] See https://deepmind.google/discover/blog/ai-solves-imo-problems-at-silver-medal-level/.（由于输入内容包含特定网址和缩写，翻译可能不适用，因此保留原样。）

reward. The learned reward model is then used as a shaping function (Section 5.2.3) when training an agent in the NetHack environment, which has very sparse reward.

In [Ma+24], they present the **Eureka** system, that learns the reward using bilevel optimization, with RL on the inner loop and LLM-powered evolutionary search on the outer loop. In particular, in the inner loop, given a candidate reward function R_i, we use PPO to train a policy, and then return a scalar quality score $S_i = S(R_i)$. In the outer loop, we ask an LLM to generate a new set of reward functions, R'_i, given a population of old reward functions and their scores, (R_i, S_i), which have been trained and evaluated in parallel on a fleet of GPUs. The prompt also includes the source code of the environment simulator. Each generated reward function R_i is represented as a Python function, that has access to the ground truth state of the underlying robot simulator. The resulting system is able to learn a complex reward function that is sufficient to train a policy (using PPO) that can control a simulated robot hand to perform various dexterous manipulation tasks, including spinning a pen with its finger tips. In [Li+24], they present a somewhat related approach and apply it to Minecraft.

In [Ven+24], they propose **code as reward**, in which they prompt a VLM with an initial and goal image, and ask it to describe the corresponding sequence of tasks needed to reach the goal. They then ask the LLM to synthesize code that checks for completion of each subtask (based on processing of object properties, such as relative location, derived from the image). These reward functions are then "verified" by applying them to an offline set of expert and random trajectories; a good reward function should allocate high reward to the expert trajectories and low reward to the random ones. Finally, the reward functions are used as auxiliary rewards inside an RL agent.

There are of course many other ways an LLM could be used to help learn reward functions, and this remains an active area of research.

5.6.2.3 LLMs for world models

There are many papers that use transformers or diffusion models to represent the world model $p(s'|s, a)$, and learn them from data collected by the agent, as we discussed in Section 4.3. Here we focus our attention on ways to use pre-trained foundation models as world models (WM).

[Yan+24] presents **UniSim**, which is an action-conditioned video diffusion model trained on large amounts of robotics and visual navigation data. Combined with a VLM reward model, this can be used for decision-time planning as follows: sample candidate action trajectories from a proposal, generate the corresponding images, feed them to the reward model, score the rollouts, and then pick the best action from this set. (Note that this is just standard MPC in image space with a diffusion WM and a random shooting planning algorithm.)

[TKE24] presents **WorldCoder**, which takes a very different approach. It prompts a frozen LLM to generate code to represent the WM $p(s'|s, a)$, which it then uses inside of a planning algorithm. The agent then executes this in the environment, and passes back failed predictions to the LLM, asking it to improve the WM. (This is related to the Eureka reward-learning system mentioned in Section 5.6.2.2.)

There are of course many other ways an LLM could be used to help learn world models, and this remains an active area of research.

5.6.2.4 LLMs for policies

Finally we turn to LLMs as policies.

One approach is to pre-train a special purpose foundation model on state-action sequences (using behavior cloning), then sample the next action from it using $a_t \sim p(a_t|o_t, h_{t-1})$, where o_t is the latest observation and $h_{t-1} = (o_{1:t-1}, a_{1:t-1})$ is the history. See e.g., **Gato** model [Ree+22] **RT-2** [Zit+23], and **RoboCat** [Bou+23].

More recently it has become popular to leverage pre-trained LLMs that are trained on web data, and then to repurpose them as "agents" using in-context learning. We can then sample an action from the policy $\pi(a_t|p_t, o_t, h_{t-1})$, where p_t is a manually chosen prompt. This approach is used by the **ReAct** paper [Yao+22] which works by prompting the LLM to "think step-by-step" ("reasoning") and then to predict an action ("acting"). This approach can be extended by prompting the LLM to first retrieve relevant past examples

奖励。当在 NetHack 环境中训练一个代理时，所学习的奖励模型随后被用作塑造函数（第 5.2.3 节），该环境具有非常稀疏的奖励。

在 [Ma+24] 中，他们提出了 **Eureka** 系统，该系统通过双层优化学习奖励，内循环使用强化学习（ RL ），外循环使用由 LLM 驱动的进化搜索。特别是，在内循环中，给定一个候选奖励函数 R_i，我们使用 PPO 训练一个策略，然后返回一个标量质量分数 $S_i = S(R_i)$。在外循环中，我们要求一个 LLM 根据一组旧的奖励函数及其分数 R'_i生成一组新的奖励函数，(R_i,S_i)，这些奖励函数已经在多个 GPU 上并行训练和评估。提示还包括环境模拟器的源代码。每个生成的奖励函数 R_i 都表示为一个 Python 函数，该函数可以访问底层机器人模拟器的真实状态。该系统能够学习一个复杂的奖励函数，足以训练一个策略（使用 PPO ），该策略可以控制模拟机器人手执行各种灵巧的操作任务，包括用指尖旋转笔。在 [Li+24] 中，他们提出了一种相关的方法并将其应用于 Minecraft。

在 [Ven+24] 中，他们提出将 **代码作为奖励**，其中他们提示一个 VLM 使用初始和目标图像，并要求它描述达到目标所需的相应任务序列。然后，他们要求 LLM 合成检查每个子任务完成情况的代码（基于从图像中处理得到的对象属性，如相对位置）。然后，通过将这些奖励函数应用于离线专家和随机轨迹集来 “ 验证 ” 这些奖励函数；一个好的奖励函数应该将高奖励分配给专家轨迹，将低奖励分配给随机轨迹。最后，这些奖励函数被用作 RL 代理内的辅助奖励。

当然，还有许多其他方法可以使用 LLM 来帮助学习奖励函数，这仍然是一个活跃的研究领域。

5.6.2.3 用于世界模型的 LLMs

有许多论文使用变压器或扩散模型来表示世界模型 $p(s'|s,a)$，并从智能体收集的数据中学习它们，正如我们在第 4.3 节中讨论的那样。在这里，我们关注如何使用预训练的基础模型作为世界模型（ WM ）。

[杨 +24] 提出了 **UniSim**，这是一个在大量的机器人和视觉导航数据上训练的动作条件视频扩散模型。结合 VLM 奖励模型，它可以用于以下决策时间规划：从提议中采样候选动作轨迹，生成相应的图像，将它们输入到奖励模型中，评分滚动，然后从这组中选择最佳动作。（注意，这只是在图像空间中具有扩散 WM 和随机射击规划算法的标准 MPC。）

[TKE24] 提出了 **WorldCoder**，它采用了非常不同的方法。它提示一个冻结的 LLM 生成代码来表示 WM $p(s'|s,a)$，然后将其用于规划算法中。代理随后在环境中执行此操作，并将失败的预测传递回 LLM，要求它改进 WM。（这与第 5.6.2.2 节中提到的 Eureka 奖励学习系统相关。）

当然，还有许多其他方式可以使用 LLM 来帮助学习世界模型，这仍然是一个活跃的研究领域。

5.6.2.4 政策用 LLM

最终，我们将转向将 LLMs 作为策略。

一种方法是针对状态 - 动作序列（使用行为克隆）预先训练一个专用基础模型，然后从中采样下一个动作 $a_t \sim p$ （ $a_t|o_t$，h_{t-1} ），其中 o_t 是最新观察结果，h_{t-1} = （ $o_{1:t-1}$，$a_{1:t-1}$ ）是历史。例如，参见 **Gato** 模型 [Ree+22] **RT-2** [Zit+23]，以及 **RoboCat**[Bou+23]。

最近，利用在网页数据上训练的预训练 LLM，并通过情境学习将它们重新用作 “ 代理 ” 的做法变得流行。然后，我们可以从策略中采样一个动作 $\pi(a_t|p_t,o_t,h_{t-1})$，其中 p_t 是一个手动选择的提示。这种方法被 **ReAct** 论文 [Yao+22] 所采用，该论文通过提示 LLM “ 逐步思考 ” （ “ 推理 ” ）然后预测一个动作（ “ 行动 ” ）来工作。这种方法可以通过提示 LLM 首先检索相关的过去例子来扩展

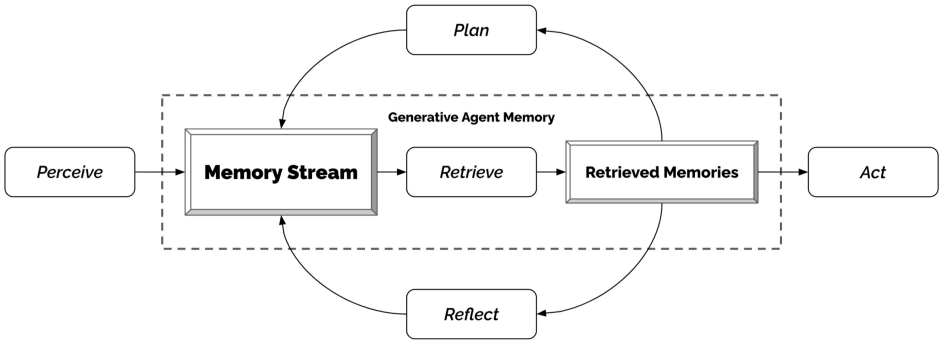


Figure 5.4: Illustration of how to use a pretrained LLM (combined with RAG) as a policy. From Figure 5 of [Par+23]. Used with kind permission of Joon Park.

from an external "memory", rather than explicitly storing the entire history h_t in the context (this is called **retrieval augmented generation** or **RAG**); see Figure 5.4 for an illustration. Note that no explicit learning (in the form of parametric updates) is performed in these systems; instead they rely entirely on in-context learning (and prompt engineering).

An alternative approach is to enumerate all possible discrete actions, and use the LLM to score them in terms of their likelihoods given the goal, and their suitability given a learned value function applied to the current state, i.e. $\pi(a_t = k|g, p_t, o_t, h_t) \propto \text{LLM}(w_k|g_t, p_t, h_t)V_k(o_t)$, where g_t is the current goal, w_k is a text description of action k, and V_k is the value function for action k. This is the approach used in the robotics **SayCan** approach [Ich+23], where the primitive actions a_k are separately trained goal-conditioned policies.

Calling the LLM at every step is very slow, so an alternative is to use the LLM to generate code that represents (parts of) the policy. For example, the **Voyager** system in [Wan+24a] builds up a reusable skill library (represented as Python functions), by alternating between environment exploration and prompting the (frozen) LLM to generate new tasks and skills, given the feedback collected so far.

There are of course many other ways an LLM could be used to help learn policies, and this remains an active area of research.

5.7 General RL, AIXI and universal AGI

The term "**general RL**" (see e.g., [Hut05; LHS13; HQC24; Maj21]) refers to the setup in which an agent receives a stream of observations $o_1, o_2, \ldots$ and rewards $r_1, r_2, \ldots$, and performs a sequence of actions in response, $a_1, a_2, \ldots$, but where we do not make any Markovian (or even stationarity) assumptions about the environment that generates the observation stream. Instead, we assume that the environment is a computable function or program p^*, which generated the observations $o_{1:t}$ and $r_{1:t}$ seen so far in response to the actions taken, $a_{1:t-1}$. We denote this by $U(p^*, \boldsymbol{a}_{1:t}) = (o_1 r_1 \cdots o_t r_t)$, where U is a universal Turing machine. If we use the receeding horizon control strategy (see Section 4.1.1), the optimal action at each step is the one that maximizes the posterior expected reward-to-go (out to some horizon m steps into the future). If we assume the agent represents the unknown environment as a program $p \in \mathcal{M}$, then the optimal action is given by the following **expectimax** formula:

$$a_t = \underset{a_t}{\text{argmax}} \sum_{o_t, r_t} \cdots \max_{a_m} \sum_{o_m, r_m} [r_t + \cdots + r_m] \sum_{p: U(p, \boldsymbol{a}_{1:m}) = (o_1 r_1 \cdots o_m r_m)} \Pr(p) \tag{5.27}$$

where $\Pr(p)$ is the prior probability of p, and we assume the likelihood is 1 if p can generate the observations given the actions, and is 0 otherwise.

The key question is: what is a reasonable prior over programs? In [Hut05], Marcus Hutter proposed to apply the idea of **Solomonoff induction** [Sol64] to the case of an online decision making agent. This

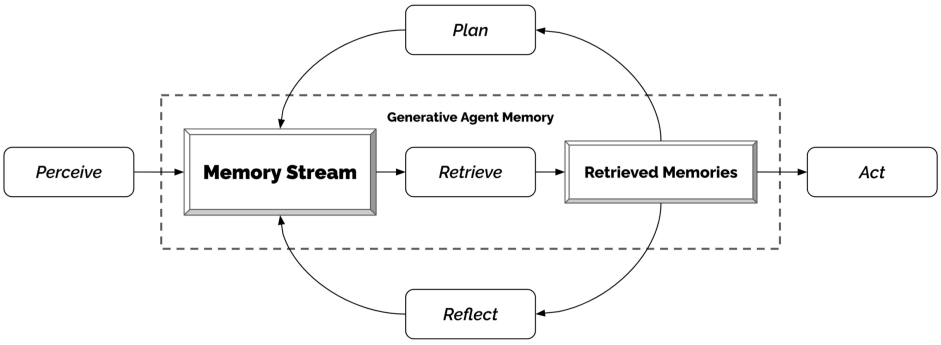


图 5.4：如何使用预训练的 LLM （结合 RAG ）作为策略的说明。来自图 5。*[Par+23]*。经 Joon Park 许可使用。

从外部 “ 记忆 ” 中获取，而不是在上下文中明确存储整个历史 h_t （这被称为**增强检索生成**或 **RAG** ）；参见图 5.4 以了解说明。请注意，这些系统中没有进行显式的学习（以参数更新的形式）；相反，它们完全依赖于上下文学习（和提示工程）。

一种替代方法是枚举所有可能的不连续动作，并使用 LLM 根据目标对它们的可能性进行评分，并根据应用于当前状态的学习值函数对它们的适用性进行评分，即 $\pi(a_t = k|g,p_t,o_t,h_t) \propto \text{LLM}(w_k|g_t,p_t,h_t)V_k(o_t)$，其中 g_t 是当前目标，w_k 是动作 k 的文本描述，而 V_k 是动作 k 的值函数。这是在机器人 **SayCan** 方法 [Ich+23] 中使用的，其中原始动作 a_k 是分别训练的目标条件策略。

每一步都调用 LLM 非常慢，因此一种替代方案是使用 LLM 生成表示（策略的）部分或全部的代码。例如，**Voyager** 系统在 [Wan+24a] 构建了一个可重用的技能库（表示为 Python 函数），通过在环境探索和提示（冻结的） LLM 生成新任务和技能之间交替，根据迄今为止收集到的反馈。

当然，还有许多其他方式可以使用 LLM 来帮助学习策略，这仍然是一个活跃的研究领域。

5.7 一般 RL，AIXI 和通用 AGI

“ **通用强化学习** ” 这一术语（例如参见 [Hut05； LHS13； HQC24； Maj21] ）指的是一种设置，其中智能体接收一系列观察 o_1 、 o_2 、 ... 和奖励 r_1 、 r_2 、 ...，并执行一系列动作作为回应 a_1 、 a_2 、 ...，但我们不对生成观察流的环境的马尔可夫性（甚至平稳性）假设。相反，我们假设环境是一个可计算的函数或程序 p^*，它根据采取的动作生成了迄今为止观察到的观察 $o_{1:t}$ 和 $r_{1:t}$ ， $a_{1:t-1}$。我们用 $U(p^* 、 \boldsymbol{a}_{1:t}) = (o_1 r_1 \cdots o_t r_t)$ 表示，其中 U 是一个通用图灵机。如果我们使用递减视野控制策略（参见第 4.1.1 节），则每一步的最优动作是最大化后验期望奖励到未来的动作（到未来 m 步）的那个动作。如果我们假设智能体将未知环境表示为一个程序 $p \in \mathcal{M}$，那么最优动作由以下 **expectimax** 公式给出：

$$a_t = \underset{a_t}{\text{argmax}} \sum_{o_t,r_t} \cdots \max_{a_m} \sum_{o_m,r_m} [r_t + \cdots + r_m] \sum_{p:U(p,\boldsymbol{a}_{1:m})=(o_1 r_1 \cdots o_m r_m)} \Pr(p) \tag{5.27}$$

$\Pr(p)$ 是 p 的先验概率，我们假设在给定动作的情况下，如果 p 可以生成观测值，则似然度为 1，否则为 0。

关键问题是：程序上的合理先验是什么？在 [Hut05], Marcus Hutter 提出将 **Solomonoff 归纳** [Sol64] 的思想应用于在线决策代理的情况。这

amounts to using the prior $\Pr(p) = 2^{-\ell(p)}$, where $\ell(p)$ is the length of program p. This prior favors shorter programs, and the likelihood filters out programs that cannot explain the data.

The resulting agent is known as **AIXI**, where "AI" stands for "Artificial Intelligence" and "XI" referring to the Greek letter ξ used in Solomonoff induction. The AIXI agent has been called the "most intelligent general-purpose agent possible" [HQC24], and can be viewed as the theoretical foundation of (universal) **artificial general intelligence** or **AGI**.

Unfortunately, the AIXI agent is intractable to compute, since it relies on Solomonoff induction and Kolmogorov complexity, both of which are intractable, but various approximations can be devised. For example, we can approximate the expectimax with MCTS (see Section 4.1.3). Alternatively, [GM+24] showed that it is possible to use meta learning to train a generic sequence predictor, such as a transformer or LSTM, on data generated by random Turing machines, so that the transformer learns to approximate a universal predictor. Another approach is to learn a policy (to avoid searching over action sequences) using TD-learning (Section 2.3.2); the weighting term in the policy mixture requires that the agent predict its own future actions, so this approach is known as **self-AIXI** [Cat+23].

Note that AIXI is a normative theory for optimal agents, but is not very practical, since it does not take computational limitations into account. In [Aru+24a; Aru+24b], they describe an approach which extends the above Bayesian framework, while also taking into account the data budget (due to limited environment interactions) that real agents must contend with (which prohibits modeling the entire environment or finding the optimal action). This approach, known as **Capacity-Limited Bayesian RL** (CBRL), combines Bayesian inference, RL, and rate distortion theory, and can be seen as a normative theoretical foundation for computationally bounded rational agents.

相当于使用先前的 $\Pr(p) = 2^{-\ell(p)}$，其中 $\ell(p)$ 是程序 p 的长度。这个先验偏好较短的程序，并且通过似然过滤掉无法解释数据的程序。

结果代理被称为 **AIXI**，其中 “AI” 代表 “ 人工智能 ”，“XI” 指的是在索洛蒙诺夫归纳中使用的希腊字母 ξ。AIXI 代理被称为 “ 可能的最智能通用代理 ”[HQC24], ，可以被视为（通用）**人工智能**或 **AGI** 的理论基础。

很遗憾，AIXI 智能体难以计算，因为它依赖于索洛蒙诺夫归纳法和 Kolmogorov 复杂度，这两者都是难以处理的，但可以设计各种近似方法。例如，我们可以用蒙特卡洛树搜索（ MCTS ）来近似 expectimax（见第 4.1.3 节）。或者，GM 展示了使用元学习在由随机图灵机生成的数据上训练通用序列预测器（如转换器或 LSTM ），这样转换器就能学会近似通用预测器。另一种方法是使用 TD-learning （见第 2.3.2 节）来学习策略（以避免搜索动作序列）；策略混合中的权重项要求智能体预测其自身的未来动作，因此这种方法被称为 self-AIXI Cat。

请注意，AIXI 是一个针对最优代理的规范性理论，但并不十分实用，因为它没有考虑到计算限制。在 [Aru+24a； Aru+24b], 他们描述了一种扩展上述贝叶斯框架的方法，同时考虑到实际代理必须应对的数据预算（由于有限的环境交互），这阻止了对整个环境的建模或找到最优行动。这种方法被称为 **容量限制贝叶斯强化学习** （ CBRL ），结合了贝叶斯推理、强化学习和率失真理论，可以被视为计算有限理性代理的规范性理论基础。