An Introduction to Reinforcement Learning

Alexander Cai

2025-03-28

Table of contents

Preface

Welcome to the study of reinforcement learning! This textbook accompanies the undergraduate course CS 1840/STAT 184 taught at Harvard. It is intended to be an approachable yet rigorous introduction to this active subfield of machine learning.

Prerequisites

This book assumes the same prerequisites as the course: You should be familiar with multivariable calculus, linear algebra, and probability. For Harvard undergraduates, this is fulfilled by Math 21a, Math 21b, and Stat 110, or their equivalents. Stat 111 is strongly recommended but not required. Specifically, we will assume that you know the following topics. The *italicized terms* have brief re-introductions in the text:

- **Linear Algebra:** Vectors and matrices, matrix multiplication, matrix inversion, eigenvalues and eigenvectors.
- **Multivariable Calculus:** Partial derivatives, the chain rule, Taylor series, *gradients, directional derivatives, Lagrange multipliers.*
- **Probability:** Random variables, probability distributions, expectation and variance, the law of iterated expectations (Adam's rule), covariance, conditional probability, Bayes's rule, and the law of total probability.

You should also be familiar with basic programming concepts such as variables, functions, loops, etc. Pseudocode listings will be provided for certain algorithms.

Course overview

The course will progress through the following units:

Chapter 1 presents motivation for the RL problem and compares RL to other fields of machine learning.

Chapter 2 introduces **Markov Decision Processes,** the core mathematical framework for describing a large class of interactive environments.

Chapter 3 is a standalone chapter on the **linear quadratic regulator** (LQR), an important tool for *continuous control*, in which the state and action spaces are no longer *finite* but rather *continuous*. This has widespread applications in robotics.

Chapter 4 introduces the **multi-armed bandit** (MAB) model for *stateless* sequential decision-making tasks. In exploring a number of algorithms, we will see how each of them strikes a different balance between *exploring* new options and *exploiting* known options. This **exploration-exploitation tradeoff** is a core consideration in RL algorithm design.

Chapter 5 is a standalone crash course on some tools from supervised learning that we will use in later chapters.

Chapter 6 introduces **fitted dynamic programming** (fitted DP) algorithms for solving MDPs. These algorithms use supervised learning to approximately evaluate policies when they cannot be evaluated exactly.

Chapter 7 explores an important class of algorithms based on iteratively improving a policy. We will also encounter the use of *deep neural networks* to express nonlinear policies and approximate nonlinear functions with many inputs and outputs.

Chapter 8 attempts to learn a good policy from expert demonstrations. At its most basic, this is an application of supervised learning to RL tasks.

Chapter 9 looks at ways to *explicitly* plan ahead when the environment's dynamics are known. We will study the *Monte Carlo Tree Search* heuristic, which has been used to great success in the famous AlphaGo algorithm and its successors.

Chapter 10 continues to investigate the exploration-exploitation tradeoff. We will extend ideas from multi-armed bandits to the MDP setting.

Appendix A contains an overview of selected background mathematical content and programming content.

Notation

It's worth it to spend a few words discussing the notation used for reinforcement learning. RL notation can *appear* quite complicated, since we often need to index across algorithm iterations, trajectories, and timesteps, so that certain values can have two or three indices attached to them. It's important to see beyond the formal notation and *interpret* the quantities being symbolized.

We will use the following notation throughout the book. This notation is inspired by Sutton & Barto (2018) and A. Agarwal et al. (2022). We use $[N]$ as shorthand for the set $\{0, 1, \dots, N-1\}$. We try to use each lowercase letter to represent an element of the set denoted by the corresponding (stylized) uppercase letter. We use $\triangle(\mathcal{X})$ to denote a distribution supported on some subset of $\mathcal{X}$. (The triangle is used to evoke the probability simplex.)

Table 1: Table of integer indices.

Index	Range	Definition (of index)
h	$[H]$	Time horizon index of an MDP (subscript).
k	$[K]$	Arm index of a multi-armed bandit (superscript).
t	$[T]$	Number of episodes.
i	$[I]$	Iteration index of an algorithm (subscript or superscript).

Table 2: Table of common notation.

Element	Space	Definition (of element)
s	$\mathcal{S}$	A state.
a	$\mathcal{A}$	An action.
r	$\mathbb{R}$	A reward.
γ	$[0, 1]$	A discount factor.
τ	$\mathcal{T}$	A trajectory $(s_0, a_0, r_0, \dots, s_{H-1}, a_{H-1}, r_{H-1})$
π	Π	A policy.
V^π	$\mathcal{S} \to \mathbb{R}$	The value function of policy π.
Q^π	$\mathcal{S} \times \mathcal{A} \to \mathbb{R}$	The action-value function (a.k.a. Q-function) of policy π.
A^π	$\mathcal{S} \times \mathcal{A} \to \mathbb{R}$	The advantage function of policy π.
v	$\mathcal{S} \to \mathbb{R}$	An approximation to the value function.

Element	Space	Definition (of element)
q	$\mathcal{S} \times \mathcal{A} \to \mathbb{R}$	An approximation to the Q function.
θ	Θ	A parameter vector.

Note that throughout the text, certain symbols will stand for either random variables or fixed values. We aim to clarify in ambiguous settings.

1 Introduction

Reinforcement learning (RL) is a branch of machine learning that studies **sequential decision-making** in **unknown environments.** An RL algorithm finds a strategy, called a **policy**, that maximizes the **reward** it obtains from the environment.

RL provides a powerful framework for attacking a wide variety of problems, including robotic control, video games and board games, resource management, language modeling, and more. It also provides an interdisciplinary paradigm for studying animal and human behaviour. Many of the most stunning results in machine learning, ranging from AlphaGo (Silver et al., 2016) to ChatGPT (OpenAI, 2022), are built using RL algorithms.

How does RL compare to the other two core machine learning paradigms, **supervised learning** and **unsupervised learning?**

- **Supervised learning** (SL) concerns itself with learning a mapping from inputs to outputs. Typically the data takes the form of *statistically independent* input-output pairs. In RL, however, the data is generated by the agent interacting with the environment, meaning the sequential observations of the state are *not independent* from each other. SL is a well-studied field that provides many useful tools for RL.
- **Unsupervised learning** concerns itself with learning the *structure* of data without the use of outside feedback or labels. In RL, though, the agent receives a **reward signal** from the environment, which can be thought of as a sort of feedback. Unsupervised learning is crucial in many real-world applications of RL for dimensionality reduction.

The key difference is that RL algorithms don't learn from some existing dataset; rather, they must go out and interact with the environment to collect their own data in an online way. This means RL algorithms face a distinct set of challenges from other kinds of machine learning. We'll discuss these more concretely in sec. 1.2.

Remark 1.1 (The reward hypothesis). Why do we only focus on maximizing a *scalar* reward signal? Surely a more descriptive or higher-dimensional signal would enable more efficient learning. Nonetheless, many (prominent) researchers hold that **scalar reward is enough** for developing behaviours that achieve a wide array of goals (Silver et al., 2021). This idea is also termed the **reward hypothesis**, and goes back at least as far as Turing, who suggested that one could train a "universal machine" using one input signal for "pain" and another for "pleasure" (Turing, 1948). Reinforcement learning takes this hypothesis seriously. It is

undeniable that maximizing scalar rewards has led to success in an assortment of sequential decision-making problems.

1.1 Core tasks of reinforcement learning

What tasks, exactly, are important for RL? Typically,

- **Policy evaluation (prediction)**: How 'good' is a specific state, or state-action pair (under a given policy)? That is, how much reward does it lead to in the long run? This is also called the task of **value estimation**.
- **Policy optimization (control)**: Suppose we fully understand how the environment behaves. What is the best action to take in every scenario?

1.2 Challenges of reinforcement learning

Recursion (bootstrapping): how can we "reuse" our current predictions to generate new information?

Exploration-exploitation tradeoff: should we try new actions, or capitalize on actions that we currently believe to be good?

Credit assignment: Consider this example: some mornings, you may wake up well rested, while other mornings, you may wake up drowsy and tired, even if the amount of time you spent asleep stays the same. What could be the cause? You take so many actions throughout the day, and any one of them could be the reason for your good or poor sleep. Was it a skipped meal? A lack of exercise? When these consequences interact with each other, it can be challenging to properly *assign credit* to the actions that cause the observed effects.

Reproducibility: the high variance inherent in interacting with the environment means that the results of RL experiments can be challenging to reproduce. Even when averaging across multiple random seeds, the same algorithm can achieve drastically different-seeming results (R. Agarwal et al., 2021).

1.3 Programming

Why include code in a textbook? We believe that implementing an algorithm is a strong test of your understanding of it; mathematical notation can often abstract away details, while a computer must be given every single instruction. We have sought to write readable Python code that is self-contained within each file. This approach is inspired by Sussman et al. (2013). There are some ways in which the code style differs from typical software projects:

- We keep use of language features to a minimum, even if it leads to code that could otherwise be more concisely or idiomatically expressed.
- The variable names used in the code match those used in the main text. For example, the variable `s` will be used instead of the more explicit `state`.

We also make extensive use of Python *type annotations* to explicitly specify variable types, including shapes of vectors and matrices using the jaxtyping library.

This is an interactive book built with Quarto (Allaire et al., 2024). It uses Python 3.11. It uses the JAX library for numerical computing. JAX was chosen for the clarity of its functional style and due to its mature RL ecosystem, sustained in large part by the Google DeepMind research group and a large body of open-source contributors. We use the standard Gymnasium library for interfacing with RL environments.

1.4 Bibliographic notes and further reading

Interest has surged in RL in the past decades, especially since AlphaGo's groundbreaking success (Silver et al., 2016). There are a number of recent textbooks that cover the field of RL:

Schultz et al. (1997) highlights RL as a normative theory for neuroscientific behaviour. Thorndike (1911) puts RL forward as a learning framework for animal behaviours.

Sutton & Barto (2018) set the framework for much of modern RL. Plaat (2022) is a graduate-level textbook on deep reinforcement learning. A. Agarwal et al. (2022) is a useful reference for theoretical guarantees of RL algorithms. S. E. Li (2023) highlights the connections between RL and optimal control. Mannor et al. (2024) is another advanced undergraduate course textbook. Bertsekas & Tsitsiklis (1996) introduced many of the core concepts of RL. Szepesvári (2010) is an invaluable resource on much of the theory underlying the methods in this book and elsewhere. Kochenderfer et al. (2022) provides a more probabilistic perspective alongside Julia code for various RL algorithms.

There are also a number of review articles that summarize recent advances. Murphy (2025) gives an overview of the past decade of advancements in RL. Ivanov & D'yakonov (2019) lists many popular algorithms.

Other textbooks focus on specific RL techniques or on applications of RL to specific fields. Albrecht et al. (2023) discusses **multi-agent** reinforcement learning. Rao & Jelvis (2022) focuses on applications of RL to finance. Y. Li (2018) surveys applications of RL to various fields.

2 Markov Decision Processes

2.1 Introduction

Machine learning studies algorithms that learn to solve a task "on their own", without needing a programmer to implement handwritten "if statements". **Reinforcement learning (RL)** is a branch of machine learning that focuses on **decision problems** like the following:

Example 2.1 (Decision problems)**.**

(a) Board games and video games, such as a game of chess, where a player character takes actions that update the state of the game (Guy, 2006).

(a) Inventory management, where a company must efficiently move resources from producers to consumers (Frans Berkelaar, 2009).

(a) Robotic control, where a robot can move and interact with the real world to complete some task (GPA Photo Archive, 2017).

All of these tasks involve taking a sequence of **actions** in order to achieve some goal. This **interaction loop** can be summarized in the following diagram:

In this chapter, we'll investigate the most popular mathematical formalization for such tasks: **Markov decision processes (MDPs)**. We will study **dynamic programming (DP)** algorithms for solving tasks when the rules of the environment are totally known. We'll describe how to *evaluate* different policies and how to compute (or approximate) the **optimal policy** for a given MDP. We'll introduce the **Bellman consistency condition**, which allows us to analyze the whole sequence of interactions in terms of individual timesteps.

Remark 2.1 (Further generalizations). In most real tasks, we *don't* explicitly know the rules of the environment, or can't concisely represent them on a computer. We will deal with this

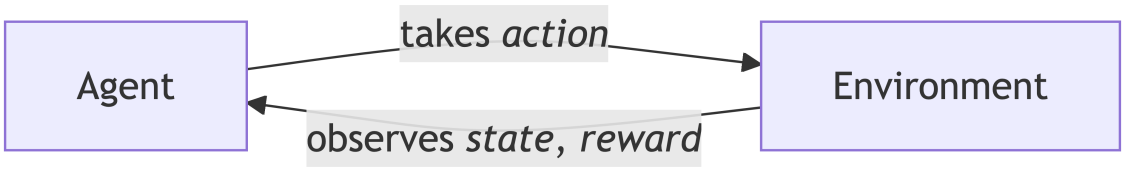


Figure 2.4: The RL interaction loop. The agent chooses an **action** that affects the environment. The environment updates according to the **state transitions**. Then the agent observes the updated environment and a **reward signal** that describes progress made towards the goal.

in future chapters; this chapter assumes the environment is known and accessible, so there's nothing to *learn* about the environment.

Additionally, in many tasks, only a *subset* of the state is visible to the observer. Such *partially observed* environments are out of scope of this textbook; see sec. 2.6 for additional resources.

2.2 Introduction to Markov decision processes

To gain a better grasp of decision processes and identify the key features that we want to formalize, let us frame the *robotic control* task in ex. 2.1 as a decision problem.

Example 2.2 (Robotic control as a decision problem)**.** Suppose the goal is to move a robot to a certain position.

- The **state** consists of the positions and velocities of the robot's joints.
- The **action** consists of the forces to apply to the robot's motors.
- The **state transitions** are essentially the rules of physics: after applying a certain force to the joints, their positions and velocities would change according to physical law.
- We **reward** positions that are closer to the desired position. To be more energy-efficient, we could also deduct reward for applying a lot of force, or add other terms that describe the ideal behaviour.

Exercise 2.1 (Identifying decision problems)**.** For each of the other examples in ex. 2.1, what information should the *state* include? What might the valid set of *actions* be? Describe the *state transitions* heuristically, and a *reward function* that describes how the task should be solved.

In many sequential decision-making problems, the state transitions only depend on the *current* state and action. For example, in ex. 2.2, if we use Newton's laws of physics to compute the state transitions, then just knowing the current positions and velocities is enough to calculate

the next positions and velocities, since Newton's laws are second-order. We say that such state transitions satisfy the **Markov property**.

Definition 2.1 (Markov property (informal))**.** An environment's state transitions satisfy the **Markov property** if "what happens next" only depends on the current state of the environment and not on the history of how we got here.

We will formally define the Markov property in def. 2.5 after introducing some notation for describing MDPs.

Exercise 2.2 (Checking for the Markov property)**.** Look back at the state transitions you described in Exercise 2.1. Do they satisfy the Markov property? (For chess, consider the threefold repetition rule: if the same position occurs three times during the game, either player may claim a draw.)

Sequential decision-making problems that satisfy the Markov property are called **Markov decision processes** (MDPs). MDPs are the core setting of modern RL research. We can further classify MDPs based on whether or not the task eventually ends.

Definition 2.2 (Episodic tasks)**.** Tasks that have a well-defined start and end, such as a game of chess or a football match, are called **episodic**. Each episode starts afresh and ends once the environment reaches a **terminal state**. The number of steps in an episode is called its **horizon**.

Definition 2.3 (Continuing tasks)**.** On the other hand, **continuing tasks**, such as managing a business's inventory, have no clear start or end point. The agent interacts with the environment in an indefinite loop.

Consider an episodic task where the horizon is fixed and known in advance. That is, the agent takes up to H actions, where H is some finite positive integer. We model such tasks with a **finite-horizon** MDP. Otherwise, if there's no limit to how long the episode can continue or if the task is continuing, we model the task with an **infinite-horizon** MDP. We'll begin with the finite-horizon case in sec. 2.3 and discuss the infinite-horizon case in sec. 2.4.

2.3 Finite-horizon Markov decision processes

Many real-world tasks, such as most sports games or video games, end after a fixed number of actions from the agent. In each episode:

1. The environment starts in some initial state s_0.
2. The agent observes the state and takes an action a_0.

3. The environment updates to state s_1 according to the action.

Steps 2 and 3 are repeated H times, resulting in a sequence of states and actions $s_0, a_0, \dots, s_{H-1}, a_{H-1}, s_H$, where s_H is some **terminal state**. Here's the notation we will use to describe an MDP:

Definition 2.4 (Finite-horizon Markov decision process)**.** The components of a finite-horizon Markov decision process are:

1. The **state** that the agent interacts with. We use s_h to denote the state at time h. $\mathcal{S}$ denotes the set of possible states, called the **state space**.
2. The **actions** that the agent can take. We use a_h to denote the action at time h. $\mathcal{A}$ denotes the set of possible actions, called the **action space**.
3. An **initial state distribution** $P_0 \in \triangle(\mathcal{S})$.
4. The **state transitions** (a.k.a. **dynamics**) $P : \mathcal{S} \times \mathcal{A} \to \triangle(\mathcal{S})$ that describe what state the agent transitions to after taking an action. We write $P(s_{h+1} \mid s_h, a_h)$ for the probability of transitioning to state s_{h+1} when starting in state s_h and taking action a_h. Note that we use the letter P for the state transition function and the symbol $\mathbb{P}$ more generally to indicate probabilities of events.
5. The **reward function**. In this course, we'll take it to be a deterministic function on state-action pairs, $r : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$, but many results will extend to a *stochastic* reward signal. We will use $r_h := r(s_h, a_h)$ to denote the reward obtained at time h.
6. A **time horizon** $H \in \mathbb{N}$ that specifies the maximum number of interactions in a **trajectory**, that is, a single sequence of states, actions, and rewards:

$$(s_0, a_0, r_0, \dots, s_{H-1}, a_{H-1}, r_{H-1}). \tag{2.1}$$

(Some sources omit the action and reward at the final time step.)

Combined together, these objects specify a finite-horizon Markov decision process:

$$\mathcal{M} = (\mathcal{S}, \mathcal{A}, P_0, P, r, H). \tag{2.2}$$

When there are **finitely** many states and actions, i.e. $|\mathcal{S}|, |\mathcal{A}| < \infty$, we can express the relevant quantities as vectors and matrices (i.e. *tables* of values):

$$P_0 \in [0,1]^{|\mathcal{S}|} \quad P \in [0,1]^{(|\mathcal{S}\times\mathcal{A}|)\times|\mathcal{S}|} \quad r \in \mathbb{R}^{|\mathcal{S}|\times|\mathcal{A}|}$$

(Verify that the types and shapes provided above make sense!)

Definition 2.5 (Markov property)**.** A decision process satisfies the **Markov property** if the next state is independent from the past states and actions when conditioned on the current state and action:

$$\mathbb{P}(s_{h+1} \mid s_0, a_0, \dots, s_h, a_h) = \mathbb{P}(s_{h+1} \mid s_h, a_h)$$

By their definition, Markov decision processes satisfy the Markov property. This is because the state transitions are *defined* using the function P, which only takes in a single state-action pair to transition from.

Example 2.3 (Tidying MDP)**.** Let's consider a simple decision problem throughout this chapter: the task of keeping your room tidy.

Your room has the possible states $\mathcal{S} = \{\text{orderly}, \text{messy}\}$. You can take either of the actions $\mathcal{A} = \{\text{ignore}, \text{tidy}\}$. The room starts off orderly, that is, $P_0(\text{orderly}) = 1$.

The **state transitions** are as follows: if you tidy the room, it becomes (or remains) orderly; if you ignore the room, it *might* become messy, according to the probabilities in tbl. 2.1.

The **rewards** are as follows: You get penalized for tidying an orderly room (a waste of time) or ignoring a messy room, but you get rewarded for ignoring an orderly room (since you can enjoy your additional time). Tidying a messy room is a chore that gives no reward.

Consider a time horizon of $H = 7$ days (one interaction per day). Let $t = 0$ correspond to Monday and $t = 6$ correspond to Sunday.

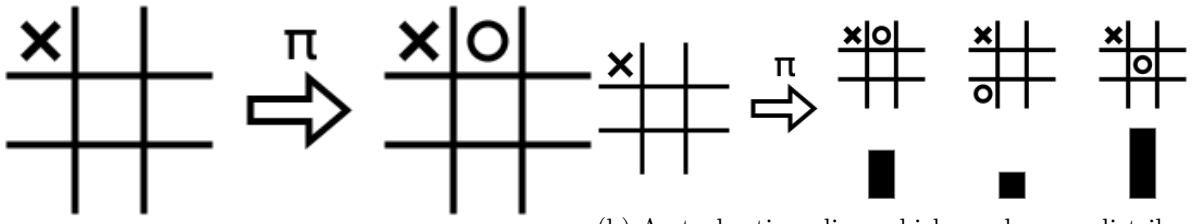
Table 2.1: Description of the MDP in ex. 2.3.

s	a	$P(\text{orderly} \mid s, a)$	$P(\text{messy} \mid s, a)$	$r(s, a)$
orderly	ignore	0.7	0.3	1
orderly	tidy	1	0	-1
messy	ignore	0	1	-1
messy	tidy	1	0	0

We'll now introduce several core concepts when working with MDPs.

Definition 2.6 (Policies)**.** A **policy** π describes the agent's strategy: which actions it takes in a given situation. A key goal of RL is to find the **optimal policy** that maximizes the total reward on average.

There are three axes along which policies can vary: their outputs, inputs, and time-dependence.



(a) A deterministic policy, which produces a single action

(b) A stochastic policy, which produces a distribution over possible actions

1. Outputs: **Deterministic or stochastic.** A deterministic policy outputs actions while a stochastic policy outputs *distributions* over actions.

2. Inputs: **history-independent or history-dependent.** A history-independent (a.k.a. "Markovian") policy only depends on the current state, while a history-dependent policy depends on the sequence of past states, actions, and rewards. We'll only consider history-independent policies in this course.

3. **Stationary or time-dependent.** A stationary (a.k.a. time-homogeneous) policy keeps the same strategy at all time steps, while a time-dependent policy can depend on the current timestep. For consistency with states and actions, we will denote the timestep as a subscript, i.e. $\pi = \{\pi_0, \dots, \pi_{H-1}\}$.

Note that for finite state and action spaces, we can represent a randomized mapping $\mathcal{S} \to \triangle(\mathcal{A})$ as a matrix $\pi \in [0,1]^{\mathcal{S}\times\mathcal{A}}$ where each row describes the policy's distribution over actions for the corresponding state.

Example 2.4 (Policies for the tidying MDP)**.** Here are some possible deterministic policies for the tidying MDP ex. 2.3:

- Always tidy: $\pi(s) = \text{tidy}$.
- Only tidy on weekends: $\pi_h(s) = \text{tidy}$ if $h \in \{5, 6\}$ and $\pi_h(s) = \text{ignore}$ otherwise.
- Only tidy if the room is messy: $\pi_h(\text{messy}) = \text{tidy}$ and $\pi_h(\text{orderly}) = \text{ignore}$ for all h.

Exercise 2.3 (Conditional independence for future states)**.** Suppose actions are chosen according to a history-independent policy. Use the chain rule of probability to show that, conditioned on the current state and action s_h, a_h, *all* future states $s_{h'}$ where $h' > h$ are conditionally independent of the past states and actions. That is, for all $h' > h$,

$$\mathbb{P}(s_{h'} \mid s_0, a_0, \dots, s_h, a_h) = \mathbb{P}(s_{h'} \mid s_h, a_h).$$

Definition 2.7 (Trajectory distribution)**.** Once we've chosen a policy, we can sample trajectories by repeatedly choosing actions according to the policy and observing the rewards and updated states returned by the environment. This generative process induces a distribution ρ^π over trajectories. (We assume that P_0 and P are clear from context.)

Example 2.5 (Trajectories in the tidying environment)**.** Here is a possible trajectory for the tidying example:

h	0	1	2	3	4	5	6
s	orderly	orderly	orderly	messy	messy	orderly	orderly
a	tidy	ignore	ignore	ignore	tidy	ignore	ignore
r	-1	1	1	-1	0	1	1

Could any of the policies in ex. 2.4 have generated this trajectory?

Note that for a history-independent policy, using the Markov property (def. 2.5), we can write down the likelihood function of this probability distribution in an **autoregressive** way (i.e. one timestep at a time):

Theorem 2.1 (Trajectory distributions of history-independent policies)**.** *Let π be a history-independent policy. Then its trajectory distribution ρ^π (def. 2.7) over sequences of actions and states $(s_0, a_0, \dots, s_{H-1}, a_{H-1})$ can be written as*

$$\rho^\pi(\tau) := P_0(s_0)\pi_0(a_0 \mid s_0)P(s_1 \mid s_0, a_0)\cdots P(s_{H-1} \mid s_{H-2}, a_{H-2})\pi_{H-1}(a_{H-1} \mid s_{H-1}). \tag{2.3}$$

Note that we use a deterministic reward function, so we only consider randomness over the states and actions.

Exercise 2.4 (Trajectory distribution for stochastic reward)**.** Modify eq. 2.3 for the case when the reward function is stochastic, that is, $r : \mathcal{S} \times \mathcal{A} \to \triangle(\mathbb{R})$.

For a deterministic policy π, we have that $\pi_h(a \mid s) = \mathbf{1}\{a = \pi_h(s)\}$; that is, the probability of taking an action is 1 if it's the unique action prescribed by the policy for that state and 0 otherwise. In this case, the only randomness in sampling trajectories comes from the initial state distribution P_0 and the state transitions P.

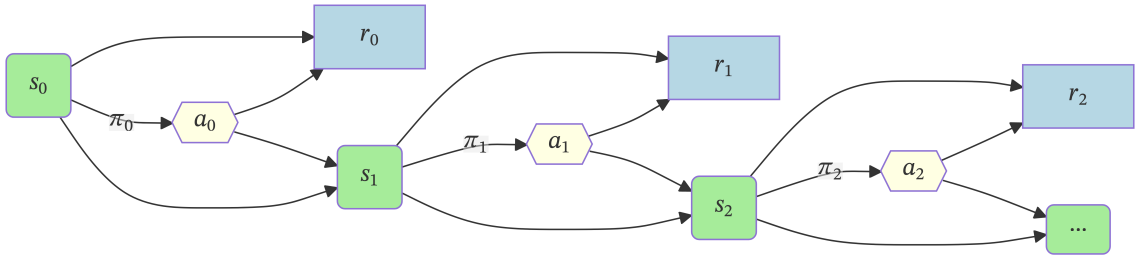


Figure 2.6: A trajectory is generated by taking a sequence of actions according to the history-independent policy.

2.3.1 Policy evaluation

Recall that the core goal of an RL algorithm is to find a policy that maximizes the expected total reward

$$\mathbb{E}[r_0 + \cdots + r_{H-1}], \tag{2.4}$$

where $r_h = r(s_h, a_h)$. Note that the quantity $r_0 + \cdots + r_{H-1}$ is a random variable whose distribution depends on the policy's trajectory distribution ρ^π (def. 2.7). We need tools for describing the expected total reward achieved by a given policy, starting in specific states and actions. This will allow us to compare the quality of different policies and compute the optimal policy.

Definition 2.8 (Value function)**.** Let π be a history-independent policy. Its **value function** at time h returns the expected remaining reward when starting in a specific state:

$$V_h^\pi(s) := \mathop{\mathbb{E}}_{\tau\sim\rho^\pi}[r_h + \cdots + r_{H-1} \mid s_h = s] \tag{2.5}$$

Definition 2.9 (Action-value function)**.** Similarly, a history-independent policy π's **action-value function** (aka Q function) at time h returns its expected remaining reward, starting in a specific state *and* taking a specific action:

$$Q_h^\pi(s, a) := \mathop{\mathbb{E}}_{\tau\sim\rho^\pi}[r_h + \cdots + r_{H-1} \mid s_h = s, a_h = a]. \tag{2.6}$$

These two functions define each other in the following way:

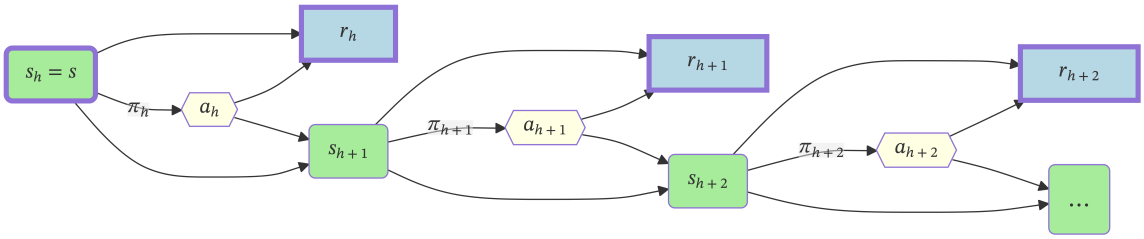


Figure 2.7: The policy starts in state s at time h. Then the expected remaining reward is computed.

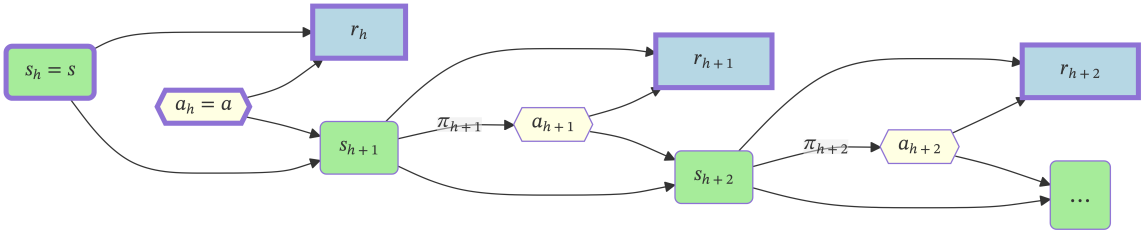


Figure 2.8: The policy starts in state s at time h and takes action a. Then the expected remaining reward is computed.

Theorem 2.2 (Relating V^π and Q^π)**.** *Let π be a history-independent policy. The value of a state is the expected action-value in that state over actions drawn from the policy:*

$$V_h^\pi(s) = \mathop{\mathbb{E}}_{a \sim \pi_h(\cdot|s)}[Q_h^\pi(s,a)]. \tag{2.7}$$

The value of a state-action pair is the sum of the immediate reward and the expected value of the following state:

$$Q_h^\pi(s,a) = r(s,a) + \mathop{\mathbb{E}}_{s' \sim P(s,a)}[V_{h+1}^\pi(s')]. \tag{2.8}$$

Exercise 2.5 (Proof of relationship between V^π and Q^π)**.** Use the law of iterated expectations to show eq. 2.7. Now apply linearity of expectation to show eq. 2.8. Where did you use the assumption that π doesn't depend on the past states and actions?

Remark 2.2 (Computing Q^π from V^π requires environment). Note that all you need to compute V_h^π from Q_h^π (eq. 2.7) is knowledge of the policy π. On the other hand, to compute Q_h^π from V_{h+1}^π (eq. 2.8), you need knowledge of the reward function and state transitions to look ahead one step. In later chapters, we'll study problems where the environment is *unknown*, making it more useful to approximate Q_h^π than V_h^π.

Putting eq. 2.7 and eq. 2.8 together reveals the *Bellman consistency equations*. By simply considering the cumulative reward as the sum of the *immediate* reward and the *remaining* reward, we can describe the value function in terms of itself. The resulting system of equations is named after **Richard Bellman** (1920–1984), who is credited with introducing dynamic programming in 1953.

Theorem 2.3 (Bellman consistency equations)**.** *For a history-independent policy π,*

$$V_h^\pi(s) = \mathop{\mathbb{E}}_{\substack{a\sim\pi_h(s)\\ s'\sim P(s,a)}}[r(s,a) + V_{h+1}^\pi(s')]. \tag{2.9}$$

The proof is left as Exercise 2.5. This is a system of $H|\mathcal{S}|$ equations (one equation per state per timestep) in $H|\mathcal{S}|$ unknowns (the values $V_h^\pi(s)$), so V^π is the unique solution, as long as no two states are exactly identical to each other.

Remark 2.3 (The Bellman consistency equation for deterministic policies). Note that for history-independent deterministic policies, the Bellman consistency equations simplify to

$$V_h^\pi(s) = r(s,\pi_h(s)) + \mathop{\mathbb{E}}_{s'\sim P(s,\pi_h(s))}[V_{h+1}^\pi(s')]. \tag{2.10}$$

How can we actually *evaluate* a given policy, that is, compute its value function?

Suppose we start with some guess $v_h : \mathcal{S} \to \mathbb{R}$ for the state values at time $h = 0, \ldots, H-1$. We write v in lowercase to indicate that it might not be an actual value function of a policy. How might we improve this guess?

Recall that the Bellman consistency equations (eq. 2.9) hold for the value functions of history-independent policies. So if v is close to the target V^π, it should nearly satisfy the system of equations associated with π. With this in mind, suppose we replace V_{h+1}^π with v_{h+1} on the right-hand side. Then the new right-hand side quantity,

$$\mathop{\mathbb{E}}_{\substack{a\sim\pi_h(s)\\ s'\sim P(s,a)}}[r(s,a) + v_{h+1}(s')], \tag{2.11}$$

can be thought of as follows: we take one action according to π, observe the immediate reward, and evaluate the next state using v. This is illustrated in fig. 2.9 below. This operation gives us an *updated* estimate of the value of $V_h^\pi(s)$ that is at least as accurate as applying $v(s)$ directly.

We can treat this operation of taking v_h to its improved version in eq. 2.11 as a *higher-order function* known as the Bellman operator.

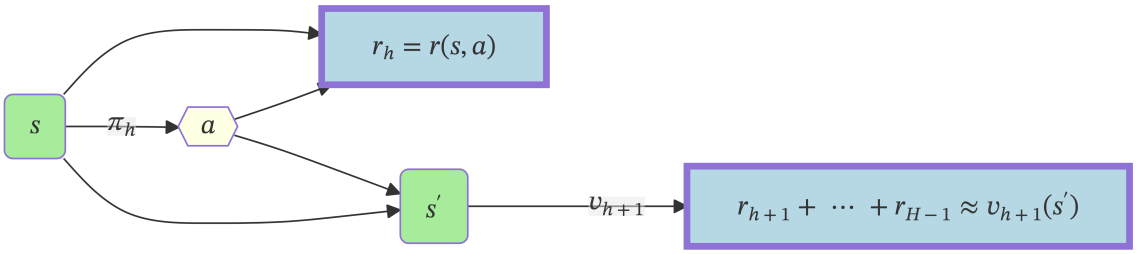


Figure 2.9: We evaluate the next state using v_{h+1}.

Definition 2.10 (Bellman operator)**.** Let $\pi : \mathcal{S} \to \triangle(\mathcal{A})$. the Bellman operator $\mathcal{J}^\pi : (\mathcal{S} \to \mathbb{R}) \to (\mathcal{S} \to \mathbb{R})$ is the higher-order function that takes in a function $v : \mathcal{S} \to \mathbb{R}$ and returns the r.h.s. of the Bellman equation with v substituted in:

$$\mathcal{J}^\pi(v) := \left(s \mapsto \mathop{\mathbb{E}}_{\substack{a\sim\pi(s) \\ s'\sim P(s,a)}} [r(s,a) + v(s')] \right). \tag{2.12}$$

The Bellman operator is a crucial tool for reasoning about MDPs. Intuitively, it answers the following question: if we evaluate the *next* state using v, how good is the *current* state, if we take a single action from the given policy?

function bellman_operator_looping(mdp : MDP, $\pi : \mathbb{R}^{S\times A}, v : \mathbb{R}^S$)
  " Looping definition of the Bellman operator. Concise version is below "
  $v^{\text{lhs}} \leftarrow \mathbf{0}^S$
  for $s \in \text{range}(S)$ **do**
    for $a \in \text{range}(A)$ **do**
      for $s' \in \text{range}(S)$ **do**
        $v^{\text{lhs}}(s) \leftarrow v^{\text{lhs}}(s) + \pi(s,a)P(s,a,s') \cdot (r(s,a) + \gamma v(s'))$
      end for
    end for
  end for
  return v^{lhs}
end function

Figure 2.10: An algorithm for computing the Bellman operator for a finite state and action space.

The Bellman operator also gives us a concise way to express the Bellman consistency equations (eq. 2.9) for the value function:

$$V_h^\pi = \mathcal{J}^{\pi_h}(V_{h+1}^\pi) \tag{2.13}$$

The Bellman consistency equations (eq. 2.9) give us a convenient algorithm for evaluating stationary policies: it expresses the value function at timestep h as a function of the value function at timestep $h+1$. This means we can start at the end of the time horizon, where the value is known, and work backwards in time, using the Bellman operator to compute the value function at each time step.

function dp_eval_finite(mdp : MDP, $\pi : \mathbb{R}^{S \times A}$)
  "Evaluate a policy using dynamic programming."
  $V^\pi \leftarrow \mathbf{0}^{\text{mdp}.H+1 \times \text{mdp}.S}$
  for $h \in \text{range}(\text{mdp}.H-1, -1, -1)$ **do**
    $V_h^\pi \leftarrow \mathcal{J}(\text{mdp}, \pi_h, V_{h+1}^\pi)$
  end for
  return $V_{:-1}^\pi$
end function

Figure 2.11: A dynamic programming algorithm for evaluating a policy in a finite-horizon MDP.

This runs in time $O(H \cdot |\mathcal{S}|^2 \cdot |\mathcal{A}|)$. Note that the implementation of the Bellman operator in fig. 2.10 can be easily modified to compute Q^π as an intermediate step. Do you see how?

Example 2.6 (Tidying policy evaluation)**.** Let's evaluate the policy from ex. 2.4 in the tidying MDP that tidies if and only if the room is messy. We'll use the Bellman consistency equation to compute the value function at each time step.

$$
\begin{aligned}
V^\pi_{H-1}(\text{orderly}) &= r(\text{orderly}, \text{ignore}) \\
&= 1 \\
V^\pi_{H-1}(\text{messy}) &= r(\text{messy}, \text{tidy}) \\
&= 0 \\
V^\pi_{H-2}(\text{orderly}) &= r(\text{orderly}, \text{ignore}) + \mathop{\mathbb{E}}_{s' \sim P(\text{orderly}, \text{ignore})}[V^\pi_{H-1}(s')] \\
&= 1 + 0.7 \cdot V^\pi_{H-1}(\text{orderly}) + 0.3 \cdot V^\pi_{H-1}(\text{messy}) \\
&= 1 + 0.7 \cdot 1 + 0.3 \cdot 0 \\
&= 1.7 \\
V^\pi_{H-2}(\text{messy}) &= r(\text{messy}, \text{tidy}) + \mathop{\mathbb{E}}_{s' \sim P(\text{messy}, \text{tidy})}[V^\pi_{H-1}(s')] \\
&= 0 + 1 \cdot V^\pi_{H-1}(\text{orderly}) + 0 \cdot V^\pi_{H-1}(\text{messy}) \\
&= 1 \\
V^\pi_{H-3}(\text{orderly}) &= r(\text{orderly}, \text{ignore}) + \mathop{\mathbb{E}}_{s' \sim P(\text{orderly}, \text{ignore})}[V^\pi_{H-2}(s')] \\
&= 1 + 0.7 \cdot V^\pi_{H-2}(\text{orderly}) + 0.3 \cdot V^\pi_{H-2}(\text{messy}) \\
&= 1 + 0.7 \cdot 1.7 + 0.3 \cdot 1 \\
&= 2.49 \\
V^\pi_{H-3}(\text{messy}) &= r(\text{messy}, \text{tidy}) + \mathop{\mathbb{E}}_{s' \sim P(\text{messy}, \text{tidy})}[V^\pi_{H-2}(s')] \\
&= 0 + 1 \cdot V^\pi_{H-2}(\text{orderly}) + 0 \cdot V^\pi_{H-2}(\text{messy}) \\
&= 1.7
\end{aligned}
$$

etc. You may wish to repeat this computation for the other policies to get a better sense of this algorithm.

s	$h=0$	$h=1$	$h=2$	$h=3$	$h=4$	$h=5$	$h=6$
orderly	5.56217	4.79277	4.0241	3.253	2.49	1.7	1
messy	4.79277	4.0241	3.253	2.49	1.7	1	0

2.3.2 Optimality

We've just seen how to *evaluate* a given policy. But how can we find the *best* policy for a given environment? We must first define what it means for a policy to be optimal. We'll investigate optimality for *history-independent* policies. In Theorem 2.6, we'll see that constraining ourselves to history-independent policies isn't actually a constraint: the best history-independent policy performs at least as well as the best history-dependent policy, in all scenarios!

Remark 2.4 (Intuition). How is this possible? Recall the Markov property (def. 2.5), which says that once we know the current state, the next state becomes independent from the past history. This means that history-dependent policies, intuitively speaking, don't benefit over simply history-independent ones in MDPs.

Definition 2.11 (Optimal policies)**.** Let $\pi^\star$ be a history-independent policy. We say that $\pi^\star$ is **optimal** if $V_h^{\pi^\star}(s) \geq V_h^\pi(s)$ for any history-independent policy π at any time $h \in [H]$. In other words, an optimal history-independent policy achieves the highest possible expected remaining reward in every state at every time.

Example 2.7 (Optimal policy in the tidying MDP)**.** For the tidying MDP (ex. 2.3), you might guess that the optimal strategy is

$$\pi(s) = \begin{cases} \text{tidy} & s = \text{messy} \\ \text{ignore} & \text{otherwise} \end{cases} \tag{2.14}$$

since this keeps the room in the "orderly" state in which reward can be obtained.

Definition 2.12 (Optimal value function)**.** Given a state s at time h, the optimal value $V_h^\star(s)$ is the *maximum* expected remaining reward achievable by any history-independent policy π:

$$V_h^\star(s) := \max_\pi V_h^\pi(s). \tag{2.15}$$

The optimal action-value function is defined analogously:

$$Q_h^\star(s,a) := \max_\pi Q_h^\pi(s,a). \tag{2.16}$$

These satisfy the following relationship (cf Theorem 2.2):

Theorem 2.4 (Relating $V^\star$ and $Q^\star$)**.** *The optimal value of a state is the maximum value across actions from that state:*

$$V_h^\star(s) = \max_{a \in \mathcal{A}} Q_h^\star(s,a). \tag{2.17}$$

The optimal value of an action is the immediate reward plus the expected value from the next state:

$$Q_h^\star(s,a) = r(s,a) + \mathop{\mathbb{E}}_{s' \sim P(s,a)}[V_{h+1}^\star(s')]. \tag{2.18}$$

Proof. We first prove eq. 2.17. We begin by expanding the definition of the optimal value function using eq. 2.7:

$$\begin{aligned} V_h^\star(s) &= \max_\pi V_h^\pi(s) \\ &= \max_\pi \mathop{\mathbb{E}}_{a\sim\pi_h(s)}[Q_h^\pi(s,a)]. \end{aligned} \tag{2.19}$$

Now note that Q_h^π doesn't depend on π_h, so we can split the maximization over $\pi = (\pi_0, \dots, \pi_{H-1})$ into an *outer* optimization over the immediate action π_h and an *inner* optimization over the remaining actions $\pi_{h+1}, \dots, \pi_{H-1}$:

$$\max_\pi \mathop{\mathbb{E}}_{a\sim\pi_h(s)}\Big[Q_h^\pi(s,a)] = \max_{\pi_h} \mathop{\mathbb{E}}_{a\sim\pi_h(s)}\Big[\max_{\pi_{h+1},\dots,\pi_{H-1}} Q_h^\pi(s,a)\Big]. \tag{2.20}$$

But now the inner quantity is exactly $Q_h^\star(s,a)$ as defined in eq. 2.16. Now note that maximizing over π_h reduces to just maximizing over the action taken:

$$\max_{\pi_h} \mathop{\mathbb{E}}_{a\sim\pi_h(s)}[\cdots] = \max_{a\in\mathcal{A}}[\cdots]. \tag{2.21}$$

This proves eq. 2.17.

We now prove eq. 2.18. We begin by using eq. 2.8:

$$\begin{aligned} Q_h^\star(s,a) &= \max_\pi Q_h^\pi(s,a) \\ &= \max_\pi \Big[r(s,a) + \mathop{\mathbb{E}}_{s'\sim P(s,a)}[V_{h+1}^\pi(s')]\Big] \end{aligned}$$

We can move the maximization over π under the expectation since s, a, and s' are all independent of π. Substituting the definition of $V^\star$ concludes the proof. □

As in the Bellman consistency equations (Theorem 2.3), combining eq. 2.17 and eq. 2.18 gives the *Bellman optimality equations.*

Theorem 2.5 (Bellman optimality equations)**.** *The optimal value function function (def. 2.12) satisfies, for all timesteps $h \in [H-1]$,*

$$V_h^\star(s) = \max_a r(s,a) + \mathop{\mathbb{E}}_{s'\sim P(s,a)}[V_{h+1}^\star(s')]. \tag{2.22}$$

Like the Bellman consistency equations (eq. 2.9), This is a system of $H|\mathcal{S}|$ equations in $H|\mathcal{S}|$ unknowns, so $V^\star$ is the unique solution (assuming all the states are distinguishable from each other).

*Note that the letter π doesn't appear. This will prove useful when we discuss **off-policy algorithms** later in the course.*

We now construct a deterministic history-independent optimal policy by acting *greedily* with respect to the optimal action-value function:

Definition 2.13 (Greedy policies)**.** For any sequence of functions $q_h : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$ for $h = 0, \dots, H-1$, we define the **greedy policy** π^q to be the deterministic policy that selects the action with the highest value according to q at each state:

$$\pi_h^q(s) := \arg\max_{a \in \mathcal{A}} q_h(s, a). \tag{2.23}$$

Note that it is *not* true in general that $Q^{\pi^q} = q$ for a greedy policy! For one, q might not even be a consistent Q function.

Theorem 2.6 (A greedy optimal policy)**.** *The greedy policy $\pi^{Q^\star}$, where $Q^\star$ is the optimal action-value function (eq. 2.16), is an optimal policy (def. 2.11). Furthermore, $\pi^{Q^\star}$ is optimal across* all *policies π^{any}, including history-dependent ones, in the sense that for any partial trajectory*

$$\tau_h = (s_0, a_0, r_0, \dots, s_{h-1}, a_{h-1}, r_{h-1}, s_h),$$

it achieves a higher expected remaining reward:

$$\mathop{\mathbb{E}}_{\tau \sim \rho^{\pi^{Q^\star}}}[r_h + \cdots + r_{H-1} \mid \tau_h] \geq \mathop{\mathbb{E}}_{\tau \sim \rho^{\pi^{any}}}[r_h + \cdots + r_{H-1} \mid \tau_h]. \tag{2.24}$$

By conditioning on τ_h, we mean to condition on the event that $s_{h'}, a_{h'}$ are the state and action visited at time $h' < h$, and s_h is the state at time h. (If the reward function were stochastic, we would condition on the observed rewards as well.)

We will now prove by induction that $\pi^{Q^\star}$ is optimal. Essentially, we begin at the *end* of the trajectory, where the optimal action is simply the one that yields highest immediate reward. Once the optimal values at time $H-1$ are known, we use these to compute the optimal values at time $H-2$ using the Bellman optimality equations (eq. 2.38), and proceed backward through the trajectory.

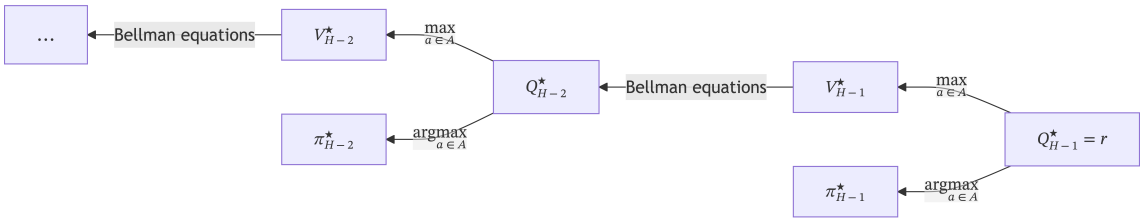


Figure 2.12: Illustrating a dynamic programming algorithm for computing the optimal policy in a finite-horizon MDP.

Proof. At the end of the trajectory (time step $H-1$), we can't take any more actions, so the Q-function simply returns the immediate reward:

$$Q^\star_{H-1}(s,a) = r(s,a). \tag{2.25}$$

$\pi^{Q^\star}$ is then optimal across *all* policies at time $H-1$, since the policy only determines a single action, the one which maximizes (expected) remaining reward:

$$\mathop{\mathbb{E}}_{\tau \sim \rho^{\pi^{Q^\star}}}[r_{H-1} \mid s_{H-1} = s] = \max_{a \in \mathcal{A}} r(s,a). \tag{2.26}$$

For the inductive step, suppose $\pi^{Q^\star}$ is optimal across *all* policies at time $h+1$. Note that this implies $V^{\pi^{Q^\star}} = V^\star$. Then for any other policy π^{any} (which could be history-dependent), and for any partial trajectory τ_h,

$$\begin{aligned}
&\mathop{\mathbb{E}}_{\tau \sim \rho^{\pi^{\text{any}}}}[r_h + \cdots + r_{H-1} \mid \tau_h] \\
&= \mathop{\mathbb{E}}_{\substack{a \sim \pi_h^{\text{any}}(\tau_h) \\ s' \sim P(s_h, a)}} \left[r_h + \mathop{\mathbb{E}}_{\tau \sim \rho^{\pi^{\text{any}}}}[r_{h+1} + \cdots + r_{H-1} \mid \tau_h, a_h = a, s_{h+1} = s'] \right] \\
&\leq \mathop{\mathbb{E}}_{\substack{a \sim \pi_h^{\text{any}}(\tau_h) \\ s' \sim P(s_h, a)}} [r_h + V^\star_{h+1}(s')] \\
&\leq \max_{a \in \mathcal{A}} r(s_h, a) + \mathop{\mathbb{E}}_{s' \sim P(s_h, a)} [V^\star_{h+1}(s')] \\
&= V^\star_h(s_h).
\end{aligned} \tag{2.27}$$

This completes the inductive step, which shows that $\pi^{Q^\star}$ is optimal across *all* policies at every time $h \in [H]$. □

function find_optimal_policy(mdp : MDP)
  $Q \leftarrow \mathbf{0}^{H \times S \times A}$
  $\pi \leftarrow \mathbf{0}^{H \times S \times A}$
  $V \leftarrow \mathbf{0}^{H+1 \times S}$
  for $h \in \text{range}(H-1, -1, -1)$ **do**
    $Q_h \leftarrow r + PV_{h+1}$
    $\pi_h \leftarrow \text{jnp.eye}(S)_{\text{jnp.argmax}(Q_h)}$
    $V_h \leftarrow \text{jnp.max}(Q_h)$
  end for
  $Q \leftarrow \text{jnp.stack}(Q)$
  $\pi \leftarrow \text{jnp.stack}(\pi)$
  $V \leftarrow \text{jnp.stack}(V_{:-1})$
  return (π, V, Q)
end function

Figure 2.13: Pseudocode for the dynamic programming algorithm.

At each of the H timesteps, we must compute $Q_h^\star$ for each of the $|\mathcal{S}||\mathcal{A}|$ state-action pairs. Each computation takes $|\mathcal{S}|$ operations to evaluate the average value over s'. Computing $\pi_h^\star$ and $V_h^\star$ then requires computing the value-maximizing action in each state, which is an additional $O(|\mathcal{S}|\mathcal{A}|)$ comparisons. This is dominated by the earlier term, resulting in a total computation time of $O(H \cdot |\mathcal{S}|^2 \cdot |\mathcal{A}|)$.

Note that this algorithm is identical to the policy evaluation algorithm (fig. 2.11), but instead of *averaging* over the actions chosen by a policy, we instead take a *maximum* over the action-values. We'll see this relationship between **policy evaluation** and **optimal policy computation** show up again in the infinite-horizon setting.

```
"Assertions passed (the 'tidy when messy' policy is optimal)"
```

Let us review some equivalent definitions of an optimal policy:

Remark 2.5 (Equivalent definitions of an optimal policy). Let π be a stationary policy. Then the following are all equivalent:

1. π is optimal across history-independent policies (def. 2.11).
2. π is optimal across all policies (eq. 2.24).
3. $V^\pi = V^\star$.
4. $Q^\pi = Q^\star$.
5. V^π satisfies the Bellman optimality equations (eq. 2.38).

1 and 3 are the same by definition. 3 and 4 are the same since $V^\star$ and $Q^\star$ uniquely define each other by eq. 2.17 and V^π and Q^π uniquely define each other by eq. 2.7 and eq. 2.8. 3 and 5

are the same by the Bellman optimality equations (Theorem 2.5). 1 and 2 are the same as shown in the proof of Theorem 2.6.

Remark 2.6 (Why stochastic policies). Given that there exists an optimal deterministic policy, why do we try ever try to learn stochastic policies? We will see a partial answer to this when we discuss the *exploration-exploitation* tradeoff in Chapter 4. So far, we've assumed that the environment is totally known. If it isn't, however, we need some way to explore different possible actions, and stochastic policies provide a natural way to do this.

2.4 Infinite-horizon Markov decision processes

What if we allow trajectories to continue for an infinite amount of time, that is, set the time horizon H to infinity? This might seem impractical, since in the real world, most of the trajectories we care about terminate after some fixed number of steps. However, we will see that **infinite-horizon** MDPs are simpler in certain regards and can serve as good approximations to finite-horizon tasks. A crucial result is that Bellman operators (def. 2.10) in this setting are **contraction mappings** (Theorem 2.8), which yield simple fixed-point iteration algorithms for policy evaluation (sec. 2.4.3.2) and computing the optimal value function (sec. 2.4.4.1). Finally, we'll present the policy iteration algorithm (sec. 2.4.4.2). In addition to being an effective tool in its own right, we will find that policy iteration serves as a useful *framework* for designing and analyzing later algorithms.

2.4.1 Differences from the finite horizon setting

Remark 2.7 (Discounted rewards). First of all, note that the total reward $r_0 + r_1 + \cdots$ might blow up to infinity. To ensure that we work with bounded quantities, we insert a **discount factor** $\gamma \in [0, 1)$ such that rewards become less valuable the further into the future they are:

$$r_0 + \gamma r_1 + \gamma^2 r_2 + \cdots = \sum_{h=0}^{\infty} \gamma^h r_h \leq R \frac{1}{1-\gamma}, \tag{2.28}$$

where R is the maximum possible reward. We can think of γ as measuring how much we care about the future: if it's close to 0, we only care about the near-term rewards; if it's close to 1, we put more weight into future rewards.

You can also analyze γ as the *probability of continuing* the trajectory at each time step. (This is equivalent to H being distributed by a First Success distribution with success probability γ.) This accords with the above interpretation: if γ is close to 0, the trajectory will likely be very short, while if γ is close to 1, the trajectory will likely continue for a long time.

The other components of the MDP remain from the finite-horizon setting (def. 2.4):

$$\mathcal{M} = (\mathcal{S}, \mathcal{A}, P_0, P, r, \gamma).$$

Remark 2.8 (Stationary policies). Time-dependent policies become difficult to handle in the infinite-horizon case. Not only would they be computationally impossible to store, but also, many of the DP approaches we saw required us to start at the end of the trajectory, which is no longer possible. We'll shift to **stationary** policies $\pi : \mathcal{S} \to \mathcal{A}$ (deterministic) or $\triangle(\mathcal{A})$ (stochastic), which don't explicitly depend on the timestep.

Exercise 2.6 (Stationary policy examples)**.** Which of the policies in ex. 2.4 (policies in the tidying MDP) are stationary?

Definition 2.14 (Value functions)**.** We'll also consider stationary value functions $V^\pi : \mathcal{S} \to \mathbb{R}$ and $Q^\pi : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$. These now represent the expected total *discounted* reward:

$$\begin{aligned} V^\pi(s) &= \mathop{\mathbb{E}}_{\tau \sim \rho^\pi}[r_0 + \gamma r_1 + \gamma^2 r_2 + \cdots \mid s_0 = s] \\ Q^\pi(s,a) &= \mathop{\mathbb{E}}_{\tau \sim \rho^\pi}[r_0 + \gamma r_1 + \gamma^2 r_2 + \cdots \mid s_0 = s, a_0 = a] \end{aligned} \tag{2.29}$$

Exercise 2.7 (Time-independent)**.** Note that we add up the rewards starting from time 0, whereas in the finite-horizon case, we needed to explicitly add the rewards from time h onwards. Heuristically speaking, why does it no longer matter which time step we start on when defining the value function? Refer back to the Markov property (def. 2.5).

Theorem 2.7 (Bellman consistency equations)**.** *The Bellman consistency equations play the same role they did in the finite-horizon setting (Theorem 2.3). We need to insert a factor of γ to account for the discounting, and the h subscript is gone since V^π is stationary:*

$$V^\pi(s) = \mathop{\mathbb{E}}_{\substack{a \sim \pi(s) \\ s' \sim P(s,a)}}[r(s,a) + \gamma V^\pi(s')]. \tag{2.30}$$

The value function and action-value function are still related in the same way:

$$\begin{aligned} V^\pi(s) &= \mathop{\mathbb{E}}_{a \sim \pi(s)}[Q(s,a)] \\ Q^\pi(s,a) &= r(s,a) + \gamma \mathop{\mathbb{E}}_{s' \sim P(s,a)}[V^\pi(s')]. \end{aligned} \tag{2.31}$$

2.4.2 Contraction mappings

Recall that a policy's *Bellman operator* takes a guess for the policy's value function and returns an *improved* guess using one step of the policy (def. 2.10). In the infinite-horizon setting, this has the same form:

Definition 2.15 (Bellman operator)**.** Let $\pi : \mathcal{S} \to \triangle(\mathcal{A})$ be a stationary policy. Its **Bellman operator** is defined

$$\mathcal{J}^\pi(v) := \Big(s \mapsto \mathop{\mathbb{E}}_{\substack{a\sim\pi(s)\\ s'\sim P(s,a)}} [r(s,a) + \gamma v(s')]\Big). \tag{2.32}$$

The crucial property of the Bellman operator is that it is a **contraction mapping** for any policy. Intuitively, if we start with two guesses $v, u : \mathcal{S} \to \mathbb{R}$, if we repeatedly apply the Bellman operator to each of them, they will get closer and closer together at an exponential rate.

Definition 2.16 (Contraction mapping)**.** Let X be a set with a norm $\|\cdot\|$. We call an operator $f : X \to X$ a **contraction mapping** if for any $x, y \in X$,

$$\|f(x) - f(y)\| \leq \gamma \|x - y\|$$

for some fixed $\gamma \in (0, 1)$.

Exercise 2.8 (Contraction mappings pull points together)**.** Show that for a contraction mapping f with coefficient γ, for all $t \in \mathbb{N}$,

$$\|f^{(t)}(x) - f^{(t)}(y)\| \leq \gamma^t \|x - y\|, \tag{2.33}$$

i.e. any two points will be pushed closer by at least a factor of γ at each iteration.

It is a powerful fact, known as the **Banach fixed-point theorem**, that every contraction mapping has a unique **fixed point** $x^\star$ such that $f(x^\star) = x^\star$. This means that if we repeatedly apply f to any starting point, we will eventually converge to $x^\star$:

$$\|f^{(t)}(x) - x^\star\| \leq \gamma^t \|x - x^\star\|. \tag{2.34}$$

Let's return to the RL setting and apply this result to the Bellman operator. How can we measure the distance between two functions $v, u : \mathcal{S} \to \mathbb{R}$? We'll take the **supremum norm** as our distance metric:

$$\|v - u\|_\infty := \sup_{s \in \mathcal{S}} |v(s) - u(s)|,$$

i.e. we compare the functions on the state that causes the biggest gap between them. The Bellman consistency equations (Theorem 2.7) state that V^π is the fixed point of $\mathcal{J}^\pi$. Then eq. 2.34 implies that if we repeatedly apply $\mathcal{J}^\pi$ to any starting guess, we will eventually converge to V^π:

$$\|(\mathcal{J}^\pi)^{(t)}(v) - V^\pi\|_\infty \leq \gamma^t \|v - V^\pi\|_\infty. \tag{2.35}$$

We'll use this useful fact to prove the convergence of several algorithms later on.

Theorem 2.8 (The Bellman operator is a contraction mapping)**.** *Let π be a stationary policy. There exists $\gamma \in (0, 1)$ such that for any two functions $u, v : \mathcal{S} \to \mathbb{R}$,*

$$\|\mathcal{J}^\pi(v) - \mathcal{J}^\pi(u)\|_\infty \leq \gamma \|v - u\|_\infty.$$

Proof. For all states $s \in \mathcal{S}$,

$$\begin{aligned}
|[\mathcal{J}^\pi(v)](s) - [\mathcal{J}^\pi(u)](s)| &= \Big| \mathop{\mathbb{E}}_{a \sim \pi(s)} \left[r(s,a) + \gamma \mathop{\mathbb{E}}_{s' \sim P(s,a)} v(s') \right] \\
&\qquad - \mathop{\mathbb{E}}_{a \sim \pi(s)} \left[r(s,a) + \gamma \mathop{\mathbb{E}}_{s' \sim P(s,a)} u(s') \right] \Big| \\
&= \gamma \left| \mathop{\mathbb{E}}_{s' \sim P(s,a)} [v(s') - u(s')] \right| \\
&\leq \gamma \mathop{\mathbb{E}}_{s' \sim P(s,a)} |v(s') - u(s')| \qquad \text{(Jensen's inequality)} \\
&\leq \gamma \max_{s'} |v(s') - u(s')| \\
&= \gamma \|v - u\|_\infty.
\end{aligned}$$

□

2.4.3 Policy evaluation

The backwards DP technique we used in the finite-horizon case (sec. 2.3.1) no longer works since there is no "final timestep" to start from. We'll need another approach to policy evaluation.

The Bellman consistency equations (Theorem 2.7) yield a system of $|\mathcal{S}|$ equations we can solve to evaluate a deterministic policy *exactly*. For a faster approximate solution, we can iterate the policy's Bellman operator, since we know that it has a unique fixed point at the true value function.

2.4.3.1 Matrix inversion for deterministic policies

Note that when the policy π is deterministic, the actions can be determined from the states, and so we can chop off the action dimension for the rewards and state transitions:

$$\begin{array}{lll} r^\pi \in \mathbb{R}^{|\mathcal{S}|} & P^\pi \in [0,1]^{|\mathcal{S}|\times|\mathcal{S}|} & P_0 \in [0,1]^{|\mathcal{S}|} \\ \pi \in \mathcal{A}^{|\mathcal{S}|} & V^\pi \in \mathbb{R}^{|\mathcal{S}|} & Q^\pi \in \mathbb{R}^{|\mathcal{S}|\times|\mathcal{A}|}. \end{array}$$

For P^π, we'll treat the rows as the states and the columns as the next states. Then $P^\pi_{s,s'}$ is the probability of transitioning from state s to state s' under policy π.

Example 2.8 (Tidying MDP)**.** The tabular MDP from ex. 2.3 has $|\mathcal{S}| = 2$ and $|\mathcal{A}| = 2$. Let's write down the quantities for the policy π that tidies if and only if the room is messy:

$$r^\pi = \begin{bmatrix} 1 \\ 0 \end{bmatrix}, \quad P^\pi = \begin{bmatrix} 0.7 & 0.3 \\ 1 & 0 \end{bmatrix}, \quad P_0 = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$$

We'll see how to evaluate this policy in the next section.

The Bellman consistency equation for a deterministic policy can be written in tabular notation as

$$V^\pi = r^\pi + \gamma P^\pi V^\pi.$$

(Unfortunately, this notation doesn't simplify the expression for Q^π.) This system of equations can be solved with a matrix inversion:

$$V^\pi = (I - \gamma P^\pi)^{-1} r^\pi. \tag{2.36}$$

Exercise 2.9 (Matrix invertibility)**.** Note we've assumed that $I - \gamma P^\pi$ is invertible. Can you see why this is the case? (Recall that a linear operator, i.e. a square matrix, is invertible if and only if its null space is trivial; that is, it doesn't map any nonzero vector to zero. Show that $I - \gamma P^\pi$ maps any nonzero vector to another nonzero vector.)

Example 2.9 (Tidying policy evaluation)**.** Let's use the same policy π that tidies if and only if the room is messy. Setting $\gamma = 0.95$, we must invert

$$I - \gamma P^\pi = \begin{bmatrix} 1 - 0.95 \times 0.7 & -0.95 \times 0.3 \\ -0.95 \times 1 & 1 - 0.95 \times 0 \end{bmatrix} = \begin{bmatrix} 0.335 & -0.285 \\ -0.95 & 1 \end{bmatrix}.$$

The inverse to two decimal points is

$$(I - \gamma P^\pi)^{-1} = \begin{bmatrix} 15.56 & 4.44 \\ 14.79 & 5.21 \end{bmatrix}.$$

Thus the value function is

$$V^\pi = (I - \gamma P^\pi)^{-1} r^\pi = \begin{bmatrix} 15.56 & 4.44 \\ 14.79 & 5.21 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 15.56 \\ 14.79 \end{bmatrix}.$$

Let's sanity-check this result. Since rewards are at most 1, the maximum cumulative return of a trajectory is at most $1/(1-\gamma) = 20$. We see that the value function is indeed slightly lower than this.

```
Array([15.56419, 14.78598], dtype=float32)
```

2.4.3.2 Iterative policy evaluation

The matrix inversion above takes roughly $O(|\mathcal{S}|^3)$ time. It also only works for deterministic policies. Can we trade off the requirement of finding the *exact* value function for a faster *approximate* algorithm that will also extend to stochastic policies?

Let's use the Bellman operator to define an iterative algorithm for computing the value function. We'll start with an initial guess $v^{(0)}$ with elements in $[0, 1/(1-\gamma)]$ and then iterate the Bellman operator:

$$v^{(t+1)} = \mathcal{J}^\pi(v^{(t)}),$$

i.e. $v^{(t)} = (\mathcal{J}^\pi)^{(t)}(v^{(0)})$. Note that each iteration takes $O(|\mathcal{S}|^2)$ time for the matrix-vector multiplication.

Then, as we showed in eq. 2.35, by the Banach fixed-point theorem:

$$\|v^{(t)} - V^\pi\|_\infty \leq \gamma^t \|v^{(0)} - V^\pi\|_\infty.$$

```
Array([15.564166, 14.785956], dtype=float32)
```

Remark 2.9 (Convergence of iterative policy evaluation). How many iterations do we need for an ϵ-accurate estimate? We can work backwards to solve for t (note that $\log \gamma < 0$):

$$\begin{aligned} \gamma^t \|v^{(0)} - V^\pi\|_\infty &\leq \epsilon \\ t &\geq \frac{\log(\epsilon / \|v^{(0)} - V^\pi\|_\infty)}{\log \gamma} \\ &= \frac{\log(\|v^{(0)} - V^\pi\|_\infty / \epsilon)}{\log(1/\gamma)}, \end{aligned}$$

and so the number of iterations required for an ϵ-accurate estimate is

$$T = O\left(\frac{1}{1-\gamma} \log\left(\frac{1}{\epsilon(1-\gamma)}\right)\right).$$

Note that we've applied the inequalities $\|v^{(0)} - V^\pi\|_\infty \leq 1/(1-\gamma)$ and $\log(1/x) \geq 1 - x$.

2.4.4 Optimality

Now let's move on to solving for an optimal policy in the infinite-horizon case. As in def. 2.11, an **optimal policy** $\pi^\star$ is one that does at least as well as any other policy in all situations. That is, for all policies π, times $h \in \mathbb{N}$, and initial trajectories $\tau_{\leq h} = (s_0, a_0, r_0, \dots, s_h)$,

$$\begin{aligned} &\mathop{\mathbb{E}}_{\tau \sim \rho^{\pi^\star}}[r_h + \gamma r_{h+1} + \gamma^2 r_{h+2} + \cdots \mid \tau_{\leq h}] \\ \geq &\mathop{\mathbb{E}}_{\tau \sim \rho^{\pi}}[r_h + \gamma r_{h+1} + \gamma^2 r_{h+2} + \cdots \mid \tau_{\leq h}] \end{aligned} \tag{2.37}$$

Once again, all optimal policies share the same **optimal value function** $V^\star$, and the greedy policy with respect to this value function is optimal.

Exercise 2.10 (The greedy optimal policy)**.** Verify this by modifying the proof Theorem 2.6 from the finite-horizon case.

So how can we compute such an optimal policy? We can't use the backwards DP approach from the finite-horizon case fig. 2.13 since there's no "final timestep" to start from. Instead, we'll exploit the fact that the Bellman consistency equation eq. 2.29 for the optimal value function doesn't depend on any policy:

$$V^\star(s) = \max_a \left[r(s,a) + \gamma \mathop{\mathbb{E}}_{s' \sim P(s,a)} V^\star(s'). \right] \tag{2.38}$$

As before, thinking of the r.h.s. of eq. 2.38 as an operator on value functions gives the **Bellman optimality operator**

$$[\mathcal{J}^\star(v)](s) = \max_a \left[r(s,a) + \gamma \mathop{\mathbb{E}}_{s' \sim P(s,a)} v(s') \right] \tag{2.39}$$

2.4.4.1 Value iteration

Since the optimal policy is still a policy, our result that the Bellman operator is a contracting map still holds, and so we can repeatedly apply this operator to converge to the optimal value function! This algorithm is known as **value iteration**.

```
Array([15.564166, 14.785956], dtype=float32)
```

Note that the runtime analysis for an ϵ-optimal value function is exactly the same as sec. 2.4.3.2! This is because value iteration is simply the special case of applying iterative policy evaluation to the *optimal* value function.

As the final step of the algorithm, to return an actual policy $\hat{\pi}$, we can simply act greedily with respect to the final iteration $v^{(T)}$ of our above algorithm:

$$\hat{\pi}(s) = \arg\max_a \left[r(s,a) + \gamma \mathop{\mathbb{E}}_{s' \sim P(s,a)} v^{(T)}(s') \right]. \tag{2.40}$$

We must be careful, though: the value function of this greedy policy, $V^{\hat{\pi}}$, is *not* the same as $v^{(T)}$, which need not even be a well-defined value function for some policy!

The bound on the policy's quality is actually quite loose: if $\|v^{(T)} - V^\star\|_\infty \leq \epsilon$, then the greedy policy $\hat{\pi}$ satisfies $\|V^{\hat{\pi}} - V^\star\|_\infty \leq \frac{2\gamma}{1-\gamma}\epsilon$, which might potentially be very large.

Theorem 2.9 (Greedy policy value worsening)**.**

$$\|V^{\hat{\pi}} - V^\star\|_\infty \leq \frac{2\gamma}{1-\gamma}\|v - V^\star\|_\infty$$

where $\hat{\pi}(s) = \arg\max_a q(s,a)$ *is the greedy policy with respect to*

$$q(s,a) = r(s,a) + \mathop{\mathbb{E}}_{s' \sim P(s,a)} v(s').$$

Proof. We first have

$$\begin{aligned} V^\star(s) - V^{\hat{\pi}}(s) &= Q^\star(s, \pi^\star(s)) - Q^{\hat{\pi}}(s, \hat{\pi}(s)) \\ &= [Q^\star(s, \pi^\star(s)) - Q^\star(s, \hat{\pi}(s))] + [Q^\star(s, \hat{\pi}(s)) - Q^{\hat{\pi}}(s, \hat{\pi}(s))]. \end{aligned}$$

Let us bound these two quantities separately.

For the first quantity, note that by the definition of $\hat{\pi}$, we have

$$q(s, \hat{\pi}(s)) \geq q(s, \pi^\star(s)).$$

Let's add $q(s, \hat{\pi}(s)) - q(s, \pi^\star(s)) \geq 0$ to the first term to get

$$\begin{aligned} Q^\star(s, \pi^\star(s)) - Q^\star(s, \hat{\pi}(s)) &\leq [Q^\star(s, \pi^\star(s)) - q(s, \pi^\star(s))] + [q(s, \hat{\pi}(s)) - Q^\star(s, \hat{\pi}(s))] \\ &= \gamma \mathop{\mathbb{E}}_{s' \sim P(s, \pi^\star(s))} [V^\star(s') - v(s')] + \gamma \mathop{\mathbb{E}}_{s' \sim P(s, \hat{\pi}(s))} [v(s') - V^\star(s')] \\ &\leq 2\gamma \|v - V^\star\|_\infty. \end{aligned}$$

The second quantity is bounded by

$$\begin{aligned} Q^\star(s, \hat{\pi}(s)) - Q^{\hat{\pi}}(s, \hat{\pi}(s)) &= \gamma \mathop{\mathbb{E}}_{s' \sim P(s, \hat{\pi}(s))} \left[V^\star(s') - V^{\hat{\pi}}(s')\right] \\ &\leq \gamma \|V^\star - V^{\hat{\pi}}\|_\infty \end{aligned}$$

and thus

$$\begin{aligned} \|V^\star - V^{\hat{\pi}}\|_\infty &\leq 2\gamma \|v - V^\star\|_\infty + \gamma \|V^\star - V^{\hat{\pi}}\|_\infty \\ \|V^\star - V^{\hat{\pi}}\|_\infty &\leq \frac{2\gamma \|v - V^\star\|_\infty}{1 - \gamma}. \end{aligned}$$

□

So in order to compensate and achieve $\|V^{\hat{\pi}} - V^\star\| \leq \epsilon$, we must have

$$\|v^{(T)} - V^\star\|_\infty \leq \frac{1 - \gamma}{2\gamma} \epsilon.$$

This means, using Remark 2.9, we need to run value iteration for

$$T = O\left(\frac{1}{1 - \gamma} \log\left(\frac{\gamma}{\epsilon(1 - \gamma)^2}\right)\right)$$

iterations to achieve an ϵ-accurate estimate of the optimal value function.

Table 2.4: Two action-value functions that result in the same greedy policy

(a) One Q-function

s	$a=0$	$a=1$
A	0.1	0.2
B	0.8	0.9
C	0.4	0.5

(b) Another Q-function

s	$a=0$	$a=1$
A	0.1	0.3
B	0.8	1
C	0.4	0.6

2.4.4.2 Policy iteration

Can we mitigate this "greedy worsening"? What if instead of approximating the optimal value function and then acting greedily by it at the very end, we iteratively improve the policy and value function *together*? This is the idea behind **policy iteration**. In each step, we simply set the policy to act greedily with respect to its own value function.

```
Array([[1., 0.],
       [0., 1.]], dtype=float32)
```

Although PI appears more complex than VI, we'll use Theorem 2.8 to show convergence. This will give us the same runtime bound as value iteration and iterative policy evaluation for an ϵ-optimal value function (Remark 2.9), although in practice, PI often converges in fewer iterations. Why so? Intuitively, by iterating through actual policies, we can "skip over" different functions that represent the same policy, which value iteration might iterate through. For a concrete example, suppose we have an MDP with three states $\mathcal{S} = \{\mathrm{A}, \mathrm{B}, \mathrm{C}\}$. Compare the two functions below:

These both map to the same greedy policy, so policy iteration would never iterate through both of these functions, while value iteration might.

Theorem 2.10 (Policy Iteration runtime and convergence)**.** *The number of iterations required for an ϵ-accurate estimate of the optimal value function is*

$$T = O\left(\frac{1}{1-\gamma}\log\left(\frac{1}{\epsilon(1-\gamma)}\right)\right).$$

Proof. This bound follows from the contraction property eq. 2.35:

$$\|V^{\pi^{t+1}} - V^\star\|_\infty \leq \gamma \|V^{\pi^t} - V^\star\|_\infty.$$

We'll prove that the iterates of PI respect the contraction property by showing that the policies improve monotonically:

$$V^{\pi^{t+1}}(s) \geq V^{\pi^t}(s).$$

Then we'll use this to show $V^{\pi^{t+1}}(s) \geq [\mathcal{J}^\star(V^{\pi^t})](s)$. Note that

$$\begin{aligned}(s) &= \max_a \left[r(s,a) + \gamma \mathop{\mathbb{E}}_{s' \sim P(s,a)} V^{\pi^t}(s') \right] \\ &= r(s, \pi^{t+1}(s)) + \gamma \mathop{\mathbb{E}}_{s' \sim P(s,\pi^{t+1}(s))} V^{\pi^t}(s')\end{aligned}$$

Since $[\mathcal{J}^\star(V^{\pi^t})](s) \geq V^{\pi^t}(s)$, we then have

$$\begin{aligned}V^{\pi^{t+1}}(s) - V^{\pi^t}(s) &\geq V^{\pi^{t+1}}(s) - \mathcal{J}^\star(V^{\pi^t})(s) \\ &= \gamma \mathop{\mathbb{E}}_{s' \sim P(s,\pi^{t+1}(s))} \left[V^{\pi^{t+1}}(s') - V^{\pi^t}(s') \right].\end{aligned} \tag{2.41}$$

But note that the expression being averaged is the same as the expression on the l.h.s. with s replaced by s'. So we can apply the same inequality recursively to get

$$\begin{aligned}V^{\pi^{t+1}}(s) - V^{\pi^t}(s) &\geq \gamma \mathop{\mathbb{E}}_{s' \sim P(s,\pi^{t+1}(s))} \left[V^{\pi^{t+1}}(s') - V^{\pi^t}(s') \right] \\ &\geq \gamma^2 \mathop{\mathbb{E}}_{\substack{s' \sim P(s,\pi^{t+1}(s)) \\ s'' \sim P(s',\pi^{t+1}(s'))}} \left[V^{\pi^{t+1}}(s'') - V^{\pi^t}(s'') \right] \\ &\geq \cdots\end{aligned}$$

which implies that $V^{\pi^{t+1}}(s) \geq V^{\pi^t}(s)$ for all s (since the r.h.s. converges to zero). We can then plug this back into eq. 2.41 to get the desired result:

$$\begin{aligned}V^{\pi^{t+1}}(s) - \mathcal{J}^\star(V^{\pi^t})(s) &= \gamma \mathop{\mathbb{E}}_{s' \sim P(s,\pi^{t+1}(s))} \left[V^{\pi^{t+1}}(s') - V^{\pi^t}(s') \right] \\ &\geq 0 \\ V^{\pi^{t+1}}(s) &\geq [\mathcal{J}^\star(V^{\pi^t})](s)\end{aligned}$$

This means we can now apply the Bellman convergence result eq. 2.35 to get

$$\|V^{\pi^{t+1}} - V^\star\|_\infty \leq \|\mathcal{J}^\star(V^{\pi^t}) - V^\star\|_\infty \leq \gamma \|V^{\pi^t} - V^\star\|_\infty.$$

□

2.5 Key takeaways

Markov decision processes (MDPs) are a framework for sequential decision making under uncertainty. They consist of a state space $\mathcal{S}$, an action space $\mathcal{A}$, an initial state distribution $P_0 \in \triangle(\mathcal{S})$, a transition function $P(s' \mid s, a)$, and a reward function $r(s, a)$. They can be finite-horizon (ends after H timesteps) or infinite-horizon (where rewards scale by $\gamma \in (0, 1)$ at each timestep). Our goal is to find a policy π that maximizes expected total reward. Policies can be **deterministic** or **stochastic**, **history-dependent** or **history-independent**, **stationary** or **time-dependent**. A policy induces a distribution over **trajectories**.

We can evaluate a policy by computing its **value function** $V^\pi(s)$, which is the expected total reward starting from state s and following policy π. We can also compute the **state-action value function** $Q^\pi(s, a)$, which is the expected total reward starting from state s, taking action a, and then following policy π. In the finite-horizon setting, these also depend on the timestep h.

The **Bellman consistency equation** is an equation that the value function must satisfy. It can be used to solve for the value functions exactly. Thinking of the r.h.s. of this equation as an operator on value functions gives the **Bellman operator**.

In the finite-horizon setting, we can compute the optimal policy using **dynamic programming**. In the infinite-horizon setting, we can compute the optimal policy using **value iteration** or **policy iteration**.

2.6 Bibliographic notes and further reading

The MDP framework can be traced to Puterman (1994). The proof of Theorem 2.6 can be found in A. Agarwal et al. (2022).

MDPs are the most common framework used throughout RL. See sec. 1.4 for a list of popular textbooks on RL.

In most real-world problems, we don't observe the entire state of the environment. Instead, we only get access to what our senses can perceive. This is what makes games like hide-and-seek enjoyable. We can model such environment as **partially observed MDPs** (POMDPs). Powell (2022) presents a unified framework for sequential decision problems.

Another important bandit algorithm is **Gittins indices** (Gittins, 2011). The derivation of this algorithm is out of scope of this course.

3 Linear Quadratic Regulators

3.1 Introduction

In Chapter 2, we considered decision problems with finitely many states and actions. However, in many applications, states and actions may take on *continuous* values. For example, consider autonomous driving, controlling a robot's joints, and automated manufacturing. How can we teach computers to solve these kinds of problems? This is the task of **continuous control**.

(a) Solving a Rubik's Cube with a robot hand

(a) Boston Dynamics's Spot robot

Aside from the change in the state and action spaces, the general problem setup remains the same: we seek to construct an *optimal policy* that outputs actions to solve the desired task. We will see that many key ideas and algorithms, in particular dynamic programming algorithms (sec. 2.3.2), carry over to this new setting.

This chapter introduces a fundamental tool to solve a simple class of continuous control problems: the **linear quadratic regulator** (LQR). We can use the LQR model as a building block for solving more complex problems.

Remark 3.1 (Control vs RL). Control theory is often considered a distinct, though related, field from RL. In both fields, the central aim is to solve sequential decision problems, i.e. find a strategy for taking actions in the environment in order to achieve some goal that is measured by a scalar signal. The two fields arose rather independently and with rather different problem settings, and use different mathematical terminology as a result.

Control theory has close ties to electrical and mechanical engineering. Control theorists typically work in a *continuous-time* setting in which the dynamics can be described by systems of differential equations. The goal is typically to ensure *stability* of the system and minimize some notion of "cost" (e.g. wasted energy in controlling the system). Rather than learning the system from data, one typically supposes a particular structure of the environment and solves for the optimal controller using analytical or numerical methods. As such, most control theory algorithms, like the ones we will explore in this chapter, are *planning methods* that do not require learning from data.

3.2 Optimal control

Let's first look at a simple example of a continuous control problem:

Example 3.1 (CartPole)**.** Try to balance a pencil on its point on a flat surface. It's much more difficult than it may first seem: the position of the pencil varies continuously, and the state transitions governing the system, i.e. the laws of physics, are highly complex. This task is equivalent to the classic control problem known as *CartPole*:

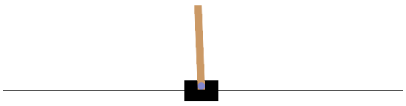


Figure 3.3: A snapshot of the cart pole environment

The state $x \in \mathbb{R}^4$ can be described by four real variables:

1. the position of the cart;
2. the velocity of the cart;
3. the angle of the pole;
4. the angular velocity of the pole.

We can *control* the cart by applying a horizontal force $u \in \mathbb{R}$.

Goal: Stabilize the cart around an ideal state and action $(x^\star, u^\star)$.

A continuous control environment is a special case of an MDP (def. 2.4). Recall that a finite-horizon MDP

$$\mathcal{M} = (\mathcal{S}, \mathcal{A}, P_0, P, r, H). \tag{3.1}$$

is defined by its state space $\mathcal{S}$, action space $\mathcal{A}$, initial state distribution P_0, state transitions P, reward function r, and time horizon H. These each have equivalents in the control setting.

Definition 3.1 (Continuous control environment)**.** A continuous control environment is defined by the following components:

- The state and action spaces $\mathcal{S}, \mathcal{A}$ are *continuous* rather than finite. That is, $\mathcal{S} \subseteq \mathbb{R}^{n_x}$ and $\mathcal{A} \subseteq \mathbb{R}^{n_u}$, where n_x and n_u are the number of coordinates required to specify a single state or action respectively. For example, in robotic control, n_x might be the number of sensors and n_u might be the number of actuators on the robot.
- $P_0 \in \triangle(\mathcal{S})$ denotes the initial state distribution.
- We call the state transitions the **dynamics** of the system and denote them by f:

$$x_{h+1} = f_h(x_h, u_h, w_h), \tag{3.2}$$

  where w_h denotes noise drawn from the distribution $\nu_h \in \triangle(\mathbb{R}^{n_w})$. We allow the dynamics to vary at each timestep h.
- Instead of maximizing the reward function, we seek to minimize the **cost function** $c_h : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$. Often, the cost function describes *how far away* we are from a **target state-action pair** $(x^\star, u^\star)$.
- The agent acts over a *finite time horizon* $H \in \mathbb{N}$.

Together, these constitute a description of the environment

$$\mathcal{M} = (\mathcal{S}, \mathcal{A}, P_0, f, \nu, c, H). \tag{3.3}$$

Definition 3.2 (Optimal control)**.** For a continuous control environment $\mathcal{M}$, the optimal control problem is to compute a policy π that minimizes the total cost

$$\begin{aligned} \min_{\pi_0,\ldots,\pi_{H-1}:\mathcal{S}\to\mathcal{A}} \quad & \mathbb{E}\left[\sum_{h=0}^{H-1} c_h(x_h, u_h)\right] \\ \text{where} \quad & x_{h+1} = f_h(x_h, u_h, w_h), \\ & u_h = \pi_h(x_h) \\ & x_0 \sim P_0 \\ & w_h \sim \nu_h. \end{aligned} \tag{3.4}$$

In this chapter, we will only consider *deterministic, time-dependent* policies

$$\pi = (\pi_0, \dots, \pi_{H-1}) \quad \text{where} \quad \pi_h : \mathcal{S} \to \mathcal{A} \text{ for each } h \in [H]. \tag{3.5}$$

To make the explicit conditioning more concise, we let ρ^π denote the trajectory distribution induced by the policy π. That is, if we sample initial states from P_0, act according to π, and transition according to the dynamics f, the resulting distribution over trajectories is ρ^π. Put another way, the procedure for sampling from ρ^π is to take rollouts in the environment using π.

Theorem 3.1 (Trajectory distributions)**.** *Let $\pi = \{\pi_0, \dots, \pi_{H-1}\}$ be a deterministic, time-dependent policy. We include states x_h, actions u_h, and noise terms w_h in the trajectory. Then the density of a trajectory $\tau = (x_0, u_0, w_0, \dots, w_{H-2}, x_{H-1}, u_{H-1})$ can be expressed as follows:*

$$\begin{aligned}
\rho^\pi(\tau) &:= P_0(x_0) \times \mathbf{1}\{u_0 = \pi_0(x_0)\} \\
&\quad \times \nu(w_0) \times \mathbf{1}\{x_1 = f(x_0, u_0, w_0)\} \\
&\quad \times \cdots \\
&\quad \times \nu(w_{H-2}) \times \mathbf{1}\{x_{H-1} = f(x_{H-2}, u_{H-2}, w_{H-2})\} \times \mathbf{1}\{u_{H-1} = \pi_{H-1}(x_{H-1})\} \\
&= P_0(x_0)\mathbf{1}\{u_0 = \pi_0(x_0)\} \prod_{h=1}^{H-1} \nu(w_{h-1})\mathbf{1}\{x_h = f(x_{h-1}, u_{h-1}, w_{h-1})\}\mathbf{1}\{u_h = \pi_h(x_h)\}
\end{aligned} \tag{3.6}$$

This expression may seem intimidating, but on closer examination, it simply uses indicator variables to enforce that only valid trajectories have nonzero probability. Henceforth, we will write $\tau \sim \rho^\pi$ instead of the explicit conditioning in eq. 3.4.

As in Chapter 2, *value functions* will be crucial for constructing the optimal policy via **dynamic programming.**

Definition 3.3 (Value functions in LQR)**.** Given a policy $= (\pi_0, \dots, \pi_{H-1})$, we can define its value function $V_h^\pi : \mathcal{S} \to \mathbb{R}$ at time $h \in [H]$ as the **cost-to-go** incurred by that policy in expectation:

$$V_h^\pi(x) = \mathop{\mathbb{E}}_{\tau \sim \rho^\pi}\left[\sum_{h'=h}^{H-1} c(x_{h'}, u_{h'}) \mid x_h = x\right] \tag{3.7}$$

Definition 3.4 (Q function in LQR)**.** The Q-function additionally conditions on the first action we take:

$$Q_h^\pi(x,u) = \mathop{\mathbb{E}}_{\tau\sim\rho^\pi}\left[\sum_{h'=h}^{H-1} c(x_{h'},u_{h'}) \mid (x_h,u_h) = (x,u)\right] \tag{3.8}$$

Since we use a *cost function* c instead of a *reward function* r, the best policies in a given state (or state-action pair) are the ones with *smaller* V^π and Q^π.

3.2.1 A first attempt: Discretization

Can we solve this problem using tools from the finite MDP setting? If $\mathcal{S}$ and $\mathcal{A}$ were finite, then we could use dynamic programming (fig. 2.13) to compute the optimal policy exactly. This inspires us to try *discretizing* the problem.

Suppose $\mathcal{S}$ and $\mathcal{A}$ are bounded, that is, $\max_{x\in\mathcal{S}}\|x\| \le B_x$ and $\max_{u\in\mathcal{A}}\|u\| \le B_u$. To make $\mathcal{S}$ and $\mathcal{A}$ finite, let's choose some small positive ϵ, and simply round each coordinate to the nearest multiple of ϵ. For example, if $\epsilon = 0.01$, then we round each element of x and u to two decimal spaces.

What goes wrong? The discretized $\tilde{\mathcal{S}}$ and $\widetilde{\mathcal{A}}$ may be finite, but for all practical purposes, they are far too large: we must divide *each dimension* into intervals of length ε, resulting in the state and action spaces

$$|\tilde{\mathcal{S}}| = (B_x/\varepsilon)^{n_x} \text{ and } |\widetilde{\mathcal{A}}| = (B_u/\varepsilon)^{n_u}. \tag{3.9}$$

To get a sense of how quickly this grows, consider $\varepsilon = 0.01, n_x = n_u = 4$, and $B_x = B_u = 1$. Then the number of elements in the transition matrix would be $|\tilde{\mathcal{S}}|^2|\widetilde{\mathcal{A}}| = (100^4)^2(100^4) = 10^{24}$! (That's a million million million million.)

What properties of the problem could we instead make use of? Note that by discretizing the state and action spaces, we implicitly assumed that rounding each state or action vector by some tiny amount ε wouldn't change the behaviour of the system by much; namely, that the cost and dynamics were relatively *continuous*. Can we use this continuous structure in other ways? Yes! We will see that the **linear quadratic regulator** makes better use of this assumption.

3.3 The Linear Quadratic Regulator

The optimal control problem def. 3.2 is quite general. Is there a relevant simplification that we can solve, where the dynamics f and cost function c have some special structure? The **linear quadratic regulator** (LQR) is a special case of a continuous control environment (def. 3.1) where the dynamics f are *linear* and the cost function c is an *upward-curved quadratic*. We also assume that the environment is *time-invariant*. This is an important environment class in which the optimal policy and value function can be solved for exactly.

Definition 3.5 (Linear quadratic regulator)**.** The dynamics f are specified by the matrices $A \in \mathbb{R}^{n_x \times n_x}$ and $B \in \mathbb{R}^{n_x \times n_u}$:

$$x_{h+1} = f(x_h, u_h, w_h) = Ax_h + Bu_h + w_h \tag{3.10}$$

for all $h \in [H-1]$, where $w_h \sim \mathcal{N}(0, \sigma^2)I$.

The cost function is specified by the symmetric positive definite matrices $Q \in \mathbb{R}^{n_x \times n_x}$ and $R \in \mathbb{R}^{n_u \times n_u}$:

$$c(x_h, u_h) = x_h^\top Q x_h + u_h^\top R u_h \tag{3.11}$$

for all $h \in [H]$. We use the same letter as the Q-function (def. 3.4); the context should make it clear which is intended.

Remark 3.2 (Properties of LQR). Recall that w_h is a noise term that makes the dynamics random. By setting its covariance matrix to $\sigma^2 I$, we mean that there is Gaussian noise of standard deviation σ added independently to each coordinate of the state. Setting $\sigma = 0$ gives us **deterministic** state transitions. Surprisingly, the optimal policy doesn't depend on the amount of noise, although the optimal value function and Q-function do. (We will show this in a later derivation.)

The LQR cost function attempts to stabilize the state and action to $(x^\star, u^\star) = (0, 0)$. We require Q and R to both be *positive definite* matrices so that c has a unique minimum. We can furthermore assume without loss of generality that they are both *symmetric* (see exercise below). This greatly simplifies later computations.

Exercise 3.1 (Symmetric Q and R)**.** Show that replacing Q and R with $(Q + Q^\top)/2$ and $(R + R^\top)/2$ (which are symmetric) yields the same cost function.

Example 3.2 (Double integrator)**.** Consider a ball moving along a line. Let's first frame this as a continuous-time dynamical system, which is often more natural for physical phenomena, and then discretize it to solve it with the discrete-time LQR.

From elementary physics, the ball's velocity $\dot{p}(t)$ is the instantaneous change in its position, and its acceleration $\ddot{p}(t)$ is the instantaneous change in its velocity. Suppose we can apply a force $u(t)$ to accelerate the ball. This dynamical system is known as the **double integrator**. Our goal is to move the ball so that it is stationary at position $p^\star = 0$.

How would we frame this as an LQR problem? The effect of the control on the ball's position is nonlinear. However, if we define our state as including both the position and velocity, i.e.

$$x(t) = \begin{pmatrix} p(t) \\ \dot{p}(t) \end{pmatrix}, \tag{3.12}$$

then we end up with the following first-order linear differential equation:

$$\dot{x}(t) = \begin{pmatrix} 0 & 1 \\ 0 & 0 \end{pmatrix} x(t) + \begin{pmatrix} 0 \\ 1 \end{pmatrix} u(t). \tag{3.13}$$

There are many ways to turn this into a discrete-time problem. The simplest way is the (forward) *Euler method*, where we choose a small step size Δt, and explicitly use the limit definition of the derivative to obtain

$$x(t + \Delta t) \approx x(t) + \Delta t \cdot \dot{x}(t). \tag{3.14}$$

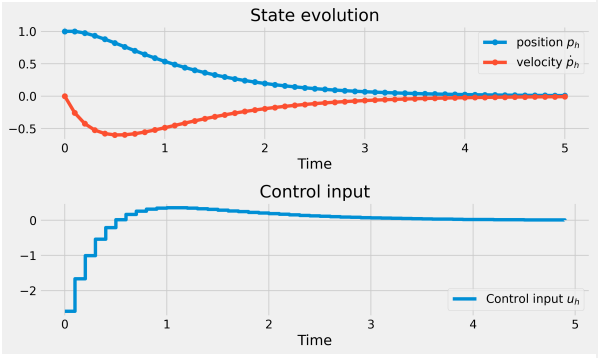
Applying this to eq. 3.13 gives the system

$$x_{h+1} = \begin{pmatrix} 1 & \Delta t \\ 0 & 1 \end{pmatrix} x_h + \begin{pmatrix} 0 \\ \Delta t \end{pmatrix} u_h, \tag{3.15}$$

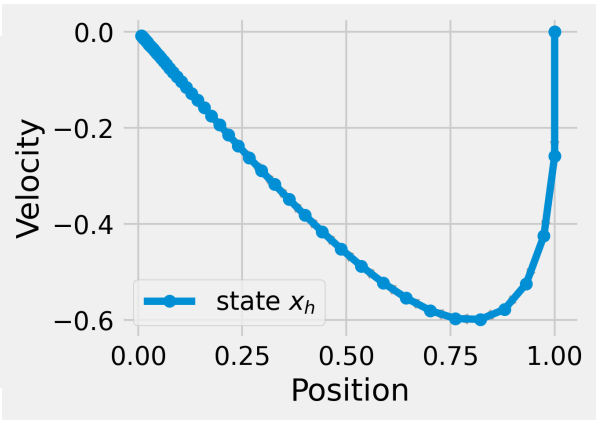
or explicitly in terms of the position and velocity,

$$\begin{aligned} p_{h+1} &= p_h + \Delta t \cdot \dot{p}_h, \\ \dot{p}_{h+1} &= \dot{p}_h + \Delta t \cdot u_h. \end{aligned} \tag{3.16}$$

We see that the LQR control smoothly controls the position of the ball.



(a) The state and control input over time.



(b) The trajectory of x_h plotted in the phase space.

Figure 3.4: Visualization of a double integrator LQR system.

3.4 Optimality and the Riccati Equation

In this section, we'll compute the optimal value function $V_h^\star$, Q-function $Q_h^\star$, and policy $\pi_h^\star$ for an LQR problem (def. 3.5) using **dynamic programming**. The algorithm is identical to the one in sec. 2.3.1, except here states and actions are vector-valued. Recall the definition of the optimal value function:

Definition 3.6 (Optimal value function in LQR)**.** The **optimal value function** is the one that, at any time and in any state, achieves *minimum cost* across all policies π that are *history-independent, time-dependent, and deterministic.*

$$\begin{aligned} V_h^\star(x) &= \min_\pi V_h^\pi(x) \\ &= \min_\pi \mathop{\mathbb{E}}_{\tau\sim\rho^\pi}\left[\sum_{h'=h}^{H-1}(x_{h'}^\top Q x_{h'} + u_{h'}^\top R u_{h'}) \mid x_h = x\right] \end{aligned} \tag{3.17}$$

Definition 3.7 (Optimal Q function in LQR)**.** The optimal Q-function is defined in the same way, conditioned on the starting state-action pair:

$$\begin{aligned} Q_h^\star(x,u) &= \min_\pi Q_h^\pi(x,u) \\ &= \min_\pi \mathbb{E}\left[\sum_{h'=h}^{H-1}(x_{h'}^\top Q x_{h'} + u_{h'}^\top R u_{h'}) \mid x_h = x, u_h = u\right] \end{aligned} \tag{3.18}$$

Remark 3.3 (Minimum over all policies). By Theorem 2.6, we do not lose any generality by only considering policies that are history-independent, time-dependent, and deterministic. That is,

the optimal policy within this class is also globally optimal across all policies, in the sense that it achieves lower cost for any given trajectory prefix.

The solution has very simple structure: $V_h^\star$ and $Q_h^\star$ are *upward-curved quadratics* and $\pi_h^\star$ is *linear* and furthermore does not depend on the amount of noise!

Theorem 3.2 (Optimal value function in LQR is an upward-curved quadratic)**.** *At each timestep $h \in [H]$,*

$$V_h^\star(x) = x^\top P_h x + p_h \tag{3.19}$$

for some symmetric, positive-definite $n_x \times n_x$ matrix P_h and scalar $p_h \in \mathbb{R}$.

Theorem 3.3 (Optimal policy in LQR is linear)**.** *At each timestep $h \in [H]$,*

$$\pi_h^\star(x) = -K_h x \tag{3.20}$$

for some $K_h \in \mathbb{R}^{n_u \times n_x}$. (The negative is due to convention.)

The construction (and inductive proof) proceeds similarly to the one in the MDP setting (sec. 2.3.1). We leave the full proof, which involves a significant amount of linear algebra, to sec. B.1.

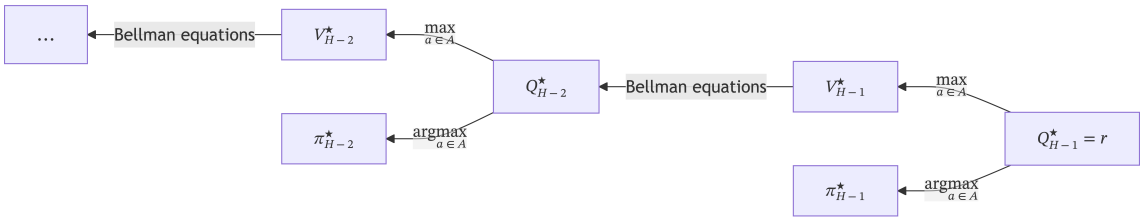


Figure 3.5: Illustrating a dynamic programming algorithm for computing the optimal policy in an LQR environment.

Here we provide the concrete P_h, p_h, and K_h used to compute the quantities above.

Definition 3.8 (Riccati equation)**.** It turns out that the matrices $P_0, \ldots, P_{H-1}$ used to define $V^\star$ obey a recurrence relation known as the **Riccati equation**:

$$P_h = Q + A^\top P_{h+1} A - A^\top P_{h+1} B (R + B^\top P_{h+1} B)^{-1} B^\top P_{h+1} A, \tag{3.21}$$

where $A \in \mathbb{R}^{n_x \times n_x}$ and $B \in \mathbb{R}^{n_x \times n_u}$ are the matrices used to define the dynamics f and $Q \in \mathbb{R}^{n_x \times n_x}$ and $R \in \mathbb{R}^{n_u \times n_u}$ are the matrices used to define the cost function c (def. 3.5).

The scalars p_h obey the recurrence relation

$$p_h = \mathrm{Tr}(\sigma^2 P_{h+1}) + p_{h+1} \tag{3.22}$$

and the matrices $K_0, \dots, K_{H-1}$ for defining the policy satisfy

$$K_h = (R + B^\top P_{h+1} B)^{-1} B^\top P_{h+1} A. \tag{3.23}$$

By setting $P_H = 0$ and $p_H = 0$,

Definition 3.9 (Dynamic programming in LQR)**.** Let A, B, Q, R be the matrices that define the LQR problem, and let σ^2 be the variance of the noise in the dynamics (def. 3.5).

1. Initialize $P_H = 0, p_H = 0$.
2. For each $h = H-1, \dots, 0$:
    1. Compute P_h using the Riccati equation (def. 3.8) and p_h using eq. 3.22.
    2. Compute K_h using eq. B.5.
    3. These define the functions $V_h^\star$ and $\pi_h^\star$ by Theorem 3.2 and Theorem 3.3.

3.4.1 Expected state at time h

Let's consider how the state at time h behaves when we act according to this optimal policy. In the field of control theory, which is often used in high-stakes circumstances such as aviation or architecture, it is important to understand the way the state evolves under the proposed controls.

How can we compute the expected state at time h when acting according to the optimal policy? Let's first express x_h in a cleaner way in terms of the history. Having linear dynamics makes it easy to expand terms backwards in time:

$$\begin{aligned} x_h &= Ax_{h-1} + Bu_{h-1} + w_{h-1} \\ &= A(Ax_{h-2} + Bu_{h-2} + w_{h-2}) + Bu_{h-1} + w_{h-1} \\ &= \cdots \\ &= A^h x_0 + \sum_{i=0}^{h-1} A^i (Bu_{h-i-1} + w_{h-i-1}). \end{aligned} \tag{3.24}$$

Let's consider the *average state* at this time, given all the past states and actions. Since we assume that $\mathbb{E}[w_h] = 0$ (this is the zero vector in d dimensions), when we take an expectation, the w_h term vanishes due to linearity, and so we're left with

$$\mathbb{E}[x_h \mid x_{0:(h-1)}, u_{0:(h-1)}] = A^h x_0 + \sum_{i=0}^{h-1} A^i B u_{h-i-1}. \tag{3.25}$$

Exercise 3.2 (Expected state)**.** Show that if we choose actions according to the optimal policy Lemma B.2, eq. 3.25 becomes

$$\mathbb{E}[x_h \mid x_0, u_i = \pi_i^\star(x_i) \quad \forall i \leq h] = \left(\prod_{i=0}^{h-1}(A - BK_i)\right) x_0.$$

This introdces the quantity $A-BK_i$, which shows up frequently in control theory. For example, one important question is: will x_h remain bounded, or will it go to infinity as time goes on? To answer this, let's imagine for simplicity that these K_is are equal (call this matrix K). Then the expression above becomes $(A - BK)^h x_0$. Now consider the maximum eigenvalue $\lambda_{\max}$ of $A - BK$. If $|\lambda_{\max}| > 1$, then there's some nonzero initial state $\bar{x}_0$, the corresponding eigenvector, for which

$$\lim_{h\to\infty} (A - BK)^h \bar{x}_0 = \lim_{h\to\infty} \lambda_{\max}^h \bar{x}_0 = \infty.$$

Otherwise, if $|\lambda_{\max}| < 1$, then it's impossible for your original state to explode as dramatically.

3.5 Extensions

We've now formulated an optimal solution for the time-homogeneous LQR and computed the expected state under the optimal policy. However, real world tasks rarely have such simple dynamics, and we may wish to design more complex cost functions. In this section, we'll consider more general extensions of LQR where some of the assumptions we made above are relaxed. Specifically, we'll consider:

1. **Time-dependency**, where the dynamics and cost function might change depending on the timestep.
2. **General quadratic cost**, where we allow for linear terms and a constant term.
3. **Tracking a goal trajectory** rather than aiming for a single goal state-action pair.

Combining these will allow us to use the LQR solution to solve more complex setups by taking *Taylor approximations* of the dynamics and cost functions.

3.5.1 Time-dependent dynamics and cost function

So far, we've considered the *time-homogeneous* case, where the dynamics and cost function stay the same at every timestep. However, this might not always be the case. As an example, in many sports, the rules and scoring system might change during an overtime period. To address such tasks, we can loosen the time-homogeneous restriction, and consider the case where the dynamics and cost function are *time-dependent.* Our analysis remains almost identical; in fact, we can simply add a time index to the matrices A and B that determine the dynamics and the matrices Q and R that determine the cost.

The modified problem is now defined as follows:

Definition 3.10 (Time-dependent LQR)**.**

$$\begin{aligned} \min_{\pi_0,\dots,\pi_{H-1}} \quad & \mathbb{E}\left[\left(\sum_{h=0}^{H-1}(x_h^\top Q_h x_h) + u_h^\top R_h u_h\right) + x_H^\top Q_H x_H\right] \\ \text{where} \quad & x_{h+1} = f_h(x_h, u_h, w_h) = A_h x_h + B_h u_h + w_h \\ & x_0 \sim \mu_0 \\ & u_h = \pi_h(x_h) \\ & w_h \sim \mathcal{N}(0, \sigma^2 I). \end{aligned}$$

The derivation of the optimal value functions and the optimal policy remains almost exactly the same, and we can modify the Riccati equation accordingly:

Definition 3.11 (Time-dependent Riccati Equation)**.**

$$P_h = Q_h + A_h^\top P_{h+1} A_h - A_h^\top P_{h+1} B_h (R_h + B_h^\top P_{h+1} B_h)^{-1} B_h^\top P_{h+1} A_h.$$

Note that this is just the time-homogeneous Riccati equation (def. 3.8), but with the time index added to each of the relevant matrices.

Exercise 3.3 (Time dependent LQR proof)**.** Walk through the proof in sec. 3.4 to verify that we can simply add h for the time-dependent case.

Additionally, by allowing the dynamics to vary across time, we gain the ability to *locally approximate* nonlinear dynamics at each timestep. We'll discuss this later in the chapter.

3.5.2 More general quadratic cost functions

Our original cost function had only second-order terms with respect to the state and action, incentivizing staying as close as possible to $(x^\star, u^\star) = (0, 0)$. We can also consider more general quadratic cost functions that also have first-order terms and a constant term. Combining this with time-dependent dynamics results in the following expression, where we introduce a new matrix M_h for the cross term, linear coefficients q_h and r_h for the state and action respectively, and a constant term c_h:

$$c_h(x_h, u_h) = (x_h^\top Q_h x_h + x_h^\top M_h u_h + u_h^\top R_h u_h) + (x_h^\top q_h + u_h^\top r_h) + c_h. \tag{3.26}$$

Similarly, we can also include a constant term $v_h \in \mathbb{R}^{n_x}$ in the dynamics. This is *deterministic*, unlike the stochastic noise w_h:

$$x_{h+1} = f_h(x_h, u_h, w_h) = A_h x_h + B_h u_h + v_h + w_h.$$

Exercise 3.4 (General cost function)**.** Derive the optimal solution. You will need to slightly modify the proof in sec. 3.4.

3.5.3 Tracking a predefined trajectory

Consider applying LQR to a task like autonomous driving, where the target state-action pair changes over time. We might want the vehicle to follow a predefined *trajectory* of states and actions $(x_h^\star, u_h^\star)_{h=0}^{H-1}$. To express this as a control problem, we'll need a corresponding time-dependent cost function:

$$c_h(x_h, u_h) = (x_h - x_h^\star)^\top Q(x_h - x_h^\star) + (u_h - u_h^\star)^\top R(u_h - u_h^\star).$$

Note that this punishes states and actions that are far from the intended trajectory. By expanding out these multiplications, we can see that this is actually a special case of the more general quadratic cost function above eq. 3.26:

$$M_h = 0, \qquad q_h = -2Qx_h^\star, \qquad r_h = -2Ru_h^\star, \qquad c_h = (x_h^\star)^\top Q(x_h^\star) + (u_h^\star)^\top R(u_h^\star).$$

3.6 Approximating nonlinear dynamics

The LQR algorithm solves for the optimal policy when the dynamics are *linear* and the cost function is an *upward-curved quadratic*. However, real settings are rarely this simple! Let's return to the CartPole example from the start of the chapter (ex. 3.1). The dynamics (physics) aren't linear. How can we approximate this by an LQR problem?

Concretely, let's consider a *noise-free* problem since, as we saw, the noise doesn't factor into the optimal policy. Let's assume the dynamics and cost function are stationary, and ignore the terminal state for simplicity:

Definition 3.12 (Nonlinear control problem)**.**

$$\begin{aligned} \min_{\pi_0,\dots,\pi_{H-1}:\mathcal{S}\to\mathcal{A}} \quad & \mathop{\mathbb{E}}_{x_0\sim P_0}\left[\sum_{h=0}^{H-1} c(x_h,u_h)\right] \\ \text{where} \quad & x_{h+1} = f(x_h,u_h) \\ & u_h = \pi_h(x_h) \\ & x_0 \sim \mu_0 \\ & c(x,u) = d(x,x^\star) + d(u,u^\star). \end{aligned}$$

Here, d denotes a function that measures the "distance" between its two arguments.

This is now only slightly simplified from the general optimal control problem (see def. 3.2). Here, we don't know an analytical form for the dynamics f or the cost function c, but we assume that we're able to *query/sample/simulate* them to get their values at a given state and action. To clarify, consider the case where the dynamics are given by real world physics. We can't (yet) write down an expression for the dynamics that we can differentiate or integrate analytically. However, we can still *simulate* the dynamics and cost function by running a real-world experiment and measuring the resulting states and costs. How can we adapt LQR to this more general nonlinear case?

3.6.1 Local linearization

How can we apply LQR when the dynamics are nonlinear or the cost function is more complex? We'll exploit the useful fact that we can take a function that's *locally continuous* around $(x^\star, u^\star)$ and approximate it nearby with low-order polynomials (i.e. its Taylor approximation). In particular, as long as the dynamics f are differentiable around $(x^\star, u^\star)$ and the cost function c is twice differentiable at $(x^\star, u^\star)$, we can take a linear approximation of f and a quadratic approximation of c to bring us back to the regime of LQR.

Linearizing the dynamics around $(x^\star, u^\star)$ gives:

$$f(x,u) \approx f(x^\star,u^\star) + \nabla_x f(x^\star,u^\star)(x-x^\star) + \nabla_u f(x^\star,u^\star)(u-u^\star)$$
$$(\nabla_x f(x,u))_{ij} = \frac{df_i(x,u)}{dx_j}, \quad i,j \le n_x \qquad (\nabla_u f(x,u))_{ij} = \frac{df_i(x,u)}{du_j}, \quad i \le n_x, j \le n_u$$

and quadratizing the cost function around $(x^\star, u^\star)$ gives:

$$\begin{aligned} c(x,u) \approx\ & c(x^\star,u^\star) && \text{constant term} \\ & + \nabla_x c(x^\star,u^\star)(x-x^\star) + \nabla_u c(x^\star,u^\star)(a-u^\star) && \text{linear terms} \\ & \left.\begin{aligned} & + \frac{1}{2}(x-x^\star)^\top \nabla_{xx} c(x^\star,u^\star)(x-x^\star) \\ & + \frac{1}{2}(u-u^\star)^\top \nabla_{uu} c(x^\star,u^\star)(u-u^\star) \\ & + (x-x^\star)^\top \nabla_{xu} c(x^\star,u^\star)(u-u^\star) \end{aligned}\right\} && \text{quadratic terms} \end{aligned}$$

where the gradients and Hessians are defined as

$$(\nabla_x c(x,u))_i = \frac{dc(x,u)}{dx_i}, \quad i \le n_x \qquad (\nabla_u c(x,u))_i = \frac{dc(x,u)}{du_i}, \quad i \le n_u$$
$$(\nabla_{xx} c(x,u))_{ij} = \frac{d^2c(x,u)}{dx_i dx_j}, \quad i,j \le n_x \qquad (\nabla_{uu} c(x,u))_{ij} = \frac{d^2c(x,u)}{du_i du_j}, \quad i,j \le n_u$$
$$(\nabla_{xu} c(x,u))_{ij} = \frac{d^2c(x,u)}{dx_i du_j}. \quad i \le n_x, j \le n_u$$

Exercise 3.5 (Expressing as quadratic)**.** Note that this cost can be expressed in the general quadratic form seen in eq. 3.26. Derive the corresponding quantities Q, R, M, q, r, c.

3.6.2 Finite differencing

To calculate these gradients and Hessians in practice, we use a method known as **finite differencing** for numerically computing derivatives. Namely, we can simply use the limit definition of the derivative, and see how the function changes as we add or subtract a tiny δ to the input.

$$\frac{d}{dx} f(x) = \lim_{\delta \to 0} \frac{f(x+\delta) - f(x)}{\delta}$$

This only requires us to be able to *query* the function, not to have an analytical expression for it, which is why it's so useful in practice.

3.6.3 Local convexification

However, simply taking the second-order approximation of the cost function is insufficient, since for the LQR setup we required that the Q and R matrices were positive definite, i.e. that all of their eigenvalues were positive.

One way to naively *force* some symmetric matrix D to be positive definite is to set any non-positive eigenvalues to some small positive value $\varepsilon > 0$. Recall that any real symmetric matrix $D \in \mathbb{R}^{n \times n}$ has an basis of eigenvectors $u_1, \dots, u_n$ with corresponding eigenvalues $\lambda_1, \dots, \lambda_n$ such that $Du_i = \lambda_i u_i$. Then we can construct the positive definite approximation by

$$\widetilde{D} = \left(\sum_{i=1,\dots,n | \lambda_i > 0} \lambda_i u_i u_i^\top \right) + \varepsilon I.$$

Exercise: Convince yourself that $\widetilde{D}$ is indeed positive definite.

Note that Hessian matrices are generally symmetric, so we can apply this process to Q and R to obtain the positive definite approximations $\widetilde{Q}$ and $\widetilde{R}$. Now that we have an upward-curved quadratic approximation to the cost function, and a linear approximation to the state transitions, we can simply apply the time-homogenous LQR methods from sec. 3.4.

But what happens when we enter states far away from $x^\star$ or want to use actions far from $u^\star$? A Taylor approximation is only accurate in a *local* region around the point of linearization, so the performance of our LQR controller will degrade as we move further away. We'll see how to address this in the next section using the **iterative LQR** algorithm.

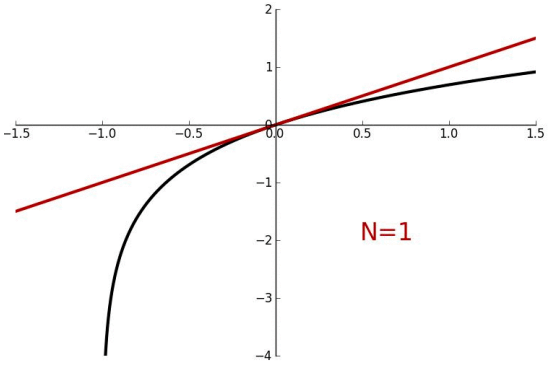


Figure 3.6: Local linearization might only be accurate in a small region around the point of linearization.

3.6.4 Iterative LQR

To address these issues with local linearization, we'll use an iterative approach, where we repeatedly linearize around different points to create a *time-dependent* approximation of the dynamics, and then solve the resulting time-dependent LQR problem to obtain a better policy. This is known as **iterative LQR** or **iLQR**:

Definition 3.13 (Iterative LQR)**.** For each iteration of the algorithm:

1. Form a time-dependent LQR problem (sec. 3.5.1) around the current candidate trajectory using local linearization.
2. Compute the optimal policy of the time-dependent LQR problem.
3. Sample a trajectory using this optimal policy.
4. Compute a better candidate trajectory by interpolating between the current and proposed actions.

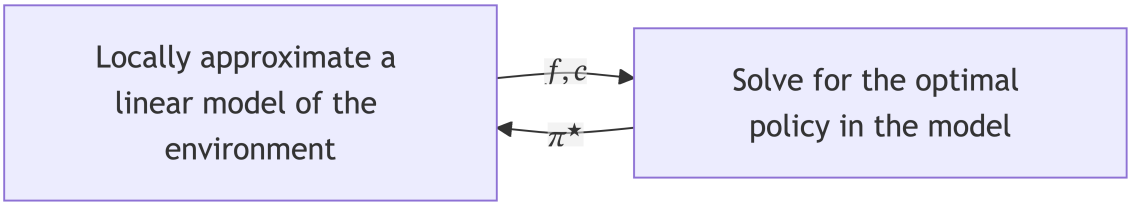


Figure 3.7: iLQR can be phrased in the style of policy iteration (sec. 2.4.4.2). It alternates between computing a locally linear model around the current policy's trajectories and computing the optimal policy that solves the linear model.

Now let's go through the details of each step. We'll use superscripts to denote the iteration of the algorithm. We'll also denote $\bar{x}_0 = \mathbb{E}_{x_0 \sim \mu_0}[x_0]$ as the expected initial state.

At iteration i of the algorithm, we begin with a **candidate** trajectory $\bar{\tau}^i = (\bar{x}_0^i, \bar{u}_0^i, \dots, \bar{x}_{H-1}^i, \bar{u}_{H-1}^i)$.

Step 1: Form a time-dependent LQR problem. At each timestep $h \in [H]$, we use the techniques from sec. 3.6 to linearize the dynamics and quadratize the cost function around $(\bar{x}_h^i, \bar{u}_h^i)$:

$$
\begin{aligned}
f_h(x,u) &\approx f(\bar{x}_h^i, \bar{u}_h^i) + \nabla_x f(\bar{x}_h^i, \bar{u}_h^i)(x - \bar{x}_h^i) + \nabla_u f(\bar{x}_h^i, \bar{u}_h^i)(u - \bar{u}_h^i) \\
c_h(x,u) &\approx c(\bar{x}_h^i, \bar{u}_h^i) + \begin{bmatrix} x - \bar{x}_h^i & u - \bar{u}_h^i \end{bmatrix} \begin{bmatrix} \nabla_x c(\bar{x}_h^i, \bar{u}_h^i) \\ \nabla_u c(\bar{x}_h^i, \bar{u}_h^i) \end{bmatrix} \\
&\quad + \frac{1}{2} \begin{bmatrix} x - \bar{x}_h^i & u - \bar{u}_h^i \end{bmatrix} \begin{bmatrix} \nabla_{xx} c(\bar{x}_h^i, \bar{u}_h^i) & \nabla_{xu} c(\bar{x}_h^i, \bar{u}_h^i) \\ \nabla_{ux} c(\bar{x}_h^i, \bar{u}_h^i) & \nabla_{uu} c(\bar{x}_h^i, \bar{u}_h^i) \end{bmatrix} \begin{bmatrix} x - \bar{x}_h^i \\ u - \bar{u}_h^i \end{bmatrix}.
\end{aligned}
$$

Step 2: Compute the optimal policy. We solve the time-dependent LQR problem using the time-dependent algebraic Riccati equation (def. 3.11) to compute the optimal policy $\pi_0^i, \dots, \pi_{H-1}^i$.

Step 3: Generate a new series of actions. We generate a new sample trajectory by taking actions according to this optimal policy:

$$\bar{x}_0^{i+1} = \bar{x}_0, \qquad \tilde{u}_h = \pi_h^i(\bar{x}_h^{i+1}), \qquad \bar{x}_{h+1}^{i+1} = f(\bar{x}_h^{i+1}, \tilde{u}_h).$$

Note that the states are sampled according to the *true* dynamics, which we assume we have query access to.

Step 4: Compute a better candidate trajectory. Note that we've denoted these actions as $\tilde{u}_h$ and aren't directly using them for the next iteration $\bar{u}_h^{i+1}$. Rather, we want to *interpolate* between them and the actions from the previous iteration $\bar{u}_0^i, \dots, \bar{u}_{H-1}^i$. This is so that the cost will *increase monotonically,* since if the new policy turns out to actually be worse, we can stay closer to the previous trajectory. The new policy might be worse, for example, if the states in the new trajectory are far enough from the original states that the local linearization is no longer accurate.

Formally, we want to find $\alpha \in [0, 1]$ to generate the next iteration of actions $\bar{u}_0^{i+1}, \dots, \bar{u}_{H-1}^{i+1}$ such that the cost is minimized:

$$\begin{aligned} \min_{\alpha \in [0,1]} \quad & \sum_{h=0}^{H-1} c(x_h, \bar{u}_h^{i+1}) \\ \text{where} \quad & x_{h+1} = f(x_h, \bar{u}_h^{i+1}) \\ & \bar{u}_h^{i+1} = \alpha \bar{u}_h^i + (1-\alpha)\tilde{u}_h \\ & x_0 = \bar{x}_0. \end{aligned}$$

Note that this optimizes over the closed interval $[0, 1]$, so by the Extreme Value Theorem, it's guaranteed to have a global maximum.

The final output of this algorithm is a policy $\pi^{n_{\text{steps}}}$ derived after n_{steps} of the algorithm. Though the proof is somewhat complex, one can show that for many nonlinear control problems, this solution converges to a locally optimal solution (in the policy space).

3.7 Key takeaways

This chapter introduced some approaches to solving simple variants of the optimal control problem def. 3.2. We began with the simple case of linear dynamics and an upward-curved quadratic cost. This model is called the linear quadratic regulator (LQR). The optimal policy

can be solved using dynamic programming. We then extended these results to the more general nonlinear case via local linearization. We finally studied the iterative LQR algorithm for solving nonlinear control problems.

In the big picture, the LQR algorithm can be seen as an extension of the dynamic programming algorithms in Chapter 2 to continuous state and action spaces. Both of these methods assume that the environment is fully known. iLQR loosens this assumption slightly: as long as we have "query access" to the environment, that is, we can efficiently sample the state following any given state-action pair, we can locally approximate the dynamics using finite differences.

3.8 Bibliographic notes and further reading

The field of control theory generally predates that of RL and can be seen as a precursor to the model-based planning algorithms used for RL tasks. People have attempted to solve the optimal control problem since antiquity: the first recorded control device (i.e. our notion of a policy, put into practice) is thought to be a water clock from the third century BCE that kept time by controlling the amount of water flowing out of a vessel (Keviczky et al., 2019; Mayr, 1970).

Watt's 1788 centrifugal governor for controlling the pace of a steam engine was one of the greatest inventions in control at the time. We refer the reader to (Bellman, 1961; MacFarlane, 1979) for more on the history of control systems.

The first attempt to formalize control theory as a field is attributed to Maxwell (1867). Maxwell (well known for Maxwell's equations) demonstrated the utility of analyzing mathematical models to design stable systems. Other researchers also studied the stability of differential equations. The work of Lyapunov (1892) has been widely influential in this field; his seminal work did not appear in English until Ljapunov & Fuller (1992).

A number of excellent textbooks have been published on optimal control (Athans & Falb, 1966; Bellman, 1957; Berkovitz, 1974; Lewis et al., 2012). We refer interested readers to these for further reading, particularly about the continuous-time case.

4 Multi-Armed Bandits

4.1 Introduction

The **multi-armed bandit** (MAB) problem is a simple setting for studying the basic challenges of sequential decision-making. In this setting, an agent repeatedly chooses from a fixed set of actions, called **arms**, each of which has an associated reward distribution. The agent's goal is to maximize the total reward it receives over some time period. Here are a few examples:

Example 4.1 (Online advertising)**.** Let's suppose you, the agent, are an advertising company. You have K different ads that you can show to users. Let's suppose you are targeting a single user. You receive 1 reward if the user clicks the ad, and 0 otherwise. Thus, the unknown *reward distribution* associated to each ad is a Bernoulli distribution defined by the probability that the user clicks on the ad. Your goal is to maximize the total number of clicks by the user.

(a) Deciding which ad to put on a billboard can be treated as a MAB problem. Image from Negative Space (2015).

The MAB is the simplest setting for exploring the **exploration-exploitation tradeoff**: should the agent choose new actions to learn more about the environment, or should it choose actions that it already knows to be good?

In this chapter, we will introduce the multi-armed bandits setting, and discuss some of the challenges that arise when trying to solve problems in this setting. We will also introduce some of the key concepts that we will use throughout the book, such as regret and exploration-exploitation tradeoffs.

Remark 4.1 (Etymology). The name "multi-armed bandit" comes from slot machines in casinos, which are often called "one-armed bandits" since they have one arm (the lever) and rob money from the player.

Example 4.2 (Clinical trials)**.** Suppose you're a pharmaceutical company, and you're testing a new drug. You have K different dosages of the drug that you can administer to patients. You receive 1 reward if the patient recovers and 0 otherwise. Thus, the unknown reward distribution associated with each dosage is a Bernoulli distribution defined by the probability that the patient recovers. Your goal is to maximize the total number of patients that recover.

(a) Optimizing between different medications can be treated as a MAB problem. Image from Pixabay (2016a).

Table 4.1: How bandits leads into the full RL problem.

Environment is...	known (planning/optimal control)	unknown (learning from experience)
stateless/invariant ($\|\mathcal{S}\| = 1$)	(trivial)	multi-armed bandits
stateful/dynamic	MDP planning (dynamic programming)	"full" RL

4.2 The multi-armed bandit problem

Let's frame the MAB problem mathematically. In both ex. 4.1 and ex. 4.2, the outcome of each decision is binary (i.e. 0 or 1). This is the **Bernoulli bandit** problem, which we'll use throughout the chapter.

Definition 4.1 (Bernoulli bandit problem)**.** Let K denote the number of arms. We'll label them $0, \ldots, K-1$ and use *superscripts* to indicate the arm index k. Since we seldom need to raise a number to a power, this won't cause much confusion.
Let μ^k be the mean of arm k, such that pulling arm k either returns reward 1 with probability μ^k or 0 otherwise.
The agent gets T chances to pull an arm. The task is to determine a strategy for maximizing the total reward.

(a) Each arm k in a Bernoulli bandit is a biased coin with probability μ^k of showing heads. Each "pull" corresponds to flipping a coin. The goal is to maximize the number of heads given T flips.

doesn't make sense.) As such, Each arm is an action: $|\mathcal{A}| = K$. the expected value of pulling arm k, which we denote μ^k in this chapter, is exactly the Q-function (def. 2.9):

$$\mu^k = Q(k) \tag{4.1}$$

Note that we omit the arguments for π, h, and s, which are meaningless in the MAB setting.

In pseudocode, the agent's interaction with the MAB environment can be described by the following process:

```
function mab_loop(mab : MAB, agent : "Agent")
  for t ∈ range(T) do
    a_t ← agent.choose_arm()
    r ← pull(a_t)
    agent.update_history(a_t, r)
  end for
end function
```

Figure 4.4: The MAB interaction loop. $a_t \in [K]$ denotes the tth arm pulled.

What's the *optimal* strategy for the agent, i.e. the one that achieves the highest expected reward? Convince yourself that the agent should try to always pull the arm with the highest expected reward:

Definition 4.2 (Optimal arm)**.** We call the arm whose reward distribution has the highest mean the **optimal arm**:

$$\begin{aligned} k^\star &:= \arg\max_{k \in [K]} \mu^k \\ \mu^\star &:= \mu^{k^\star} = \max_{k \in [K]} \mu^k. \end{aligned} \tag{4.2}$$

4.2.1 Regret

When analyzing an MAB algorithm, rather than measuring its performance in terms of the (expected) total reward, we instead compare it to the "oracle" strategy that knows the optimal arm in advance. This gives rise to a measure of performance known as **regret**, which answers, how much better *could* you have done had you known the optimal arm from the beginning? This provides a more meaningful baseline than the expected total reward, whose magnitude may vary widely depending on the bandit problem.

Example 4.3 (Regret provides a more meaningful baseline)**.** Consider the problem of grade inflation or deflation. If you went to a different school or university, you might have gotten a very different grade, independently of your actual ability. This is like comparing bandit algorithms based on the expected total reward: the baseline varies depending on the problem (i.e. school), making it hard to compare the algorithms (i.e. students) themselves. Instead, it makes more sense to measure you relative to the (theoretical) best student from your school. This is analogous to measuring algorithm performance using regret.

Definition 4.3 (Regret)**.** The agent's (cumulative) **regret** after T pulls is defined as

$$\text{Regret}_T := \sum_{t=0}^{T-1} \mu^\star - \mu^{a_t}, \tag{4.3}$$

where $a_t \in [K]$ is the arm chosen on the tth pull.

This depends on the *true means* of the pulled arms, and is independent of the observed rewards. The regret Regret_T is a random variable where the randomness comes from the agent's strategy (i.e. the sequence of actions $a_0, \dots, a_{K-1}$).

Also note that we care about the *total* regret across all decisions, rather than just the final decision. If we only cared about the quality of the final decision, the best strategy would be to learn as much as possible about the environment, and then use that knowledge to make the best possible decision at the last step. This is a **pure exploration** problem since the agent is never penalized for taking suboptimal actions up to the final decision. Minimizing the *cumulative* regret (or maximizing the *cumulative* reward) means we must strike a balance between *exploration* and *exploitation* throughout the entire decision-making process.

Throughout the chapter, we will try to upper bound the regret of various algorithms in two different senses:

1. Upper bound the *expected regret*, i.e. show that for some upper bound M_T,
$$\mathbb{E}[\text{Regret}_T] \le M_T. \tag{4.4}$$
2. Find a *high-probability* upper bound on the regret, i.e. show that for some small *failure probability* $\delta > 0$,
$$\mathbb{P}(\text{Regret}_T \le M_{T,\delta}) \ge 1 - \delta \tag{4.5}$$
for some upper bound $M_{T,\delta}$.

The first approach says that the regret is at most M_T in expectation. However, the agent might still achieve a higher or lower regret on a particular randomization. The second approach says that, with probability at least $1 - \delta$, the agent will achieve regret at most $M_{T,\delta}$. However, it doesn't say anything about the regret in the remaining δ fraction of runs, which could be arbitrarily high.

Exercise 4.1 (Expected regret bounds yield probabilistic regret bounds)**.** Suppose we have an upper bound M_T on the expected regret. Show that, with probability $1-\delta$, $M_{T,\delta} := M_T/\delta$ upper bounds the regret. Note this is a much higher bound! For example, if $\delta = 0.01$, the high-probability bound is 100 times as large.

Remark 4.3 (Issues with regret)*.* Is regret the right way to measure performance? Consider the following two algorithms:

1. An algorithm which sometimes chooses very bad arms.
2. An algorithm which often chooses slightly suboptimal arms.

We might care a lot about this distinction in practice, for example, in healthcare, where an occasional very bad decision could have extreme consequences. However, these two algorithms could achieve the exact same expected total regret.

Other performance measures include **probably approximately correct** (PAC) bounds and **uniform** high-probability regret bounds. There is also a **Uniform-PAC** bound. We won't discuss these in detail here; see the bibliographic notes for more details.

We'd like to achieve **sublinear regret** in expectation:

$$\mathbb{E}[\text{Regret}_T] = o(T). \tag{4.6}$$

(See sec. A.1 if you're unfamiliar with big-oh notation.) An algorithm with sublinear regret will eventually do as well as the oracle strategy as $T \to \infty$.

Remark 4.4 (Interpreting sublinear regret)*.* An algorithm with sublinear regret in expectation doesn't necessarily always choose the correct arm. In fact, such an algorithm can still choose a suboptimal arm infinitely many times. Consider a two-armed Bernoulli bandit with $\mu^0 = 0$ and $\mu^1 = 1$. An algorithm that chooses arm 0 $\sqrt{T}$ times out of T still achieves $O(\sqrt{T})$ regret in expectation.

Also, sublinear regret measures the *asymptotic* performance of the algorithm as $T \to \infty$. It doesn't tell us about the *rate* at which the algorithm approaches optimality.

In the remaining sections, we'll walk through a series of MAB algorithms and see how they make different tradeoffs between exploration and exploitation.

4.3 Pure exploration

A trivial strategy is to always choose arms at random (i.e. "pure exploration").

Definition 4.4 (Pure exploration)**.** On the tth pull, the arm a_t is sampled uniformly at random from all arms:

$$a_t \sim \text{Unif}([K]) \tag{4.7}$$

Theorem 4.1 (Pure exploration expected regret)**.** *The pure exploration strategy achieves regret $\Theta(T)$ in expectation.*

Proof. By choosing arms uniformly at random,

$$\mathop{\mathbb{E}}_{a_t \sim \text{Unif}([K])}[\mu^{a_t}] = \bar{\mu} := \frac{1}{K}\sum_{k=0}^{K-1} \mu^k \tag{4.8}$$

so the expected regret is simply

$$\begin{aligned} \mathbb{E}[\text{Regret}_T] &= \sum_{t=0}^{T-1} \mathbb{E}[\mu^\star - \mu^{a_t}] \\ &= T \cdot (\mu^\star - \bar{\mu}). \end{aligned} \tag{4.9}$$

This scales as $\Theta(T)$ in expectation, i.e. *linear* in the number of pulls T. □

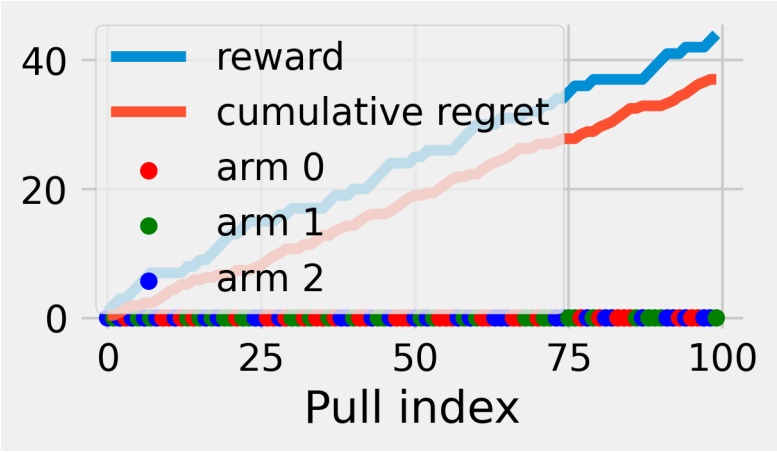


Figure 4.5: Reward and cumulative regret of the pure exploration algorithm.

Pure exploration doesn't use any information about the environment to improve its strategy. The distribution over its arm choices always appears "(uniformly) random".

4.4 Pure greedy

How might we improve on pure exploration? Instead, we could try each arm once, and then commit to the one with the highest observed reward. We'll call this the **pure greedy** strategy.

Definition 4.5 (Pure greedy)**.** On the tth pull, the arm a_t is chosen according to

$$a_t := \begin{cases} t & t < K \\ \arg\max_{k \in [K]} r_k & t \geq K \end{cases} \tag{4.10}$$

where r_t denotes the reward obtained on the tth pull.

How does the expected regret of this strategy compare to that of pure exploration? We'll do a more general analysis in the following section. Now, for intuition, suppose there's just $K = 2$ arms with means $\mu^1 > \mu^0$. If $r_1 > r_0$, then we repeatedly pull arm 1 and achieve zero regret per pull. If $r_0 > r_1$, then we repeatedly pull arm 0 and incur $\mu^1 - \mu^0$ regret per pull. Accounting for the first two pulls and the case of a tie, we obtain the following lower bound:

$$\mathbb{E}[\text{Regret}_T] \geq \mathbb{P}(r^0 > r^1) T \cdot (\mu^1 - \mu^0) \tag{4.11}$$

This is still $\Theta(T)$, the same as pure exploration!

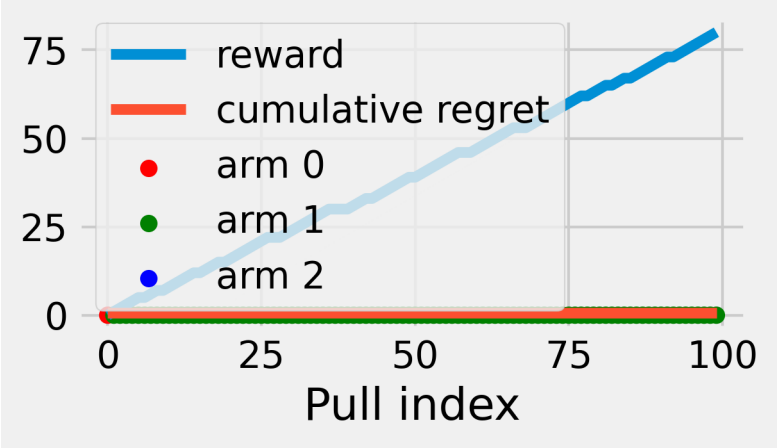


Figure 4.6: Reward and cumulative regret of the pure greedy algorithm.

The cumulative regret is a straight line because the regret only depends on the arms chosen and not the actual reward observed. In fact, we see in fig. 4.6 that the greedy algorithm can get lucky on the first set of pulls and act entirely optimally for that experiment! But its *average* regret is still linear, since the chance of sticking with a suboptimal arm is nonzero.

4.5 Explore-then-commit

We can improve the pure greedy algorithm (def. 4.5) as follows: let's reduce the variance of the reward estimates by pulling each arm $N_{\text{explore}} \geq 1$ times before committing. This is called the **explore-then-commit** strategy.

Definition 4.6 (Explore-then-commit)**.** The explore-then-commit algorithm first pulls each arm N_{explore} times. Let

$$\hat{\mu}^k := \frac{1}{N_{\text{explore}}} \sum_{n=0}^{N_{\text{explore}}-1} r_{kN_{\text{explore}}+n} \tag{4.12}$$

denote the empirical mean reward of arm k during these pulls. Afterwards, on the tth pull (for $t \geq N_{\text{explore}}K$), it chooses

$$a_t := \arg\max_{k \in [K]} \hat{\mu}^k. \tag{4.13}$$

Note that the "pure greedy" strategy above is just the special case where $N_{\text{explore}} = 1$.

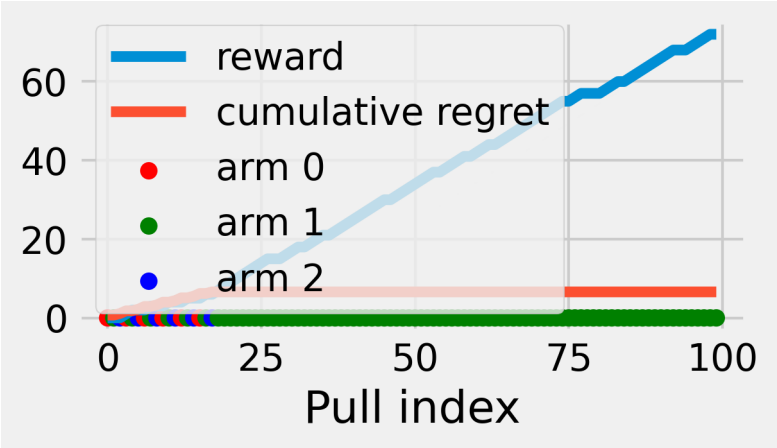


Figure 4.7: Reward and cumulative regret of the explore-then-commit algorithm

This algorithm finds the true optimal arm more frequently than the pure greedy strategy. We would expect ETC to then have a lower expected regret. We will show that ETC achieves the following regret bound:

Theorem 4.2 (ETC high-probability regret bound)**.** *The regret of ETC satisfies, with high probability,*

$$\mathit{Regret}_T = \widetilde{O}(T^{2/3}K^{1/3}), \tag{4.14}$$

where the tilde means we ignore logarithmic factors.

Proof. Let's analyze the expected regret of the explore-then-commit strategy by splitting it up into the exploration and exploitation phases.

Exploration phase. This phase takes $N_{\text{explore}}K$ pulls. Since at each step we incur at most 1 regret, the total regret is at most $N_{\text{explore}}K$.

Exploitation phase. This will take a bit more effort. We'll prove that for any total time T, we can choose N_{explore} such that with arbitrarily high probability, the regret is sublinear.

Let $\hat{k}$ denote the arm chosen after the exploration phase. We know the regret from the exploitation phase is

$$T_{\text{exploit}}(\mu^\star - \mu^{\hat{k}}) \qquad \text{where} \qquad T_{\text{exploit}} \coloneqq T - N_{\text{explore}}K. \tag{4.15}$$

So we'd need $\mu^\star - \mu^{\hat{k}} \to 0$ as $T \to \infty$ in order to achieve sublinear regret. How can we do this?

Let's define $\Delta^k \coloneqq \hat{\mu}^k - \mu^k$ to denote how far the mean estimate for arm k is from the true mean. How can we bound this quantity? We'll use the following useful inequality for i.i.d. bounded random variables:

Theorem 4.3 (Hoeffding's inequality)**.** *Let $X_0, \dots, X_{N-1}$ be i.i.d. random variables with $X_n \in [0,1]$ almost surely for each $n \in [N]$. Then for any $\delta > 0$,*

$$\mathbb{P}\left(\left|\frac{1}{N}\sum_{n=1}^{N}(X_n - \mathbb{E}[X_n])\right| > \sqrt{\frac{\ln(2/\delta)}{2N}}\right) \leq \delta. \tag{4.16}$$

Proof. The proof of this inequality is beyond the scope of this book. See Vershynin (2018, ch. 2.2). □

We can apply this directly to the rewards for a given arm k, since the rewards from that arm are i.i.d.:

$$\mathbb{P}\left(|\Delta^k| > \sqrt{\frac{\ln(2/\delta)}{2N_{\text{explore}}}}\right) \leq \delta. \tag{4.17}$$

This can be thought of as a $1-\delta$ confidence interval for the true arm mean μ^k. But we can't apply this to arm $\hat{k}$ directly since $\hat{k}$ is itself a random variable. Instead, we need to bound the error across all the arms simultaneously, so that the resulting bound will apply *no matter what* $\hat{k}$ "crystallizes" to. The **union bound** provides a simple way to do this. (See Theorem A.1 for a review.)

In our case, we have one event per arm, stating that the CI for that arm contains the true mean. The probability that *all* of the CIs are accurate is then

$$\mathbb{P}\left(\forall k \in [K] : |\Delta^k| \leq \sqrt{\frac{\ln(2/\delta)}{2N_{\text{explore}}}}\right) \geq 1 - K\delta. \tag{4.18}$$

This lower-bounds the probability that the CI for arm $\hat{k}$ is accurate. Let us return to our original task of bounding $\mu^\star - \mu^{\hat{k}}$. We apply the useful trick of "adding zero":

$$\begin{aligned}
\mu^{k^\star} - \mu^{\hat{k}} &= \mu^{k^\star} - \mu^{\hat{k}} + (\hat{\mu}^{k^\star} - \hat{\mu}^{k^\star}) + (\hat{\mu}^{\hat{k}} - \hat{\mu}^{\hat{k}}) \\
&= \Delta^{\hat{k}} - \Delta^{k^\star} + \underbrace{(\hat{\mu}^{k^\star} - \hat{\mu}^{\hat{k}})}_{\leq 0 \text{ by definition of } \hat{k}} \\
&\leq 2\sqrt{\frac{\ln(2K/\delta')}{2N_{\text{explore}}}} \text{ with probability at least } 1-\delta'
\end{aligned} \tag{4.19}$$

where we've set $\delta' := K\delta$. Putting this all together, we've shown that, with probability $1-\delta'$,

$$\text{Regret}_T \leq N_{\text{explore}}K + T_{\text{exploit}} \cdot \sqrt{\frac{2\ln(2K/\delta')}{N_{\text{explore}}}}. \tag{4.20}$$

Note that it suffices for N_{explore} to be on the order of $\sqrt{T}$ to achieve sublinear regret. In particular, we can find the optimal N_{explore} by setting the derivative of eq. 4.20 with respect to N_{explore} to zero:

$$
\begin{aligned}
0 &= K - T_{\text{exploit}} \cdot \frac{1}{2}\sqrt{\frac{2\ln(2K/\delta')}{N_{\text{explore}}^3}} \\
N_{\text{explore}} &= \left(T_{\text{exploit}} \cdot \frac{\sqrt{\ln(2K/\delta')/2}}{K} \right)^{2/3}
\end{aligned}
\tag{4.21}
$$

Plugging this into the expression for the regret, we have (still with probability $1 - \delta'$):

$$
\begin{aligned}
\text{Regret}_T &\leq 3T^{2/3} \sqrt[3]{K \ln(2K/\delta')/2} \\
&= \widetilde{O}(T^{2/3} K^{1/3}),
\end{aligned}
\tag{4.22}
$$

satisfying Theorem 4.2. □

The ETC algorithm is rather "abrupt" in that it switches from exploration to exploitation after a fixed number of pulls. What if the total number of pulls T isn't known in advance? We'll need to use a more gradual transition, which brings us to the *epsilon-greedy* algorithm.

4.6 Epsilon-greedy

Instead of doing all of the exploration and then all of the exploitation in separate phases, which requires knowing the time horizon beforehand, we can instead *interleave* exploration and exploitation by, at each pull, choosing a random action with some probability. We call this the **epsilon-greedy** algorithm.

Definition 4.7 (Epsilon-greedy)**.** Let N_t^k denote the number of times that arm k is pulled within the first t overall pulls, and let $\hat{\mu}_t^k$ denote the sample mean of the corresponding rewards:

$$
\begin{aligned}
N_t^k &:= \sum_{t'=0}^{t-1} \mathbf{1}\{a_{t'} = k\}, \\
\hat{\mu}_t^k &:= \frac{1}{N_t^k} \sum_{t'=0}^{t-1} \mathbf{1}\{a_{t'} = k\} r_{t'}.
\end{aligned}
\tag{4.23}
$$

On the tth pull, the arm a_t is chosen according to

$$
a_t := \begin{cases} \text{sample from Unif}([K]) & \text{with probability } \epsilon_t \\ \arg\max_{k \in [K]} \hat{\mu}_t^k & \text{with probability } 1 - \epsilon_t, \end{cases}
\tag{4.24}
$$

where $\epsilon_t \in [0, 1]$ is the probability of choosing a random action.

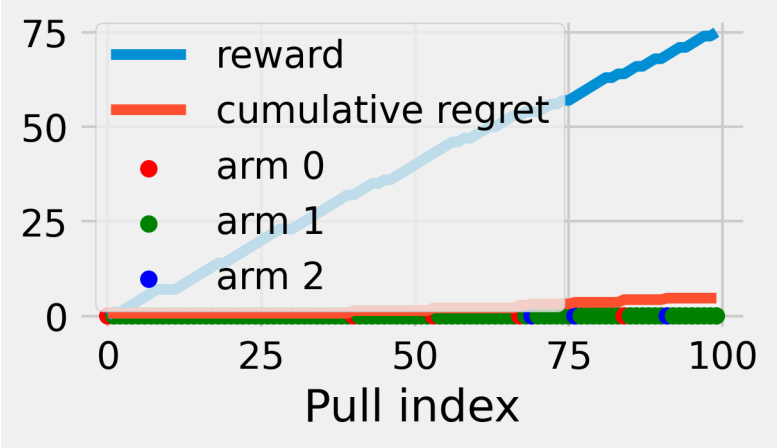


Figure 4.8: Reward and cumulative regret of the epsilon-greedy algorithm

We let the exploration probability ϵ_t vary over time. This lets us gradually *decrease* ϵ_t as we learn more about the reward distributions and no longer need to explore as much.

Exercise 4.2 (Regret for constant epsilon)**.** What is the asymptotic expected regret of the algorithm if we set ϵ to be a constant? Is this different from pure exploration (def. 4.4)?

Theorem 4.4 (Epsilon-greedy expected regret)**.** *Let* $\epsilon_t := \sqrt[3]{K \ln(t)/t}$*. Then the epsilon-greedy achieves a regret of* $\widetilde{O}(t^{2/3}K^{1/3})$ *in expectation (ignoring logarithmic factors).*

Proof. We won't prove this here. See A. Agarwal et al. (2022) for a proof. □

In ETC, we had to set N_{explore} based on the total number of pulls T. But the epsilon-greedy algorithm handles the exploration incrementally: the regret rate holds for *any* t, and doesn't depend on the total number of pulls T.

But the way epsilon-greedy and ETC explore is rather naive: they explore *uniformly* across all the arms. But what if we could be smarter about it, and explore *more* for arms that we're less certain about?

4.7 Upper Confidence Bound (UCB)

The upper confidence bound algorithm is the first strategy we study that explores *adaptively*: it identifies arms it is less certain about, and explicitly trades off between learning more about these and exploiting arms that already seem good.

To quantify how *certain* we are about the mean of each arm, UCB computes statistical *confidence intervals* (CIs) for the arm means, and then chooses the arm with the highest *upper confidence bound* (hence its name). Concretely, after t pulls, we compute upper confidence bounds M_t^k for each arm k such that $\mu^k \leq M_t^k$ with high probability, and select $a_t := \arg\max_{k \in [K]} M_t^k$. This operates on the principle of **the benefit of the doubt (i.e. optimism in the face of uncertainty)**: we'll choose the arm that we're most *optimistic* about.

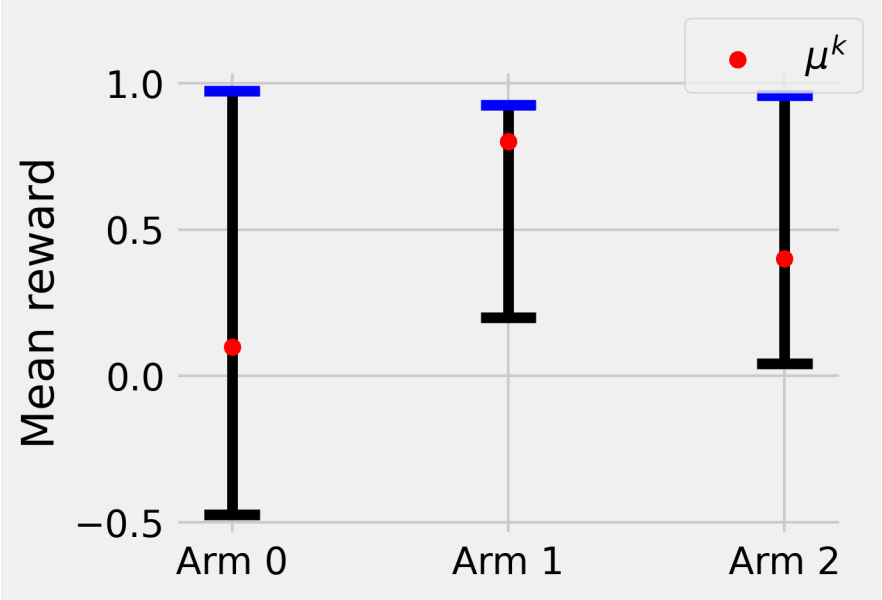


Figure 4.9: Visualization of confidence intervals for the arm means. UCB would choose arm 0 since it has the highest upper confidence bound.

Theorem 4.5 (Confidence intervals for arm means)**.** *As in our definition of the epsilon-greedy algorithm (def. 4.7), let N_t^k denote the number of times we pull arm k within the first t pulls, and let $\hat{\mu}_t^k$ denote the sample mean of the corresponding rewards. Then with probability at least $1-\delta$, for all $t \geq K$,*

$$\mu^k \in \left[\hat{\mu}_t^k - \frac{1}{\sqrt{N_t^k}} \cdot \sqrt{\frac{\ln(2t/\delta)}{2}}, \hat{\mu}_t^k + \frac{1}{\sqrt{N_t^k}} \cdot \sqrt{\frac{\ln(2t/\delta)}{2}} \right]. \tag{4.25}$$

We assume that during the first K pulls, we pull each arm once.

Definition 4.8 (Upper confidence bound algorithm (Auer, 2002))**.** The UCB algorithm first pulls each arm once. Then it chooses the tth pull (for $t \geq K$) by taking

$$a_t := \arg\max_{k \in [K]} \hat{\mu}_t^k + \sqrt{\frac{\ln(2t/\delta)}{2N_t^k}}, \tag{4.26}$$

where $\delta \in (0, 1)$ is a parameter that controls the width of the confidence interval.

Intuitively, UCB prioritizes arms where:

1. $\hat{\mu}_t^k$ is large, i.e. the arm's corresponding sample mean is high, and we'd choose it for *exploitation*, and
2. $\sqrt{\frac{\ln(2t/\delta)}{2N_t^k}}$ is large, i.e. N_t^k is small and we're still uncertain about the arm, and we'd choose it for *exploration*.

δ is formally the coverage probability of the confidence interval, but we can also treat it as a parameter that trades off between exploration and exploitation:

- A smaller δ would give us a larger interval, emphasizing the exploration term.
- A larger δ would give a tighter interval, prioritizing the current sample means.

We now prove Theorem 4.5.

Proof. Our proof is similar to that in Theorem 4.2: we use Hoeffding's inequality to construct a CI for the mean of a bounded random variable (i.e. the rewards from a given arm). However, we must be careful, since the number of samples from arm k up to time t, N_t^k, is a random variable.

Hoeffding's inequality (Theorem 4.3) tells us that, for N i.i.d. samples $\tilde{r}_0, \dots, \tilde{r}_{N-1}$ from arm k,

$$|\tilde{\mu}_N^k - \mu^k| \leq \sqrt{\frac{\ln(2/\delta)}{2N}} \tag{4.27}$$

with probability at least $1 - \delta$, where $\tilde{\mu}_N^k = \frac{1}{N} \sum_{n=0}^{N-1} \tilde{r}_n$. The union bound then tells us that with probability at least $1 - \delta'$, for all $N = 1, \dots, t$,

$$\forall N = 1, \dots, t : |\tilde{\mu}_N^k - \mu^k| \leq \sqrt{\frac{\ln(2t/\delta')}{2N}}. \tag{4.28}$$

Since $N_t^k \in \{1, \dots, t\}$ (we assume each arm is pulled once at the start), eq. 4.28 implies that

$$|\tilde{\mu}_{N_t^k}^k - \mu^k| \leq \sqrt{\frac{\ln(2t/\delta')}{N}} \tag{4.29}$$

with probability at least $1-\delta'$ as well. But notice that $\tilde{\mu}^k_{N^k_t}$ is identically distributed to $\hat{\mu}^k_t$: both refer to the sample mean of the first N^k_t pulls from arm k. This gives the bound in Theorem 4.5 (after relabelling $\delta' \mapsto \delta$). □

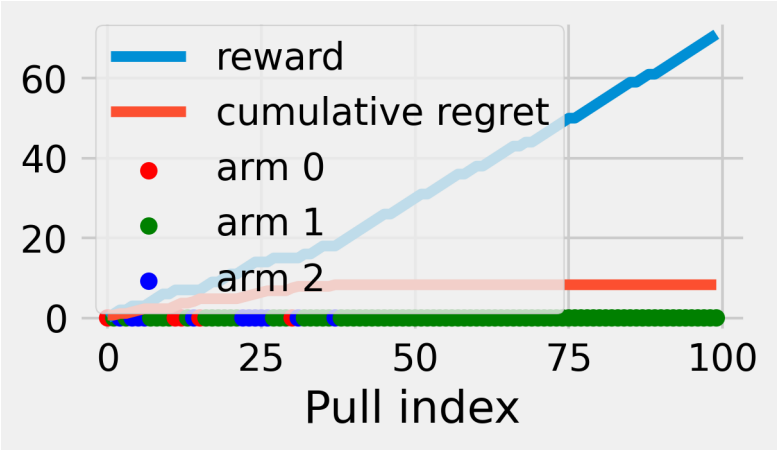


Figure 4.10: Reward and cumulative regret of the upper confidence bound algorithm

As desired, this explores in a smarter, *adaptive* way compared to the previous algorithms. Does it achieve lower regret?

Theorem 4.6 (UCB high-probability regret bound)**.** *With probability* $1-\delta''$*,*

$$Regret_T = \widetilde{O}(K\sqrt{T}) \tag{4.30}$$

Proof. First we'll bound the regret incurred at each timestep. Then we'll bound the *total* regret across timesteps.

For the sake of analysis, we'll use a slightly looser bound that applies across the whole time horizon and across all arms. We'll omit the derivation since it's very similar to the above (walk through it yourself for practice).

$$\begin{aligned} &\mathbb{P}\left(\forall k \le K, t < T : |\hat{\mu}^k_t - \mu^k| \le B^k_t\right) \ge 1-\delta'' \\ &\qquad \text{where} \quad B^k_t := \sqrt{\frac{\ln(2TK/\delta'')}{2N^k_t}}. \end{aligned} \tag{4.31}$$

Intuitively, B^k_t denotes the *width* of the CI for arm k at time t. Then, assuming the above uniform bound holds (which occurs with probability $1-\delta''$), we can bound the regret at each timestep as follows:

$$\begin{aligned}
\mu^\star - \mu^{a_t} &\le \hat{\mu}_t^{k^\star} + B_t^{k^\star} - \mu^{a_t} && \text{applying UCB to arm } k^\star \\
&\le \hat{\mu}_t^{a_t} + B_t^{a_t} - \mu^{a_t} && \text{since UCB chooses } a_t = \arg\max_{k\in[K]} \hat{\mu}_t^k + B_t^k \\
&\le 2B_t^{a_t} && \text{since } \hat{\mu}_t^{a_t} - \mu^{a_t} \le B_t^{a_t} \text{ by definition of } B_t^{a_t}
\end{aligned} \tag{4.32}$$

Summing this across timesteps gives

$$\begin{aligned}
\text{Regret}_T &\le \sum_{t=0}^{T-1} 2B_t^{a_t} \\
&= \sqrt{2\ln(2TK/\delta'')} \sum_{t=0}^{T-1} (N_t^{a_t})^{-1/2} \\
\sum_{t=0}^{T-1} (N_t^{a_t})^{-1/2} &= \sum_{t=0}^{T-1} \sum_{k=0}^{K-1} \mathbf{1}\{a_t = k\} (N_t^k)^{-1/2} \\
&= \sum_{k=0}^{K-1} \sum_{n=1}^{N_T^k} n^{-1/2} \\
&\le K \sum_{n=1}^{T} n^{-1/2} \\
\sum_{n=1}^{T} n^{-1/2} &\le 1 + \int_1^T x^{-1/2}\, \mathrm{d}x \\
&= 1 + (2\sqrt{x})_1^T \\
&= 2\sqrt{T} - 1 \\
&\le 2\sqrt{T}
\end{aligned}$$

Putting everything together gives, with probability $1 - \delta''$,

$$\begin{aligned}
\text{Regret}_T &\le 2K\sqrt{2T\ln(2TK/\delta'')} \\
&= \widetilde{O}(K\sqrt{T}),
\end{aligned} \tag{4.33}$$

as in Theorem 4.6. □

In fact, we can do a more sophisticated analysis to trim off a factor of $\sqrt{K}$ and show $\text{Regret}_T = \widetilde{O}(\sqrt{TK})$.

4.7.1 Lower bound on regret (intuition)

Is it possible to do better than $\Omega(\sqrt{T})$ in general? In fact, no! We can show that any algorithm must incur $\Omega(\sqrt{T})$ regret in the worst case. We won't rigorously prove this here, but the intuition is as follows.

The Central Limit Theorem tells us that with T i.i.d. samples from some distribution, we can only learn the mean of the distribution to within $\Omega(1/\sqrt{T})$ (the standard deviation). Then, since we get T samples spread out across the arms, we can only learn each arm's mean to an even looser degree.

That is, if two arms have means that are within about $1/\sqrt{T}$, we won't be able to confidently tell them apart, and will sample them about equally. But then we'll incur regret

$$\Omega((T/2) \cdot (1/\sqrt{T})) = \Omega(\sqrt{T}). \tag{4.34}$$

4.8 Thompson sampling and Bayesian bandits

So far, we've treated the parameters $\mu^0, \dots, \mu^{K-1}$ of the reward distributions as *fixed*. Instead, we can take a **Bayesian** approach where we treat them as random variables from some **prior distribution**. Then, upon pulling an arm and observing a reward, we can simply *condition* on this observation to exactly describe the **posterior distribution** over the parameters. This fully describes the information we gain about the parameters from observing the reward.

From this Bayesian perspective, the **Thompson sampling** algorithm follows naturally: just sample from the distribution of the optimal arm, given the observations!

In other words, we sample each arm proportionally to how likely we think it is to be optimal, given the observations so far. This strikes a good exploration-exploitation tradeoff: we explore more for arms that we're less certain about, and exploit more for arms that we're more certain about. Thompson sampling is a simple yet powerful algorithm that achieves state-of-the-art performance in many settings.

Example 4.4 (Bayesian Bernoulli bandit)**.** We've been working in the Bernoulli bandit setting, where arm k yields a reward of 1 with probability μ^k and no reward otherwise. The vector of success probabilities $\mu = (\mu^0, \dots, \mu^{K-1})$ thus describes the entire MAB.

Under the Bayesian perspective, we think of μ as a *random* vector drawn from some prior distribution $p \in \triangle([0,1]^K)$. For example, we might have p be the Uniform distribution over the unit hypercube $[0,1]^K$, that is,

$$p(\mu) = \begin{cases} 1 & \text{if } \mu \in [0,1]^K \\ 0 & \text{otherwise} \end{cases} \tag{4.35}$$

In this case, upon viewing some reward, we can exactly calculate the **posterior** distribution of μ using Bayes's rule (i.e. the definition of conditional probability):

$$\begin{aligned}\mathbb{P}(\mu \mid a_0, r_0) &\propto \mathbb{P}(r_0 \mid a_0, \mu)\mathbb{P}(a_0 \mid \mu)\mathbb{P}(\mu) \\ &\propto (\mu^{a_0})^{r_0}(1-\mu^{a_0})^{1-r_0}.\end{aligned}$$

This is the PDF of the Beta$(1+r_0, 1+(1-r_0))$ distribution, which is a conjugate prior for the Bernoulli distribution. That is, if we start with a Beta prior on μ^k (note that Unif$([0,1])$ = Beta$(1,1)$), then the posterior, after conditioning on samples from Bern(μ^k), will also be Beta. This is a very convenient property, since it means we can simply update the parameters of the Beta distribution upon observing a reward, rather than having to recompute the entire posterior distribution from scratch.

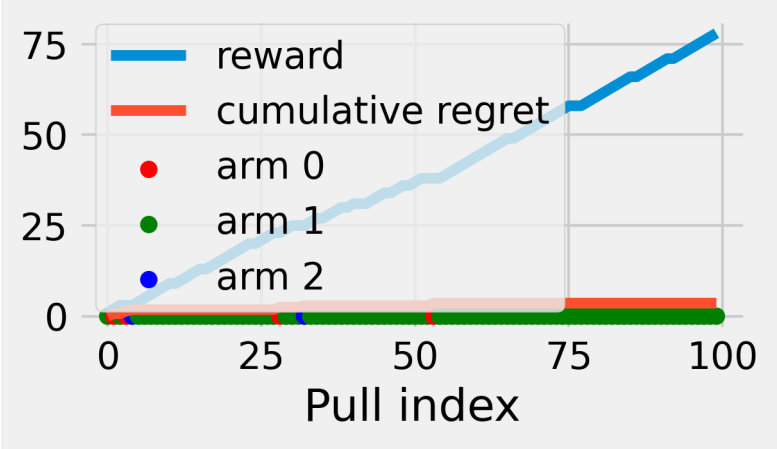


Figure 4.11: Reward and cumulative regret of the Thompson sampling algorithm

It turns out that asymptotically, Thompson sampling is optimal in the following sense. Lai & Robbins (1985) prove an *instance-dependent* lower bound that says for *any* bandit algorithm,

$$\liminf_{T\to\infty} \frac{\mathbb{E}[N_T^k]}{\ln(T)} \geq \frac{1}{\mathrm{KL}\left(\mu^k \parallel \mu^\star\right)} \tag{4.36}$$

where

$$\mathrm{KL}\left(\mu^k \parallel \mu^\star\right) := \mu^k \ln \frac{\mu^k}{\mu^\star} + (1-\mu^k) \ln \frac{1-\mu^k}{1-\mu^\star} \tag{4.37}$$

measures the **Kullback-Leibler divergence** from the Bernoulli distribution with mean μ^k to the Bernoulli distribution with mean $\mu^\star$. It turns out that Thompson sampling achieves

this lower bound with equality! That is, not only is the error *rate* optimal, but the *constant factor* is optimal as well.

4.9 Contextual bandits

In the multi-armed bandits setting described above, each pull is exactly the same. We don't obtain any prior information about the reward distributions. However, consider the online advertising example (ex. 4.1), where each arm corresponds to an ad we could show a given user, and we receive a reward if and only if the user clicks on the ad. Suppose Alice is interested in sports and politics, while Bob is interested in news and entertainment. The ads that Alice is likely to click differ from those that Bob is likely to click. That is, the reward distributions of the arms depend on some prior *context*. We can model such environments using **contextual bandits**.

Definition 4.9 (Contextual bandit)**.** At each pull t, a *context* x_t is chosen from the set $\mathcal{X}$ of all possible contexts. The learner gets to observe the context *before* they choose an action. That is, we treat the action chosen on the tth pull as a function of the context: $a_t(x_t)$. Then, the learner observes the reward from the chosen arm, where the reward distribution also depends on the context.

Example 4.5 (Online advertising as a contextual bandit problem)**.** Suppose you represent an online advertising company and you are tasked with placing ads on the New York Times website. Articles are categorized into six sections, each with its own web page: U.S., World, Business, Arts, Lifestyle, and Opinion. You might decide to frame this as a contextual bandits problem with $|\mathcal{X}| = 6$. This is because the demographic of readers likely differs across the different pages, and so the reward distributions differ across pages accordingly.

Suppose our context is *discrete*, as in ex. 4.5. Then we could treat each context as its own MAB, assigning each context-arm pair to a distinct arm in an enlarged MAB of $K|\mathcal{X}|$ arms.

Exercise 4.3 (Naive UCB for contextual bandits)**.** Write down the UCB algorithm for this enlarged MAB. That is, write an expression M_t^k in terms of K and $|\mathcal{X}|$ such that $a_t(x_t) = \arg\max_{k\in[K]} M_t^k$.

Recall that executing UCB for T timesteps on an MAB with K arms achieves a regret bound of $\widetilde{O}(\sqrt{TK})$ (Theorem 4.6). So in this problem, we would achieve regret $\widetilde{O}(\sqrt{TK|\mathcal{X}|})$ in the contextual MAB, which has a polynomial dependence on $|\mathcal{X}|$. But in a situation where we have too many possible contexts, or the context is a continuous value, this algorithm no longer works.

Note that this "enlarged MAB" treats the different contexts as entirely unrelated to each other, while in practice, often contexts are *related* to each other in some way. In the medical trial

example (ex. 4.2), clients with similar health situations may benefit from the same medications. How can we incorporate this structure into our solution?

4.9.1 Linear contextual bandits

Remark 4.5 (Supervised learning). The material in this section depends on basic knowledge of supervised learning (Chapter 5).

Suppose now that we observe a vector-valued context, that is, $\mathcal{X} = \mathbb{R}^D$. For example, in a medical trial (ex. 4.2), we might know the patient's height, age, blood pressure, etc., each of which would be a separate element of the vector. The approach of treating each context as its own MAB problem no longer works out-of-the-box since the number of contexts is infinite. Instead, we assume that the mean reward of each arm k is a function of the context: $\mu^k : \mathcal{X} \to \mathbb{R}$.

Rather than the Bernoulli bandit problem (def. 4.1), we'll assume that the mean reward is *linear* in the context. (This model is easier to work with for our purposes.) That is, a reward r^k from arm k is drawn according to

$$r^k = x^\top \theta^k + \varepsilon, \tag{4.38}$$

where $\theta^k \in \mathbb{R}^D$ describes a *feature direction* for arm k and ε is some independent noise with mean 0 and variance σ^2. This clearly has

$$\mu^k(x) = x^\top \theta^k. \tag{4.39}$$

Then, on the tth pull, given context x_t, we can model the mean reward from arm k as

$$\hat{\mu}_t^k(x_t) := x_t^\top \hat{\theta}_t^k, \tag{4.40}$$

where $\hat{\theta}_t^k$ is some estimator of θ^k computed from the first t pulls.

Remark 4.6 (Assumptions for linear models). Why did we switch from Bernoulli reward distributions to the additive model in eq. 4.38? The estimator eq. 4.40 is a bit odd for the Bernoulli bandit problem: since μ^k is the success probability of arm k, it would have to lie between 0 and 1, but $\hat{\mu}_t^k(x_t)$ extends outside this interval. The question of how to model the unknown parameters of a distribution is a **supervised learning** problem (Chapter 5). For instance, in Bernoulli bandit, we could instead use a *logistic regression* model for each arm's success probability. We won't dive into more advanced supervised learning topics here.

Our goal is still to develop an algorithm that chooses an arm $a_t(x_t)$ to pull based on the experience gained during the first t trials. We will use *supervised learning*, in particular, empirical risk minimization (sec. 5.3), to estimate θ^k for each arm. That is, we compute $\hat{\theta}_t^k$ as the vector that minimizes squared error from the observed rewards:

$$\hat{\theta}_t^k := \arg\min_{\theta \in \mathbb{R}^D} \sum_{t' \in \mathcal{J}_t^k} (r_{t'} - x_{t'}^\top \theta)^2, \tag{4.41}$$

where

$$\mathcal{J}_t^k := \{t' \in [t] \mid a_{t'} = k\} \tag{4.42}$$

denotes the subset of $[t]$ at which arm k was pulled. Note that $|\mathcal{J}_t^k| = N_t^k$ as defined in eq. 4.23. The ERM problem in eq. 4.41 has the closed-form solution known as the **ordinary least squares** (OLS) estimator:

$$\begin{aligned} \hat{\theta}_t^k &= (\widehat{\Sigma}_t^k)^{-1} \left(\frac{1}{N_t^k} \sum_{t' \in \mathcal{J}_t^k} x_{t'} r_{t'} \right) \\ \text{where} \quad \widehat{\Sigma}_t^k &= \frac{1}{N_t^k} \sum_{t' \in \mathcal{J}_t^k} x_{t'} x_{t'}^\top. \end{aligned} \tag{4.43}$$

$\widehat{\Sigma}_t^k$ is the (biased) empirical covariance matrix of the contexts across the trials where arm k was pulled (out of the first t trials).

Remark 4.7 (Invertibility). In eq. 4.43, we write the expression $(\widehat{\Sigma}_t^k)^{-1}$, assuming that $\widehat{\Sigma}_t^k$ is invertible. This only holds if the context features are linearly independent, that is, there does not exist some coordinate $d \in [D]$ that can be written as a linear combination of the other coordinates. One way to ensure that $\widehat{\Sigma}_t^k$ is invertible is to add a λI regularization term to it. This is equivalent to solving a *ridge regression* problem instead of the unregularized least squares problem.

Now, given a sequence of t contexts, pulls, and rewards $x_0, a_0, r_0, \ldots, x_{t-1}, a_{t-1}, r_{t-1}$, we can estimate $\mu^k(x_t)$ for each arm. This is helpful, but we haven't yet solved the problem of *deciding* which arms to pull in each context to trade off between exploration and exploitation.

The upper confidence bound (UCB) algorithm (def. 4.8) was a useful strategy. What would we need to adapt UCB to this new setting? Recall that at each step, UCB estimates a *confidence interval* for each arm mean, and chooses the arm with the highest upper confidence bound. We already have an estimator $\hat{\mu}_t^k(x_t)$ of each arm mean, so all that is left is to compute the width of the confidence interval itself.

Theorem 4.7 (LinUCB confidence interval)**.** *Suppose we pull each arm once during the first K trials. Then at each pull $t \geq K$, for all arms $k \in [K]$ and all contexts $x \in \mathcal{X}$, we have that with probability at least $1 - 1/\beta^2$,*

$$\mu^k(x) \in \left[x^\top \hat{\theta}_t^k - \beta \frac{\sigma}{\sqrt{N_t^k}} \cdot \sqrt{x^\top (\widehat{\Sigma}_t^k)^{-1} x}, x^\top \hat{\theta}_t^k + \beta \frac{\sigma}{\sqrt{N_t^k}} \cdot \sqrt{x^\top (\widehat{\Sigma}_t^k)^{-1} x} \right], \tag{4.44}$$

where

- *σ is the standard deviation of the reward (eq. 4.38),*
- *N_t^k is the number of times arm k was pulled in the first t pulls (eq. 4.23),*
- *$\hat{\theta}_t^k$ is the OLS estimator for arm k (eq. 4.43),*
- *and $\widehat{\Sigma}_t^k$ is the sample covariance matrix of the contexts given arm k (eq. 4.43).*

We can then plug this interval into the UCB strategy (def. 4.8) to obtain what is known as the LinUCB algorithm:

Definition 4.10 (LinUCB (L. Li et al., 2010))**.** LinUCB first pulls each arm once. Then, for pull $t \geq K$, LinUCB chooses an arm according to

$$a_t(x_t) := \arg\max_{k \in [K]} x^\top \hat{\theta}_t^k + \beta \frac{\sigma}{\sqrt{N_t^k}} \sqrt{x_t^\top (\widehat{\Sigma}_t^k)^{-1} x_t}, \tag{4.45}$$

where $\sigma, N_t^k, \hat{\theta}_t^k, \widehat{\Sigma}_t^k$ are defined as in Theorem 4.7, and $\beta > 0$ controls the width of the confidence interval (and thereby the exploration-exploitation tradeoff).

The first term is exactly our predicted mean reward $\hat{\mu}_t^k(x_t)$. We can think of it as the term encouraging *exploitation* of an arm if its predicted mean reward is high. The second term is the width of the confidence interval given by Theorem 4.7. This is the *exploration* term that encourages an arm when x_t is *not aligned* with the data seen so far, or if arm k has not been explored much and so N_t^k is small.

Now we prove Theorem 4.7.

Proof. How can we construct the upper confidence bound in Theorem 4.7? Previously, we treated the pulls of an arm as i.i.d. samples and used Hoeffding's inequality to bound the distance of the sample mean, our estimator, from the true mean. However, now our estimator is not a sample mean, but rather the OLS estimator above eq. 4.43. We will construct the confidence interval using **Chebyshev's inequality**, which gives a confidence interval for the mean of a distribution based on that distribution's *variance*.

Theorem 4.8 (Chebyshev's inequality). *For a random variable Y with mean μ and variance σ^2,*

$$|Y - \mu| \leq \beta\sigma \quad \textit{with probability} \geq 1 - \frac{1}{\beta^2} \tag{4.46}$$

We want to construct a CI for $x^\top \theta^k$, the mean reward from arm k. Our estimator is $x^\top \hat{\theta}_t^k$. Applying Chebyshev's inequality requires us to compute the variance of our estimator.

Theorem 4.9 (Variance of OLS). *Given the linear reward model in eq. 4.38,*

$$Var(x^\top \hat{\theta}_t^k) = \frac{\sigma^2}{N_t^k} x^\top (\widehat{\Sigma}_k^t)^{-1} x, \tag{4.47}$$

where $\widehat{\Sigma}_k^t$ is defined in eq. 4.43.

Proof. We leave the proof as an exercise. It follows from applications of the law of iterated expectations. □

Now we can substitute eq. 4.47 into eq. 4.46 to obtain the desired CI:

$$x_t^\top \theta^k \leq x_t^\top \hat{\theta}_t^k + \beta \sqrt{x_t^\top (A_t^k)^{-1} x_t} \quad \text{with probability} \geq 1 - \frac{1}{\beta^2} \tag{4.48}$$

□

4.10 Key takeaways

In this chapter, we explored the **multi-armed bandit** setting for analyzing sequential decision-making in an unknown environment. An MAB consists of multiple arms, each with an unknown reward distribution. The agent's task is to learn about these through interaction, eventually minimizing the *regret*, which measures how suboptimal the chosen arms were.

We saw algorithms such as **upper confidence bound** and **Thompson sampling** that handle the tradeoff between *exploration* and *exploitation*, that is, the tradeoff between choosing arms that the agent is *uncertain* about and arms the agent already supposes are be good.

We finally discussed **contextual bandits**, in which the agent gets to observe some *context* that affects the reward distributions. We can approach these problems through **supervised learning** approaches.

4.11 Bibliographic notes and further reading

The problem that a bandit algorithm faces after t pulls is to infer differences between the means of a set of distributions given t samples spread out across those distributions. This is a well-studied problem in statistics.

A common practice in various fields such as psychology is to compare within-group variances to the variance of the group means. This popular class of methods is known as **analysis of variance** (ANOVA). It has been studied at least since Laplace in the 1770s (Stigler, 2003) and was further popularized by Fisher (1925). William Thompson studied the two-armed mean distinction problem in Thompson (1933) and Thompson (1935). Our focus has been on the decision-making aspect of the bandit problem, rather than the hypothesis testing aspect.

Adding the sequential decision-making aspect turns the above into the full multi-armed bandit problem. Wald (1949) studies the sequential analysis problem in depth. Robbins (1952) then builds on Wald's work and provides a clear exposition of the bandit problem in the form presented in this chapter. Berry & Fristedt (1985) is a comprehensive treatment of bandit problems from a statistical perspective.

Remark 4.8 (Disambiguation of LinUCB). The name "LinUCB" is used for various similar algorithms throughout the literature. Here we use it as defined in L. Li et al. (2010). This algorithm was also considered in Auer (2002), the paper that introduced UCB, under the name "Associative Reinforcement Learning with Linear Value Functions".

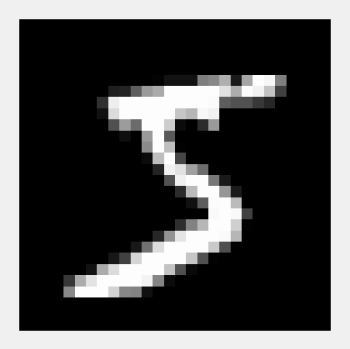
5 Supervised learning

5.1 Introduction

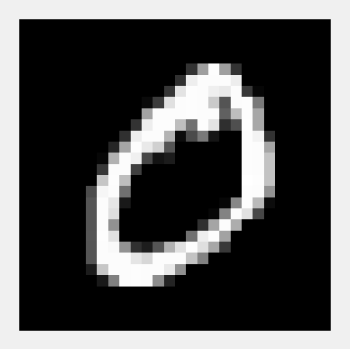
Supervised learning (SL) is a core subfield of machine learning alongside RL and unsupervised learning. The typical SL task is to approximate an *unknown function* given a dataset of input-output examples from that function.

Example 5.1 (Image classification)**.** One of the most common examples of an SL problem is the task of image classification: Given a dataset of images and their respective labels, construct a function that takes an image and outputs the correct label.

fig. 5.1 illustrates two samples (that is, input-output pairs) from the MNIST database of handwritten digits (Deng, 2012). This is a task that most humans can easily accomplish. By providing many samples of digits and their labels to a machine, SL algorithms can learn to solve this task as well.



(a) The handwritten digit 5.



(b) The handwritten digit 0.

Figure 5.1: The MNIST image classification dataset of 28*imes*28 handwritten digits.

When might function approximation be useful in RL? Recall that the definition of an MDP includes the state transitions $P : \mathcal{S} \times \mathcal{A} \to \triangle(\mathcal{S})$ and a reward function $r : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$ (def. 2.4). If we *know* both of these functions, in the sense that we are able to query them on arbitrary inputs, we can use powerful planning algorithms to solve for the optimal policy exactly (see sec. 2.3.2). Thus, if either (or both) of these is not known, we can use an SL algorithm to

model the environment and then solve the modeled environment using dynamic programming. This approach is called **fitted DP** and will be covered in Chapter 6.

We do not seek to comprehensively discuss supervised learning; see the bibiographic notes at the end of the chapter (sec. 5.6) for further resources. We hope to to leave you with an understanding of what types of problems SL can solve and how SL algorithms can be applied to RL.

5.2 The supervised learning task

In SL, we are given a dataset of labelled samples $(x_1, y_1), \dots, (x_N, y_N)$ that are independently sampled from some **data generating process**. Mathematically, we describe this data generating process as a joint distribution $p \in \triangle(\mathcal{X} \times \mathcal{Y})$, where $\mathcal{X}$ is the space of possible inputs and $\mathcal{Y}$ is the space of possible outputs. Note that, by the chain rule of probability, this can be factored as $p(x, y) = p(y \mid x)p(x)$.

Example 5.2 (Joint distributions for image classification)**.** For example, in ex. 5.1, the marginal distribution over x is assumed to be the distribution of handwritten digits by humans, scanned as 28×28 grayscale images. The conditional distribution $y \mid x$ is assumed to be the distribution over $\{0, \dots, 9\}$ that a human would assign to the image x.

Our task is to compute a "good" **prediction rule** $\hat{f} : \mathcal{X} \to \mathcal{Y}$ that takes an input and tries to predict the corresponding output.

5.2.1 Loss functions

How can we measure how "good" a prediction rule is? The most common way is to use a **loss function** $\ell : \mathcal{Y} \times \mathcal{Y} \to \mathbb{R}$ that compares the guess $\hat{y} := \hat{f}(x)$ with the true output y. $\ell(\hat{y}, y)$ should be low if the prediction rule accurately guessed the output, and high if the prediction was incorrect.

Example 5.3 (Zero-one loss)**.** In the image classification task ex. 5.1, we have $X = [0, 1]^{28 \times 28}$ (the space of 28-by-28 grayscale images) and $Y = \{0, \dots, 9\}$ (the image's label). We could use the zero-one loss function,

$$\ell(\hat{y}, y) = \begin{cases} 0 & \hat{y} = y \\ 1 & \hat{y} \neq y \end{cases} \tag{5.1}$$

to measure the accuracy of the prediction rule. That is, if the prediction rule assigns the wrong label to an image, it incurs a *loss* of one for that sample.

Example 5.4 (Square loss)**.** For a continuous output (i.e. $\mathcal{Y} \subseteq \mathbb{R}$), we typically use the **squared difference** as the loss function:

$$\ell(\hat{y}, y) = (\hat{y} - y)^2 \tag{5.2}$$

The square loss is nice to work with analytically since its derivative with respect to $\hat{y}$ is simply $2(\hat{y} - y)$. (Sometimes authors define the square loss as *half* of the above value to cancel the factor of 2 in the derivative. Generally speaking, scaling the loss by some constant scalar has no practical effect.)

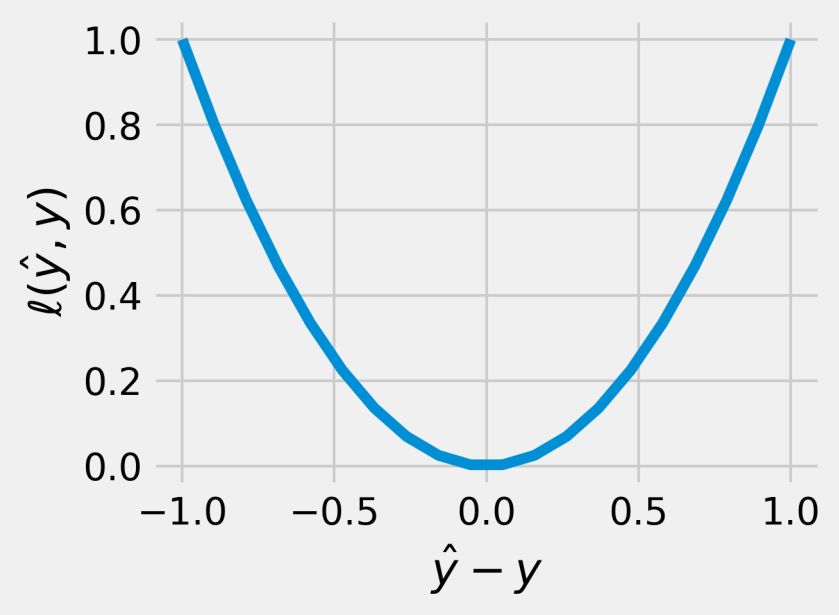


Figure 5.2: Square loss.

5.2.2 Model selection

Ultimately, we want a prediction rule that does well on new, unseen samples from the data generating process. We can thus ask, how much loss does the prediction rule incur *in expectation*? This is called the prediction rule's **generalization error** or **test error**.

Definition 5.1 (Generalization error)**.** Given a loss function ℓ and a prediction rule $\hat{f}$, the *generalization error* of the prediction rule is defined as the expected loss over the data generating process.

$$\text{error}_{\text{g}}(\hat{f}) := \mathop{\mathbb{E}}_{(x,y)\sim p}[\ell(\hat{f}(x), y)] \tag{5.3}$$

Suppose we sample a new input and output from the data generating process, make a guess according to our prediction rule, and use the loss function to compare our guess to the true output. If we repeat this many times, the average loss would approach the generalization error.

The goal of SL is then to find the prediction rule that minimizes the test error. For certain loss functions, the theoretical optimum can be analytically computed, such as for squared error.

Theorem 5.1 (The conditional expectation minimizes mean squared error)**.** *An important result is that, under the square loss, the optimal prediction rule is the* ***conditional expectation****:*

$$\arg\min_f \mathbb{E}[(y - f(x))^2] = (x \mapsto \mathbb{E}[y \mid x]) \tag{5.4}$$

Proof. We can decompose the mean squared error as

$$\begin{aligned} \mathbb{E}[(y - f(x))^2] &= \mathbb{E}[(y - \mathbb{E}[y \mid x] + \mathbb{E}[y \mid x] - f(x))^2] \\ &= \mathbb{E}[(y - \mathbb{E}[y \mid x])^2] + \mathbb{E}[(\mathbb{E}[y \mid x] - f(x))^2] \\ &\quad + 2\,\mathbb{E}[(y - \mathbb{E}[y \mid x])(\mathbb{E}[y \mid x] - f(x))] \end{aligned} \tag{5.5}$$

We leave it as an exercise to show that the last term is zero. (Hint: use the law of iterated expectations.) The first term is the **noise**, or irreducible error, that doesn't depend on f, and the second term is the error due to the approximation, which is minimized at 0 when $f(x) = \mathbb{E}[y \mid x]$. □

In most applications, such as in ex. 5.2, we can't integrate over the joint distribution of x, y, and so we can't evaluate $\mathbb{E}[y \mid x]$ analytically. Instead, all we get are N samples from the joint distribution of x and y. How might we use these to *approximate* the generalization error?

5.3 Empirical risk minimization

To estimate the generalization error, we could simply take the *sample mean* of the loss over the training data. This is called the **training loss** or **empirical risk**:

Definition 5.2 (Empirical risk)**.** Given a dataset $(x_1, y_1), \ldots, (x_N, y_N)$ sampled i.i.d. from the data generating process, and a loss function ℓ, the empirical risk of the prediction rule $\hat{f}$ is the average loss across the dataset:

$$\text{training loss}(\hat{f}) := \frac{1}{N} \sum_{n=1}^{N} \ell(\hat{f}(x_n), y_n). \tag{5.6}$$

By the law of large numbers, as N grows to infinity, the training loss converges to the generalization error (def. 5.1).

The **empirical risk minimization** (ERM) approach is to find a prediction rule that minimizes the empirical risk.

Definition 5.3 (Empirical risk minimization)**.** An ERM algorithm requires two ingredients to be chosen based on our **domain knowledge** about the DGP:

1. A **function class** $\mathcal{F}$, that is, the space of functions to consider.
2. A **fitting method** that uses the dataset to find the element of $\mathcal{F}$ that minimizes the training loss.

This allows us to compute the empirical risk minimizer:

$$\begin{aligned} \hat{f}_{\text{ERM}} &:= \arg\min_{f \in \mathcal{F}} \text{training loss}(f) \\ &= \arg\min_{f \in \mathcal{F}} \frac{1}{N} \sum_{n=1}^{N} \ell(f(x_n), y_n). \end{aligned} \tag{5.7}$$

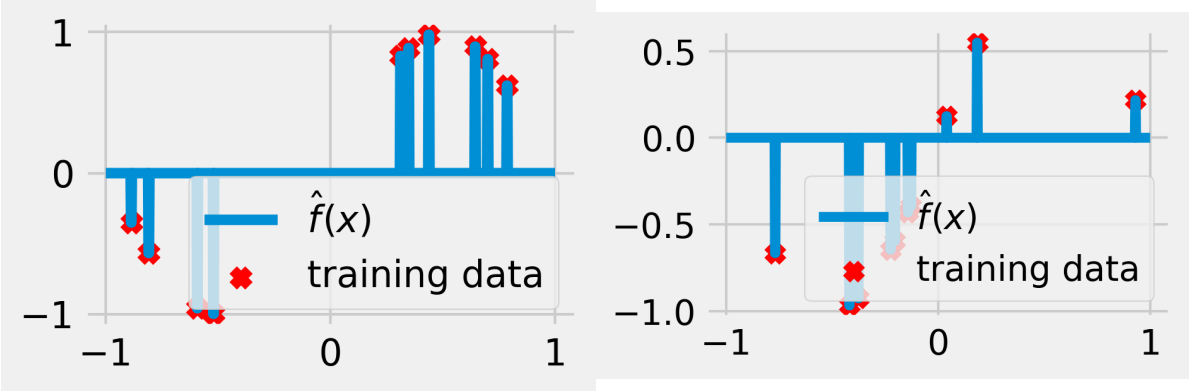
5.3.1 Function classes

How should we choose the correct function class? In fact, why do we need to constrain our search at all?

Exercise 5.1 (Overfitting)**.** Suppose we are trying to approximate a relationship between real-valued inputs and outputs using square loss as our loss function. Consider the prediction rule (visualized in fig. 5.3)

$$\hat{f}(x) = \sum_{n=1}^{N} y_n \mathbf{1}\{x = x_n\}. \tag{5.8}$$

What is the empirical risk of this function? How well does it perform on newly generated samples?

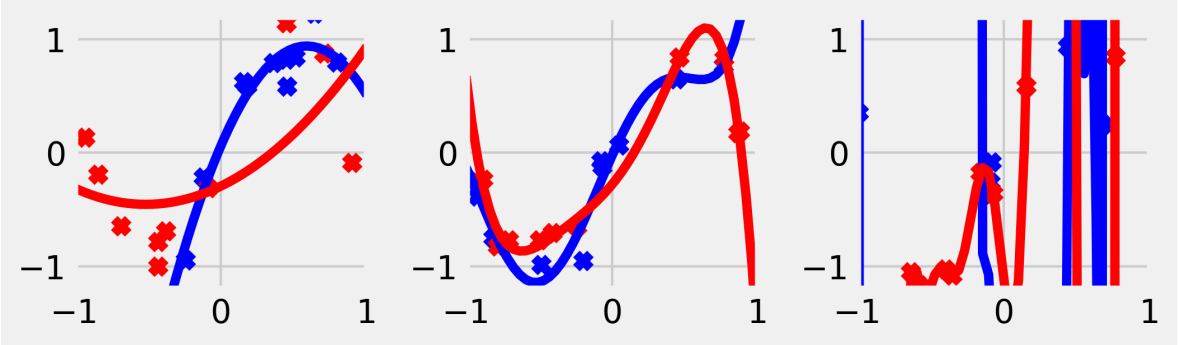


(a) One training dataset.

(b) Another training dataset.

Figure 5.3: A pathological prediction rule.

The choice of $\mathcal{F}$ depends on our **domain knowledge** about the task. On one hand, $\mathcal{F}$ should be large enough to contain the true relationship, but on the other, it shouldn't be *too* expressive; otherwise, it will **overfit** to random noise in the labels. The larger and more complex the function class, the more accurately we will be able to approximate any particular training dataset (i.e. smaller **bias**), but the more drastically the function will vary for *different* training datasets (i.e. larger **variance**). For most loss functions, including the square loss, it is possible to express the generalization error (def. 5.1) as a sum of a bias term and a variance term. The mathematical details of this so-called **bias-variance tradeoff** can be found, for example, in Hastie et al. (2013, Chapter 2.9).



(a) Degree 2 polynomials

(b) Degree 5 polynomials

(c) Degree 50 polynomials

Figure 5.4: Demonstrating the bias-variance tradeoff through polynomial regression. Increasing the degree increases the complexity of the polynomial function class.

We must also consider *practical* constraints on the function class. We need an efficient algorithm to actually compute the function in the class that minimizes the training error. This point should not be underestimated! The success of modern deep learning, for example, is

in large part due to hardware developments that have made certain operations more efficient than others.

5.3.2 Parameterized function classes

Both of the function classes we will consider, linear maps and neural networks, are **finite-dimensional**, a.k.a. **parameterized**. The notion of a parameterized function class is best illustrated by example:

Example 5.5 (Quadratic functions)**.** consider the class of **quadratic functions**, i.e. polynomials of degree 2. This is a three-dimensional function space ($D = 3$), since we can describe any quadratic p as

$$p(x) = ax^2 + bx + c, \tag{5.9}$$

where a, b, c are the three parameters. We could also use a different parameterization:

$$p(x) = a'(x - b')^2 + c'. \tag{5.10}$$

Definition 5.4 (Parameters)**.** Let $\mathcal{F}$ be a class of functions mapping from $\mathcal{X}$ to $\mathcal{Y}$. We say that $\mathcal{F}$ is a **parameterized** function class if each $f \in \mathcal{F}$ can be identified using D **parameters**.

Exercise 5.2 (Parameterization matters)**.** Note that the choice of parameterization can impact the performance of the chosen fitting method. What is the derivative of eq. 5.9 with respect to a, b, c? Compare this to the derivative of eq. 5.10 with respect to a', b', c'. This shows that gradient-based fitting methods may change their behaviour depending on the parameterization.

Using a parameterized function class allows us to reframe the ERM problem def. 5.3 in terms of optimizing over the *parameters* instead of over the functions they represent:

$$\begin{aligned} \hat{\theta}_{\text{ERM}} &:= \arg\min_{\theta \in \mathbb{R}^D} \text{training loss}(f_\theta) \\ &= \frac{1}{N} \sum_{n=1}^{N} (y_n - f_\theta(x_n))^2 \end{aligned} \tag{5.11}$$

In general, optimizing over a *finite-dimensional* space is much, much easier than optimizing over an *infinite-dimensional* space.

5.3.3 Gradient descent

One widely applicable fitting method for parameterized function classes is **gradient descent**.

Let $L(\theta) = \text{training loss}(f_\theta)$ denote the empirical risk in terms of the parameters. The **gradient descent** algorithm iteratively updates the parameters according to the rule

$$\theta^{t+1} = \theta^t - \eta \nabla_\theta L(\theta^t)$$

where $\eta > 0$ is the **learning rate** and $\nabla_\theta L(\theta^t)$ indicates the **gradient** of L at the point θ^t. Recall that the gradient of a function at a point is a vector in the direction that increases the function's value the most within a *neighborhood*. So by taking small steps in the oppposite direction, we obtain a solution that achieves a slightly lower loss than the current one.

In sec. 7.3.1, we will discuss methods for implementing the `grad` function above, which takes in a function and returns its gradient, which can then be evaluated at a point.

Why do we need to scale down the step size by η? The key word above is "neighborhood". The gradient only describes the function within a local region around the point, whose size depends on the function's smoothness. If we take a step that's too large, we might end up with a *worse* solution by overshooting the region where the gradient is accurate. Note that, as a result, we can't guarantee finding a *global* optimum of the function; we can only find *local* optima that are the best parameters within some neighborhood.

Another issue is that it's often expensive to compute $\nabla_\theta L$ when N is very large. Instead of calculating the gradient for every point in the dataset and averaging these, we can simply draw a **batch** of samples from the dataset and average the gradient across just these samples. Note that this is an unbiased random *estimator* of the true gradient. This algorithm is known as **stochastic gradient descent**. The added noise sometimes helps to jump to better solutions with a lower overall empirical risk.

Stepping for a moment back into the world of RL, you might wonder, why can't we simply apply gradient descent to the total reward? It turns out that the gradient of the total reward with respect to the policy parameters known as the **policy gradient**, is challenging but possible to approximate. In Chapter 7, we will do exactly this.

5.4 Examples of parameterized function classes

5.4.1 Linear regression

In linear regression, we assume that the function f is linear in the parameters:

$$\mathcal{F} = \{x \mapsto \theta^\top x \mid \theta \in \mathbb{R}^D\}$$

You may already be familiar with linear regression from an introductory statistics course. This function class is extremely simple and only contains linear functions, whose graphs look like "lines of best fit" through the training data. It turns out that, when minimizing the squared error, the empirical risk minimizer has a closed-form solution, known as the **ordinary least squares** estimator. Let us write $Y = (y_1, \dots, y_n)^\top \in \mathbb{R}^N$ and $X = (x_1, \dots, x_N)^\top \in \mathbb{R}^{N \times D}$. Then we can write

$$\begin{aligned} \hat{\theta} &= \arg\min_{\theta \in \mathbb{R}^D} \frac{1}{2} \sum_{n=1}^{N} (y_n - \theta^\top x_n)^2 \\ &= \arg\min_{\theta \in \mathbb{R}^D} \frac{1}{2} \|Y - X\theta\|^2 \\ &= (X^\top X)^{-1} X^\top Y, \end{aligned} \tag{5.12}$$

where we have assumed that the columns of X are linearly independent so that the matrix $X^\top X$ is invertible.

What happens if the columns aren't linearly independent? In this case, out of the possible solutions with the minimum empirical risk, we typically choose the one with the *smallest norm*.

Exercise 5.3 (Gradient descent finds the minimum norm solution)**.** Gradient descent on the ERM problem (eq. 5.12), initialized at the origin and using a small enough step size, eventually finds the parameters with the smallest norm. In practice, since the squared error gradient is convenient to compute, running gradient descent can be faster than explicitly computing the inverse (or pseudoinverse) of a matrix.

Assume that $N < D$ and that the data points are linearly independent.

1. Let $\hat{\theta}$ be the solution found by gradient descent. Show that $\hat{\theta}$ is a linear combination of the data points, that is, $\hat{\theta} = X^\top a$, where $a \in \mathbb{R}^N$.
2. Let $w \in \mathbb{R}^D$ be another empirical risk minimizer i.e. $Xw = y$. Show that $\hat{\theta}^\top (w - \hat{\theta}) = 0$.
3. Use this to show that $\|\hat{\theta}\| \leq \|w\|$, showing that the gradient descent solution has the smallest norm out of all solutions that fit the data. (No need for algebra; there is a nice geometric solution!)

Though linear regression may appear trivially simple, it is a very powerful tool for more complex models to build upon. For instance, to expand the expressiveness of linear models, we can first *transform* the input x using some feature mapping ϕ, i.e. $\tilde{x} = \phi(x)$, and then fit a linear model in the transformed space instead. By using domain knowledge to choose a useful feature mapping, we can obtain a powerful SL method for a particular task.

5.4.2 Neural networks

In neural networks, we assume that the function f is a composition of linear functions (represented by matrices W_i) and non-linear activation functions (denoted by σ):

$$\mathcal{F} = \{x \mapsto \sigma(W_L \sigma(W_{L-1} \dots \sigma(W_1 x + b_1) \cdots + b_{L-1}) + b_L)\}$$

where $W_\ell \in \mathbb{R}^{D_{\ell+1} \times D_\ell}$ and $b_\ell \in \mathbb{R}^{D_{\ell+1}}$ are the parameters of the i-th layer, and σ is the activation function.

This function class is highly expressive and allows for more parameters. This makes it more susceptible to overfitting on smaller datasets, but also allows it to represent more complex functions. In practice, however, neural networks exhibit interesting phenomena during training, and are often able to generalize well even with many parameters.

Another reason for their popularity is the efficient **backpropagation** algorithm for computing the gradient of the output with respect to the parameters. Essentially, the hierarchical structure of the neural network, i.e. computing the output of the network as a composition of functions, allows us to use the chain rule to compute the gradient of the output with respect to the parameters of each layer.

5.5 Key takeaways

Supervised learning is a field of machine learning that seeks to approximate some unknown function given a dataset of example input-output pairs from that function. In particular, we typically seek to compute a **prediction rule** that takes in an input value and returns a good guess for the corresponding output. We score prediction rules using a **loss function** that measures how incorrectly it guesses. We want to find a prediction rule that achieves low loss on unseen data points. We do this by searching over a class of functions to find one that minimizes the **empirical risk** over the training dataset. We finally saw two popular examples of **parameterized** function classes: linear regression and neural networks.

5.6 Bibliographic notes and further reading

Supervised learning is the largest subfield of machine learning; we do not attempt to comprehensively survey recent progress here. Rather, here are some textbooks for interested students to read further.

James et al. (2023) provides an accessible introduction to supervised learning. Hastie et al. (2013) examines the subject in even further depth and covers many relevant supervised learning methods. Nielsen (2015) provides a comprehensive introduction to neural networks and

backpropagation. Vapnik (2000) is another prominent textbook on classical statistical learning from before the "deep learning era". Bishop (2006) focuses on the Bayesian perspective.

6 Fitted Dynamic Programming Algorithms

In Chapter 2, we discussed how to solve known tabular Markov decision processes where the state and action spaces $\mathcal{S}$ and $\mathcal{A}$ were finite and small and we knew the reward function and state transitions of the environment. For most interesting tasks, however, we don't know the analytical form of the reward functions or state transitions. In such settings, we must *learn* through interaction with the environment.

Remark 6.1 (Tackling unknown environments). How can we deal with an unknown environment? One way is to learn a *model* of the environment, that is, approximations of the reward function and state transitions. We can then plan in the model environment. This is the **model-based** approach. If the approximation is good enough, the optimal policy in the model environment will be near-optimal in the true environment.

Instead of learning a model and planning within it, we could also try to learn the optimal value function and/or optimal policy directly. Such **model-free** methods are popular in practice, though they generally require many more samples.

In this short chapter, we will tackle the full RL problem by extending DP algorithms to account for unknown environments. We will also relax the assumption that the state space is small and finite: the methods we will cover support large or continuous state spaces by using *parameterized* function approximation. This will require *learning* about the environment by collecting data through interaction and then applying *supervised learning* (see Chapter 5) to approximate relevant functions.

Each of the algorithms we'll cover is analogous to one of the algorithms from Chapter 2:

Table 6.1: Fitted algorithms for general environments.

Algorithm	Description	Fitted version
Policy evaluation (sec. 2.4.3)	Given a policy π, compute its value function V^π by iterating the Bellman consistency equations.	Fitted policy evaluation (sec. 6.1)
Value iteration (sec. 2.4.4.1)	Compute the optimal value function $V^\star$ by iterating the Bellman optimality equations.	Fitted value iteration (sec. 6.2)
Policy iteration (sec. 2.4.4.2)	Repeatedly make the policy π greedy with respect to its own Q function.	Fitted policy iteration (sec. 6.3)

The term "fitted" means that, since we can no longer compute the Bellman consistency equations or Bellman optimality equations exactly, we plug data from the environment into a supervised learning algorithm to *approximately* solve these equations. As such, these are also termed "approximate DP" algorithms, or "neuro-DP" algorithms, as in Bertsekas & Tsitsiklis (1996). We will also introduce some useful language for classifying RL algorithms.

Remark 6.2 (Notation). To simplify our notation, we will assume that each state s includes the current timestep h. That is, if the original state space was $\mathcal{S}$, the augmented state space is $\mathcal{S} \times [H]$. This allows us to treat all state-dependent quantities as time-dependent without explicitly carrying around a subscript h. This also holds true in most real-world episodic tasks, where the state conveys how many timesteps have passed. For example, in tic-tac-toe, one can simply count the number of moves on the board.

6.1 Fitted policy evaluation

Recall the task of *policy evaluation:* given a policy $\pi : \mathcal{S} \to \triangle(\mathcal{A})$, compute its *Q function* Q^π, which expresses the expected total reward in a given starting state-action pair.

Remark 6.3 (Value function vs Q function). In Chapter 2, we sought to compute the *value function* V^π. Here, we'll instead approximate the action-value function Q^π, since we'll need the ability to take the *action* with the maximum expected remaining reward. If we don't know the environment, we can't compute the action-values using only V^π.

Remark 6.4 (Fixed-point policy evaluation review). The fixed-point policy evaluation algorithm (sec. 2.4.3.2) makes use of the Bellman consistency equations (Theorem 2.3), restated here for the Q-function:

$$Q^\pi(s,a) = r(s,a) + \mathop{\mathbb{E}}_{\substack{s' \sim P(\cdot|s,a) \\ a' \sim \pi(\cdot|s')}} [Q^\pi(s',a')]. \tag{6.1}$$

In fixed-point iteration, we treat eq. 6.1 not as an *equation,* but as an "operator" that takes in a function $q^i : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$ and returns an updated function q^{i+1} by substituting q^i in place of Q^π.

$$q^{i+1}(s,a) := r(s,a) + \mathop{\mathbb{E}}_{\substack{s' \sim P(\cdot|s,a) \\ a' \sim \pi(\cdot|s')}} [q^i(s',a')]. \tag{6.2}$$

Roughly speaking, q^{i+1} approximates Q^π slightly better than q^i since it incorporates a single step of reward from the environment. Since this operator is a contraction mapping (Theorem 2.8), by iterating this process, q will eventually converge to the true policy Q^π.

However, computing the update step in eq. 6.2 requires computing an expectation over the state transitions $s' \sim P(\cdot \mid s, a)$. As mentioned in the introduction, this is intractable for most real-world tasks. Either the state space is too large, or we simply don't know what P is. Instead, we will apply *supervised learning* methods to approximately solve the Bellman consistency equations using data.

Recall that supervised learning is good at learning *conditional expectations* of the form

$$f(x) = \mathbb{E}[y \mid x], \tag{6.3}$$

where x are the input features and y is the scalar response. Can we rewrite the one-step value target (eq. 6.1) as a conditional expectation? In fact, it already is! Let us use notation that makes this more clear. We explicitly treat s', a' as random variables and move the conditioning on s, a into the brackets:

$$Q^\pi(s, a) = \mathbb{E}[r(s, a) + Q^\pi(s', a') \mid s, a]. \tag{6.4}$$

We can see that the input x corresponds to s, a and the response y corresponds to $r(s, a) + q^i(s', a')$, where q^i is our current estimate of Q^π. Now we just need to obtain a dataset of input-output pairs and run empirical risk minimization (sec. 5.3). We can classify data collection strategies as either *offline* or *online*.

Definition 6.1 (Offline and online algorithms)**.** We say that a learning algorithm works *offline* if the learning is performed as a function of a static dataset, without requiring further interaction with the environment. In contrast, *online* learning algorithms require interaction with the environment during learning.

We'll begin with an offline version of fitted policy evaluation and then see an online version.

6.1.1 Offline fitted policy evaluation

In particular, we use π, the policy we're trying to evaluate, to obtain a dataset of N trajectories $\tau^1, \ldots, \tau^N \sim \rho^\pi$. Let us indicate the trajectory index in the superscript, so that

$$\tau^n = \{s_0^n, a_0^n, r_0^n, s_1^n, a_1^n, r_1^n, \ldots, s_{H-1}^n, a_{H-1}^n, r_{H-1}^n\}. \tag{6.5}$$

This would give us $N(H-1)$ samples in the dataset. We subtract one from the horizon since each (x, y) sample requires a *pair* of consecutive timesteps:

$$x_h^n = (s_h^n, a_h^n) \qquad y_h^n = r(s_h^n, a_h^n) + q^i(s_{h+1}^n, a_{h+1}^n) \tag{6.6}$$

where q^i is our current estimate of Q^π. This makes our new algorithm a fixed-point algorithm as well, since it uses q^i to compute the next iterate q^{i+1}.

Now we can use empirical risk minimization to find a function q^I that approximates the optimal Q-function.

Figure 6.1: Fitted policy evaluation.

Definition 6.2 (Fitted policy evaluation)**.** Let $\pi : \mathcal{S} \to \triangle(\mathcal{A})$ be the policy we wish to evaluate. Our goal is to compute an approximation of its Q-function Q^π.

1. Collect a dataset $\tau^1, \ldots, \tau^N \sim \rho^\pi$.
2. Initialize some function $q^0 : \mathcal{S} \times \mathcal{A} \in \mathbb{R}$.
3. For $i = 0, \ldots, I-1$:
    1. Generate labels $y^1, \ldots, y^N$ from the trajectories and the current estimate q^i, where the labels come from eq. 6.1.
    2. Set q^{i+1} to the function that minimizes the empirical risk:

$$q \leftarrow \arg\min_q \frac{1}{NH} \sum_{n=1}^{N} \sum_{h=1}^{H-2} (y_h^n - q(x_h^n))^2. \tag{6.7}$$

function fitted_evaluation(trajectories : list$_{\text{Trajectory}}$, fit : FittingMethod, π : Policy, epochs : $\mathbb{Z}$, Q_{init} : QFuncti
" Run fitted policy evaluation using the given dataset. Returns an estimate of the Q-function of the given p
$\widehat{Q} \leftarrow Q_{\text{init}}$
$X \leftarrow$ get_X(trajectories)
for epoch $\in$ tqdm(range(epochs)) **do**
$y \leftarrow$ get_y$\left(\text{trajectories}, \widehat{Q}, \pi\right)$
$\widehat{Q} \leftarrow \text{fit}(X, y)$
end for
return $\widehat{Q}$
end function

Fitted policy evaluation is offline since the "interaction phase" and the "learning phase" are disjoint. In other words, we can think of fitted policy evaluation as taking a dataset of trajectories collected by some unknown policy π_{data}, and then approximating the Q function of π_{data}.

Fitted policy evaluation is *on-policy* because the update rule uses trajectories sampled from π. Where do we need this assumption? Pay close attention to the target, and compare it to the true one-step value target (eq. 6.1):

$$y_h = r(s_h, a_h) + q(s_{h+1}, a_{h+1})$$
$$Q^\pi(s,a) = r(s,a) + \mathop{\mathbb{E}}_{\substack{s' \sim P(\cdot|s,a) \\ a' \sim \pi(\cdot|s')}} Q^\pi(s', a'). \tag{6.8}$$

Notice that (s_{h+1}, a_{h+1}) is a single sample from the joint distribution $s' \sim P(\cdot \mid s, a)$ and $a' \sim \pi(\cdot \mid s')$. If the trajectories were collected from a *different policy,* then a_{h+1} would *not* be a sample from $\pi(\cdot \mid s')$, making the target a biased sample for evaluating π.

Definition 6.3 (On-policy and off-policy algorithms)**.** We say that a learning algorithm is *on-policy* if the update rule must use data collected by the current policy. On the other hand, we call a learning algorithm *off-policy* if its update rule doesn't care about how the data was collected.

Exercise 6.1 (Off-policy fitted policy evaluation)**.** Now suppose you are given a dataset of trajectories sampled from a policy π_{data}, and you want to evaluate a *different* policy π. You are *not* given access to the environment. How could you use the dataset to evaluate π? Explain what makes this an *off-policy* algorithm.

6.1.2 Bootstrapping and target networks

Using the current guess q to compute the labels is known as **bootstrapping.** (This has nothing to do with the term bootstrapping in statistical inference.)

This term comes from the following metaphor: if you are trying to get on a horse, and there's nobody to help you up, you need to "pull yourself up by your bootstraps," or in other words, start from your existing resources to build up to something more effective.

Using a bootstrapped estimate makes the optimization more complicated. Since we are constantly updating our Q function estimate, the labels are also constantly changing, destabilizing learning. Sutton & Barto (2018) calls bootstrapping one prong of the **deadly triad** of deep reinforcement learning, alongside **function approximation** and **off-policy learning**. When an algorithm relies on all three of these properties, it is possible for learning to *diverge*, so that the fitted Q function is far from the true Q function (van Hasselt et al., 2018).

Many tricks exist to mitigate this issue in practice. One such trick is to use a *target network.* That is, when computing y, instead of using q, which is constantly changing, we maintain another *target network* q_{target} that "updates more slowly." Concretely, q_{target} is an exponential moving average of the iterates of q. Whenever we update q, we update q_{target} accordingly:

$$q_{\text{target}} \leftarrow (1 - \lambda_{\text{target}}) q_{\text{target}} + \lambda_{\text{target}} q, \tag{6.9}$$

where $\lambda_{\text{target}} \in (0, 1)$ is some mixing parameter: the larger it is, the more we update towards the current estimate q.

6.1.3 Online fitted policy evaluation

In the offline algorithm above, we collect the whole dataset before starting the learning process. What could go wrong with this? Since the environment is unknown, the dataset only contains information about some portion of the environment. When we update q, it may only learn to approximate Q^π well for states that are in the initial dataset. It would be much better if q were accurate in states that the policy will actually find itself in. This leads to the following simple shift: collect trajectories *inside* the iteration loop! This results in an *online* algorithm for fitted policy evaluation, also known as TD(0).

function fitted_policy_evaluation_online(env, π : Policy, epochs : $\mathbb{Z}$, learning_rate : $\mathbb{R}$, q_{init})
  $q \leftarrow q_{\text{init}}$
  for epoch $\in$ range(epochs) **do**
    trajectory $\leftarrow$ collect_data(env)
    for $((s, a, r), (s_{\text{next}}, a_{\text{next}}, \_)) \in \text{zip}(\text{trajectory}_{:-1}, \text{trajectory}_{1:})$ **do**
      target $\leftarrow r + q_{s_{\text{next}}, a_{\text{next}}}$
      $q_{s,a} \leftarrow q_{s,a} - \text{learning\_rate} \cdot (q_{s,a} - \text{target})$
    end for
  end for
  return q
end function

Figure 6.2: Pseudocode for online policy evaluation (TD(0))

Note that we explicitly write out one step of gradient descent on the squared "temporal difference error".

Remark 6.5 (On notation). What does TD(0) stand for? This notation suggests the existence of other TD(λ) algorithms. In fact, TD(λ) is a well-studied algorithm with a parameter $\lambda \in [0, 1]$, and TD(0) is simply the special case $\lambda = 0$. TD stands for "temporal difference": we use the value function at a different point in time to estimate the value of the current state at the current time step. The parameter λ is one way to average between the *near-term* estimates and the *long-term* estimates at later timesteps.

6.2 Fitted value iteration

We'll now explore an algorithm for computing the *optimal* value function $V^\star$ when the environment is unknown. This method is analogous to value iteration (sec. 2.4.4.1), except instead of solving the Bellman optimality equations (eq. 2.38) exactly, which would require full knowledge of the environment, we will collect a dataset of trajectories and apply *supervised learning* to solve the Bellman optimality equations *approximately*. This is exactly analogous to fitted

policy evaluation (sec. 6.1), except we use the Bellman *optimality* equations (eq. 2.38) instead of the Bellman *consistency* equations (eq. 2.9) for a given policy.

Table 6.2: How fitted value iteration relates to existing algorithms.

	Known environment	Unknown environment
Bellman *consistency* equations (evaluation)	Policy evaluation	Fitted policy evaluation
Bellman *optimality* equations (optimization)	Value iteration	Fitted value iteration

Remark 6.6 (Value iteration review). Value iteration was an algorithm we used to compute the optimal value function $V^\star : \mathcal{S} \to \mathbb{R}$ in an infinite-horizon MDP (sec. 2.4.4.1). Here, we will present the equivalent algorithm for the optimal action-value function $Q^\star : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$. The optimal action-value function satisfies the Bellman optimality equations

$$Q^\star(s,a) = r(s,a) + \mathop{\mathbb{E}}_{s' \sim P(\cdot|s,a)}[\max_{a' \in \mathcal{A}} Q^\star(s',a')]. \tag{6.10}$$

Now let us treat eq. 6.10 as an "operator" instead of an equation, that is,

$$q(s,a) \leftarrow r(s,a) + \mathop{\mathbb{E}}_{s' \sim P(\cdot|s,a)}[\max_{a' \in \mathcal{A}} q(s',a')]. \tag{6.11}$$

If we start with some guess $q : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$, and repeatedly apply the update step eq. 6.11, the iterates will eventually converge to $Q^\star$ since eq. 6.11 is a contraction mapping.

When we can't compute expectations over $s' \sim P(\cdot \mid s,a)$, we can instead apply supervised learning to *approximately* solve the Bellman optimality equations. As before, we can write $Q^\star$ explicitly as a conditional expectation

$$Q^\star(s,a) = \mathbb{E}\left[r(s,a) + \max_{a' \in \mathcal{A}} Q^\star(s',a') \mid s,a\right]. \tag{6.12}$$

From this expression, we can read off the inputs x and the targets y for supervised learning. As before, since we don't know $Q^\star$, we replace it with our current guess $q \approx Q^\star$:

$$\begin{aligned} x &:= (s,a) \\ y &:= r(s,a) + \max_{a' \in \mathcal{A}} q(s',a'). \end{aligned} \tag{6.13}$$

The only difference from fitted policy evaluation (sec. 6.1) is how we compute the targets y. Instead of using the next action from the trajectory, we use the action with the *maximum* value. Notice that these equations don't reference a policy anywhere! In other words, fitted value iteration is an *off-policy* algorithm.

Figure 6.3: Fitted policy evaluation.

6.2.1 Offline fitted value iteration

To construct an offline algorithm, we take some dataset of trajectories, and then do all of the learning without ever interacting with the environment.

Definition 6.4 (Fitted value iteration)**.** Suppose we have some dataset of trajectories $\tau^1, \ldots, \tau^N$ collected by interacting with the environment.

1. Initialize some function $q^0(s, a, h) \in \mathbb{R}$.
2. Compute $x_h^n = (s_h^n, a_h^n)$.
3. For $t = 0, \ldots, T-1$:
    1. Use q^t to generate the targets
    $$y_h^n = r_h^n + \max_{a'} q^t(s_h^n, a'). \tag{6.14}$$
    2. Set q^{t+1} to the function that minimizes the empirical risk:
    $$f^{t+1} \leftarrow \arg\min_f \frac{1}{N(H-1)} \sum_{n=1}^{N} \sum_{h=0}^{H-2} (y_h^n - f(x_h^n))^2, \tag{6.15}$$

6.2.2 Q-learning

In fitted value iteration (fig. 6.4), we collect the whole dataset beforehand from some unknown policy (or policies). This doesn't interact with the environment during the learning process. What could go wrong? Since the environment is unknown, the dataset only contains information about some portion of the environment. When we update q, it may therefore only learn to approximate $Q^\star$ well for states that are in the initial dataset. It would be much better if q was accurate in states that the policy will actually find itself in. Given access to the environment, we can instead collect trajectories while learning. This turns fitted value interation into an *online* algorithm known as **Q-learning.**

function fitted_q_iteration(trajectories, fit, epochs, Q_{init})
  $\widehat{Q} \leftarrow Q_{\text{init}} \vee Q_{\text{zero}}(\text{get\_num\_actions}(\text{trajectories}))$
  $X \leftarrow \text{get\_X}(\text{trajectories})$
  for $\_ \in \text{range}(\text{epochs})$ **do**
    $y \leftarrow \text{get\_y}\left(\text{trajectories}, \widehat{Q}\right)$
    $\widehat{Q} \leftarrow \text{fit}(X, y)$
  end for
  return $\widehat{Q}$
end function

Figure 6.4: Pseudocode for fitted value iteration.

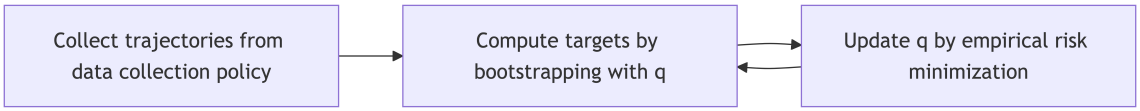


Figure 6.5: Fitted value iteration

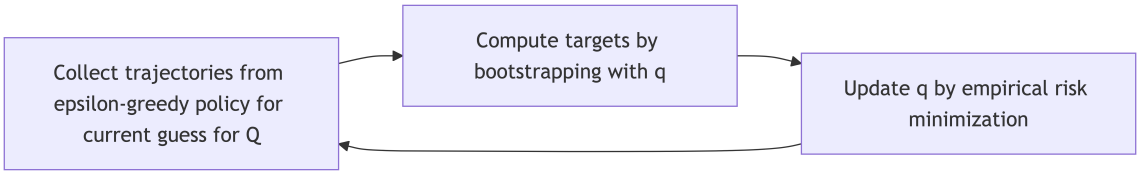


Figure 6.6: Q learning

function Q-learning(env, Q_{init}, epochs : $\mathbb{Z}$)
  $q \leftarrow Q_{\text{init}}$
  for _ $\in$ range(epochs) **do**
    trajectories $\leftarrow$ collect_trajectories(env, EpsilonGreedyPolicy(θ))
    $X \leftarrow$ get_X(trajectories)
    $y \leftarrow$ get_y(trajectories, q)
    $q \leftarrow$ fit(X, y)
  end for
  return q
end function

Note that it doesn't actually matter how the trajectories are collected, making Q-learning an **off-policy** algorithm. One common choice is to collect trajectories using an *epsilon-greedy* policy with respect to the current guess q.

Another common trick used in practice is to *grow* the dataset, called a **replay buffer,** at each iteration. Then, in the improvement step, we randomly sample a batch of (x, y) samples from the replay buffer and use these for empirical risk minimization.

function q_{learning}(env, Q_{init}, epochs : $\mathbb{Z}$)
  $q \leftarrow Q_{\text{init}}$
  for _ $\in$ range(epochs) **do**
    trajectories $\leftarrow$ collect_trajectories(env, EpsilonGreedyPolicy(θ))
    $X \leftarrow$ get_X(trajectories)
    $y \leftarrow$ get_y(trajectories, q)
    $q \leftarrow$ fit(X, y)
  end for
  return q
end function

6.3 Fitted policy iteration

We can use fitted policy evaluation to extend *policy iteration* (sec. 2.4.4.2) to this new, more general setting. The algorithm remains exactly the same – repeatedly make the policy greedy with respect to its own value function – except now we evaluate the policy π (i.e. estimate Q^π) using fitted policy evaluation.

Exercise 6.2 (Classification of fitted policy iteration)**.** Is fitted policy iteration online or offline? On-policy or off-policy?

function fitted-policy-iteration(trajectories : $\text{list}_{\text{Trajectory}}$, fit : FittingMethod, epochs : $\mathbb{Z}$, evaluation_epochs : $\mathbb{Z}$

"Run fitted policy iteration using the given dataset."

$\pi \leftarrow \pi_{\text{init}}$

for _ $\in$ range(epochs) **do**

$\widehat{Q} \leftarrow$ fitted_evaluation(trajectories, fit, π, evaluation_epochs)

$\pi \leftarrow$ q_to_greedy$\left(\widehat{Q}\right)$

end for

return π

end function

Figure 6.7: Pseudocode for fitted policy iteration.

6.4 Key takeaways

In a finite MDP where the environment is known, we can apply **dynamic programming** (sec. 2.3.2) to compute the optimal policy. But in most real-world settings, we *don't* know the state transitions P. This means we can't *exactly* compute the Bellman consistency or optimality equations (Theorem 2.3), which require taking an expectation over the next state.

In an unknown environment, we must learn from data collected from the environment. **Model-based methods** learn a model of the environment, and then plan within that model to choose the best action. **Model-free methods** instead use the data to approximate quantities of interest (e.g. the optimal Q function or optimal policy) directly.

We began by considering *offline* algorithms that used a dataset of interactions to learn some quantity of interest. We then saw the *online* equivalents that observe data by interacting with the environment.

6.5 Bibliographic notes and further reading

The fitted dynamic programming algorithms we discuss in this chapter are a special case of *temporal difference* (TD) methods, an important class of reinforcement learning algorithms. In particular, this chapter only discusses one-step methods, which can be directly generalized to n-step methods and the λ-return, the formulation most commonly used in practice. This allows us to draw direct parallels to the tabular DP methods of Chapter 2 (see tbl. 6.1). TD learning is perhaps the "central and novel [idea of] reinforcement learning" (Sutton & Barto, 2018, ch. 6).

TD methods are based in the psychology of animal learning

Witten (1977), which proposed the tabular TD(0) method, appears to be the earliest instance of a temporal difference algorithm.

Richard Sutton's PhD thesis (Sutton, 1988) proved theoretical guarantees for many of the algorithms presented here. Sutton & Barto (2018) gives a comprehensive discussion of the methods discussed here.

In later work, Hausknecht & Stone (2017) made the Q-function recurrent, so that it depends on the past history of states. The resulting policy is non-Markovian, which additionally accounts for the partial observability of the environment.

Maei et al. (2009) further discusses the **deadly triad** of off-policy sampling, function approximation, and value bootstrapping.

Deep RL rose to prominence when Mnih et al. (2013) used a deep Q network to achieve state of the art performance on the Atari games. However, the combination of deep learning and the underlying variance in RL problems made it challenging to optimize these neural networks. Many changes have been proposed. Wang et al. (2016) suggests learning the value function and advantage function separately in a "dueling" architecture. Hasselt (2010) suggested the "double learning" algorithm to compensate for Q-learning's tendency to overestimate the true values. Hasselt et al. (2016) applied double learning to deep neural networks. Hessel et al. (2018) combined

Bertsekas & Tsitsiklis (1996) coined the term "neuro-dynamic programming" to refere to the combination of (artificial) neural networks with dynamic programming techniques.

7 Policy Gradient Methods

7.1 Introduction

The core task of RL is finding the **optimal policy** in a given environment. This is essentially an *optimization problem* (i.e. computing a minimum or maximum): out of some class of policies Π, we want to find the one that achieves the maximum expected total reward:

$$\hat{\pi} = \arg\max_{\pi \in \Pi} \mathop{\mathbb{E}}_{\tau \sim \rho^\pi}[R(\tau)] \quad \text{where} \quad R(\tau) := \sum_{h=0}^{H-1} r(s_h, a_h). \tag{7.1}$$

In a known environment with a finite set of states and actions, we can compute the optimal policy using dynamic programming in $O(H \cdot |\mathcal{S}|^2 \cdot |\mathcal{A}|)$ steps (sec. 2.3.2). In more general settings, though, such as when the environment is *unknown*, it is typically intractable to compute the optimal policy exactly. Instead, we start from some random policy, and then iteratively *improve* it by interacting with the environment. Policy iteration (sec. 2.4.4.2) and iterative LQR (sec. 3.6.4) are two algorithms we've seen that do this.

In this chapter, we will explore *policy gradient algorithms,* which also iteratively improve a policy. The *gradient* of a function at a specific input point is the direction that increases the output value most rapidly. If we think of the expected total reward as a function of the policy, we can repeatedly shift the policy by the gradient of that function to obtain policies that achieve higher expected total rewards.

Policy gradient methods are responsible for groundbreaking applications including AlphaGo, OpenAI Five, and large language models. This chapter will explore:

1. Examples of *parameterized policies* that can be expressed in terms of a finite set of *parameters.*
2. *Gradient descent,* a general optimization algorithm for differentiable functions.
3. Different estimators of the *policy gradient,* enabling us to apply (stochastic) gradient descent to RL.
4. *Proximal (policy) optimization* techniques that ensure the steps taken are "not too large". These are some of the most popular and widely used RL algorithms at the time of writing.

Remark 7.1 (Conventions). Policy gradient algorithms typically optimize over *stochastic policies* (def. 2.6). In this chapter, we will only discuss the case of stochastic policies, though there exist policy gradient algorithms for deterministic policies as well (Silver et al., 2014).

To ease notation, we will treat the horizon H as finite and avoid adding a discount factor γ. We make the policy's dependence on the timestep h implicit by assuming that the state space conveys information about the timestep. We also use

$$R(\tau) := \sum_{h=0}^{H-1} r_h \tag{7.2}$$

to denote the total reward in a trajectory.

7.2 Parameterized policies

Optimizing over the entire space of policies is usually intractable: there's just too many! There are there are $|\mathcal{A}|^{|\mathcal{S}|}$ deterministic mappings from state to actions. In continuous state spaces, $|\mathcal{S}|$ is infinite, so the space of policies becomes *infinite-dimensional*. Infinite-dimensional spaces are usually hard to optimize over. Instead, we choose a *parameterized policy class* with a finite number D of adjustable parameters. Each parameter vector corresponds to a mapping from states to actions. (We also discussed the notion of a parameterized function class in sec. 5.3.2). The following examples seek to make this more concrete.

Example 7.1 (Tabular representation)**.** If both the state and action spaces are finite, perhaps we could simply learn a preference value $\theta_{s,a}$ for each state-action pair, resulting in a table of $|\mathcal{S}||\mathcal{A}|$ values. These are the D *parameters* of the policy. Then, for each state, to compute the distribution over actions, we perform a **softmax** operation: we exponentiate each of the values, and then normalize to form a valid distribution.

$$\pi_\theta^{\text{softmax}}(a \mid s) = \frac{\exp(\theta_{s,a})}{\sum_{s,a'} \exp(\theta_{s,a'})}. \tag{7.3}$$

In this way, we turn a vector of length $|\mathcal{S}||\mathcal{A}|$ into a mapping from $\mathcal{S} \to \triangle(\mathcal{A})$. However, this doesn't make use of any structure in the states or actions, so while this is flexible, it is also prone to overfitting. For small state and action spaces, it makes more sense to use a dynamic programming algorithm from Chapter 2.

For a given *parameterization,* such as the one above, we call the set of resulting policies the **parameterized policy class,** that is,

$$\{\pi_\theta \mid \theta \in \mathbb{R}^D\}, \tag{7.4}$$

where D is the fixed number of parameters. Let us explore some more examples of parameterized policy classes.

Example 7.2 (Linear in features)**.** Another approach is to map each state-action pair into some **feature space** $\phi(s, a) \in \mathbb{R}^D$. Then, to map a feature vector to a probability, we take a linear combination of the features and take a softmax. The parameters θ are the coefficients of the linear combination:

$$\pi_\theta^{\text{linear}}(a \mid s) = \frac{\exp(\theta^\top \phi(s, a))}{\sum_{a'} \exp(\theta^\top \phi(s, a'))}. \tag{7.5}$$

Another interpretation is that θ represents the feature vector of the "desired" state-action pair, as state-action pairs whose features align closely with θ are given higher probability.

Example 7.3 (Neural policies)**.** More generally, we could map states and actions to unnormalized scores via some parameterized function $f_\theta : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$, such as a neural network, and choose actions according to a softmax:

$$\pi_\theta^{\text{neural}}(a \mid s) = \frac{\exp(f_\theta(s, a))}{\sum_{a'} \exp(f_\theta(s, a'))}. \tag{7.6}$$

Remark 7.2 (Why softmax)*.* The three parameterized policy classes above all use the *softmax* operation. Of course, this isn't the only way to turn a list of values into a probability distribution: for example, you could also subtract by the minimum value and divide by the range. Why is softmax preferred in practice? One reason is that it is a *smooth, differentiable* function of the input values. It is also nice to work with analytically, and has other interpretations from physics, where it is known as the *Gibbs* or *Boltzmann* distribution, and economics, where it is known as the *Bradley-Terry model* for ranking choices.

Example 7.4 (Diagonal Gaussian policies for continuous action spaces)**.** Consider an N-dimensional action space $\mathcal{A} = \mathbb{R}^N$. Then for a stochastic policy, we could predict the *mean* action and then add some random noise to it. For example, we could use a linear model to predict the mean action and then add some noise $\epsilon \sim \mathcal{N}(0, \sigma^2 I)$ to it.

$$\pi_\theta(\cdot \mid s) = \mathcal{N}(\theta^\top \phi(s), \sigma^2 I).$$

Now we've seen some examples of *parameterized policies.* Optimizing over a parameterized policy class makes the policy optimization problem finite-dimensional:

$$\begin{aligned} \hat{\theta} &= \arg\max_{\theta \in \mathbb{R}^D} J(\theta) \\ \text{where} \quad J(\theta) &:= \mathop{\mathbb{E}}_{\tau \sim \rho^{\pi_\theta}} \sum_{h=0}^{H-1} r(s_h, a_h). \end{aligned} \tag{7.7}$$

This enables us to apply one of the most popular and general optimization algorithms: *gradient descent.*

7.3 Gradient descent

Gradient descent is an optimization algorithm that can be applied to any differentiable function. That is, it is a tool for solving

$$\arg\min_{\theta \in \mathbb{R}^D} J(\theta), \tag{7.8}$$

where $J(\theta) \in \mathbb{R}$ is the function to be minimized. For two-dimensional inputs, a suitable analogy for this algorithm is making your way down a hill (with no cliffs), where you keep taking steps in the steepest direction downwards from your current position. Your horizontal position (x, z) is the input and your vertical position y is the function to be minimized. The *slope* of the mountain at your current position can be expressed using the *gradient*, written $\nabla y(x, z) \in \mathbb{R}^2$. This can be computed as the vector of partial derivatives,

$$\nabla y(x, z) = \begin{pmatrix} \frac{\partial y}{\partial x} \\ \frac{\partial y}{\partial z} \end{pmatrix}. \tag{7.9}$$

To calculate the *slope* (aka "directional derivative") of the mountain in a given direction $(\Delta x, \Delta z)$, you take the dot product of the difference vector with the gradient:

$$\Delta y = \begin{pmatrix} \Delta x \\ \Delta z \end{pmatrix} \cdot \nabla y(x, z), \tag{7.10}$$

where x, z is your current position. What direction should we walk to go down the hill as quickly as possible? That is, we want to find the direction $(\Delta x, \Delta z)$ that minimizes the slope:

$$\arg\min_{\Delta x, \Delta z} \Delta y. \tag{7.11}$$

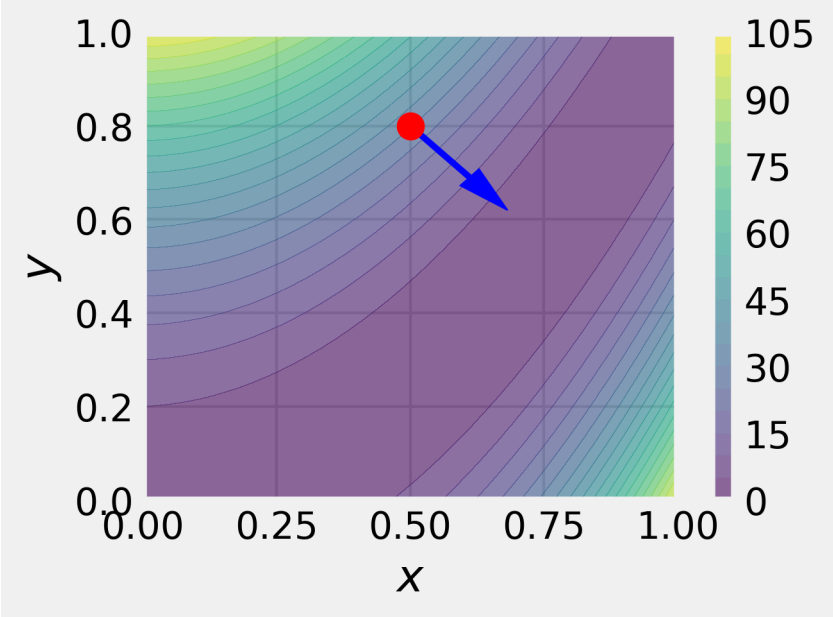


Figure 7.1: Example of gradient descent on the Rosenbrock function.

We use the useful fact that, for a given vector v, the direction u that minimizes the dot product $u \cdot v$ points in the opposite direction to v. This should make intuitive sense, since $u \cdot v = \|u\|\|v\| \cos\theta$, where θ is the angle between u and v, and the cosine function is minimized at $\theta = \pi$. Applying this to eq. 7.10, we see that the direction of steepest decrease is opposite to the gradient. "Walking" in that direction corresponds to subtracting a multiple of the gradient from your current position:

$$\begin{pmatrix} x^{t+1} \\ z^{t+1} \end{pmatrix} = \begin{pmatrix} x^t \\ z^t \end{pmatrix} - \eta \nabla y(x^t, z^t)$$

where t denotes the iteration of the algorithm and $\eta > 0$ is a "step size" hyperparameter that controls the size of the steps we take. (Note that we could also vary the step size across iterations, that is, $\eta^0, \ldots, \eta^T$.)

Remark 7.3 (Minimization vs maximization). Optimization problems are usually posed as *minimization* problems by convention. To solve a *maximization* problem with gradient descent, you could simply take steps in the direction of the *positive* gradient. This also corresponds to flipping the sign of the objective function.

The case of a two-dimensional input is easy to visualize. The analogy to climbing a hill (if we were maximizing the function) is why gradient descent is sometimes called **hill climbing** and the graph of the objective function is called the **loss landscape**. (The term "loss" comes from supervised learning, which you can read about in Chapter 5.) But this idea can be

straightforwardly extended to higher-dimensional inputs. From now on, we'll use J to denote the function we're trying to maximize, and θ to denote the parameters being optimized over. (In the above example, $\theta = \begin{pmatrix} x & z \end{pmatrix}^\top$). Let's summarize the algorithm in general:

Definition 7.1 (Gradient descent)**.** Suppose we are trying to solve the optimization problem

$$\theta^\star = \arg\min_{\theta \in \mathbb{R}^D} J(\theta), \tag{7.12}$$

where $J(\theta) \in \mathbb{R}$ is the differentiable function to be minimized. Gradient descent starts with an initial guess θ^0 and then takes steps in the direction of $-\nabla J(\theta^t)$, where θ^t is the current iterate:

$$\theta^{t+1} = \theta^t - \eta \nabla J(\theta^t). \tag{7.13}$$

Note that we scale $\nabla J(\theta^t)$ by the **step size** $\eta > 0$, also known as the learning rate.

```
function gradient_descent(θ_init : ℝ^D, J : (ℝ^D) → ℝ, η : ℝ, n_steps : ℕ)
  θ ← θ_init
  history ← [θ]
  for step ∈ range(n_steps) do
    θ ← θ + η · ∇(J)(θ)
    history.append(θ)
  end for
  return (θ, jnp.array(history))
end function
```

Figure 7.2: Pseudocode for gradient descent.

Notice that the parameters will stop changing once $\nabla J(\theta) = 0$. Once we reach this **stationary point,** our current parameters are 'locally optimal' (assuming we're at a local minimum): it's impossible to increase the function by moving in any direction. If J is *convex* (i.e. the function looks like an upward-curved bowl), then the only point where this happens is at the *global optimum.* Otherwise, if J is nonconvex, the best we can hope for is a *local optimum.* We won't go deeper into the theory of convex functions here. For more details, refer to the textbook of Boyd & Vandenberghe (2004).

Remark 7.4 (Limitations of gradient descent)*.* Gradient descent and its variants are responsible for most of the major achievements in modern machine learning. It is important to note, however, that for many problems, gradient descent is *not* a good optimization algorithm to reach for! If you have more information about the problem, such as access to *second* derivatives, you can apply more powerful optimization algorithms that converge much more rapidly. Read Nocedal & Wright (2006) if you're curious about the field of numerical optimization.

7.3.1 Computing derivatives

How does a computer compute the gradient of a function?

One way is **symbolic differentiation,** which is similar to the way you might compute it by hand: the computer applies a list of rules to transform the *symbols* involved. Python's `sympy` package (Meurer et al., 2017) supports symbolic differentiation. However, functions implemented as algorithms in code may not always have a straightforward symbolic representation.

Another way is **numerical differentiation,** which is based on the limit definition of a (directional) derivative:

$$\nabla_u J(x) = \lim_{\varepsilon \to 0} \frac{J(x + \varepsilon u) - J(x)}{\varepsilon} \tag{7.14}$$

Then, we can substitute a small value of ε on the r.h.s. to approximate the directional derivative. How small, though? Depending on how smooth the function is and how accurate our estimate needs to be, we may need such a small value of ε that typical computers will run into *rounding errors*. Also, to compute the full gradient, we would need to compute the r.h.s. once for each input dimension. This is an issue if computing J is expensive.

Automatic differentiation achieves the best of both worlds. Like symbolic differentiation, we manually implement the derivative rules for a few basic operations. However, instead of executing these on the *symbols*, we execute them on the *values* when the function gets called, like in numerical differentiation. This allows us to differentiate through programming constructs such as branches or loops, and doesn't involve any arbitrarily small values. Baydin et al. (2018) provides an accessible survey of automatic differentiation. At the time of writing, all of the popular Python libraries for machine learning, such as PyTorch (Ansel et al., 2024) and Jax (Bradbury et al., 2018), use automatic differentiation.

7.3.2 Stochastic gradient descent

In real applications, computing the gradient of the target function is not so simple. As an example from supervised learning, $J(\theta)$ might be the sum of squared prediction errors across an entire training dataset. If our dataset is very large, it might not fit into our computer's memory, making it impossible to evaluate $\nabla J(\theta)$ at once. We will see that computing the exact gradient in RL faces a similar challenge where computing the gradient would require computing a complicated integral.

In these cases, we can compute some *gradient estimate*

$$g_x(\theta) \approx \nabla J(\theta) \tag{7.15}$$

of the gradient at each step, using some observed data x, and walk in that direction instead. This is called **stochastic** gradient descent. In the SL example above, we might randomly choose a *minibatch* of samples and use them to estimate the true prediction error.

function sgd(rng : jr.PRNGKey, $\theta_{\text{init}} : \mathbb{R}^D$, $g : (\text{jr.PRNGKey} \times \mathbb{R}^D) \to \mathbb{R}^D$, $\eta : \mathbb{R}$, $n_{\text{steps}} : \mathbb{Z}$)
  $\theta \leftarrow \theta_{\text{init}}$
  rngs $\leftarrow$ jr.split$\big(\text{rng}, n_{\text{steps}}\big)$
  history $\leftarrow [\theta]$
  for step $\in$ range$\big(n_{\text{steps}}\big)$ **do**
    $\theta \leftarrow \theta + \eta \cdot g\big(\text{rngs}_{\text{step}}, \theta\big)$
    history.append(θ)
  end for
  return $(\theta, \text{jnp.array(history)})$
end function

Figure 7.3: Pseudocode for stochastic gradient descent.

What makes one gradient estimator better than another? Ideally, we want this estimator to be *unbiased;* that is, on average, it matches a single true gradient step.

Definition 7.2 (Unbiased gradient estimator)**.** We call the gradient estimator g **unbiased** if, for all $\theta \in \mathbb{R}^D$,

$$\mathop{\mathbb{E}}_{x \sim p_\theta}[g_x(\theta)] = \nabla J(\theta), \tag{7.16}$$

where p_θ denotes the distribution of the observed data x.

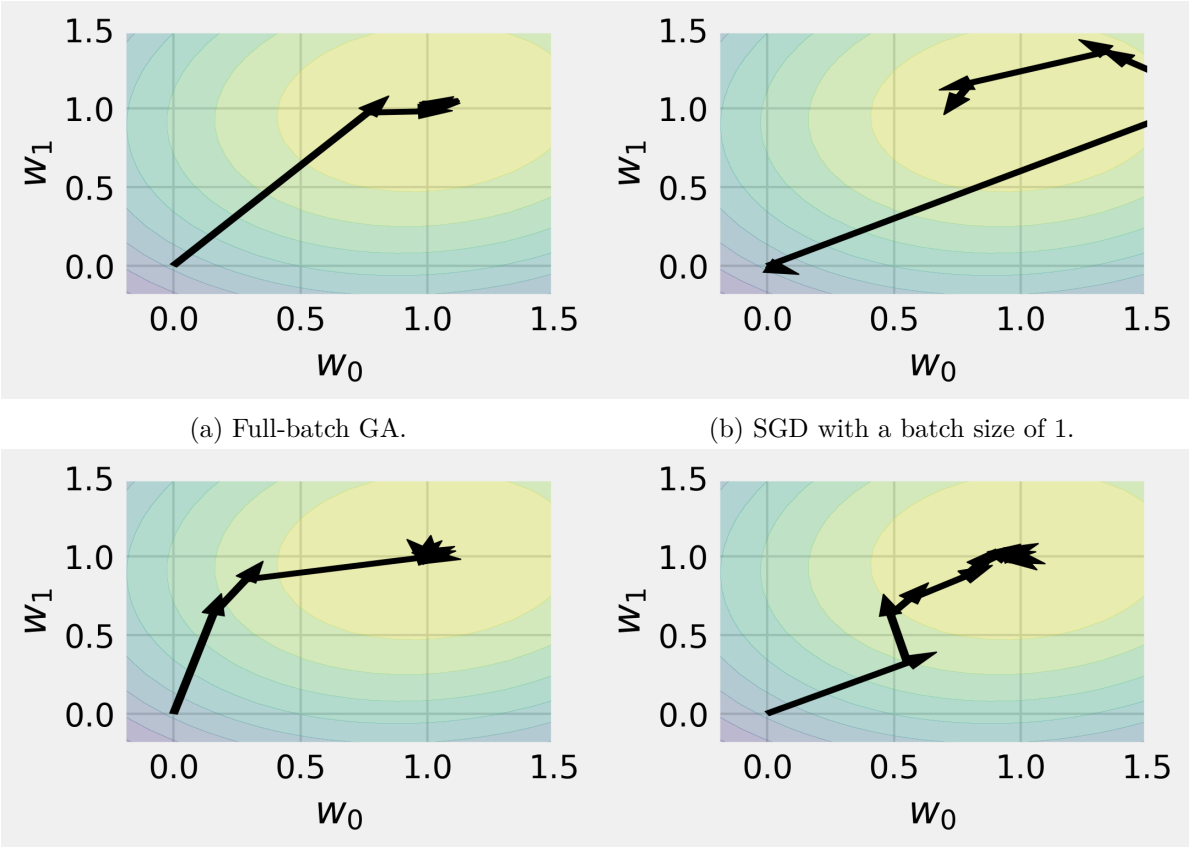
We also want the *variance* of the estimator to be low so that its performance doesn't change drastically at each step.

We can actually show that, for many "nice" functions, in a finite number of steps, SGD will find a θ that is "close" to a stationary point. In another perspective, for such functions, the local "landscape" of J around θ becomes flatter and flatter the longer we run SGD.

Theorem 7.1 (SGD convergence)**.** *More formally, suppose we run SGD for K steps, using an unbiased gradient estimator. Let the step size η^k scale as $O(1/\sqrt{k})$. Then if J is bounded and β-smooth (see below), and the* norm *of the gradient estimator has a bounded second moment σ^2,*

$$\|\nabla J(\theta^i)\|^2 \leq O\left(M\beta\sigma^2/K\right).$$

We call a function β-smooth if its gradient is Lipschitz continuous with constant β:



(a) Full-batch GA.

(b) SGD with a batch size of 1.

(c) SGD with a batch size of 5.

(d) SGD with a batch size of 10.

Figure 7.4: GA and SGD on a two-dimensional convex optimization task. Note that a larger batch size reduces the variance in the step direction.

$$\|\nabla J(\theta) - \nabla J(\theta')\| \leq \beta \|\theta - \theta'\|.$$

We'll now see a concrete application of stochastic gradient descent in the context of policy optimization.

7.4 Policy (stochastic) gradient descent

Policy gradient methods boil down to applying gradient descent to the policy optimization problem for a chosen parameterized policy class (eq. 7.4):

$$\theta^{t+1} = \theta^t + \eta \nabla J(\theta^t) \quad \text{where} \quad J(\theta) = \mathop{\mathbb{E}}_{\tau \sim \rho^{\pi_\theta}} \left[\sum_{h=0}^{H-1} r(s_h, a_h) \right]. \tag{7.17}$$

The challenges lie mostly in computing unbiased gradient estimators (def. 7.2) and in *constraining* the size of each update to improve stability of the learning algorithm.

To provide some intuition for the gradient estimators we will later derive, the following section constructs a general policy gradient algorithm in a "bottom-up" way. The remainder of the chapter focuses on methods for improving the stability of the policy gradient algorithm.

Remark 7.5 (On-policy). Note that since the gradient estimator is computed using data collected by the current policy, policy gradient algorithms are generally *on-policy* (def. 6.3). Note that on-policy algorithms generally suffer from *worse sample efficiency* than off-policy algorithms: while off-policy algorithms (such as Q-learning (sec. 6.2.2)) can use data collected by any method, on-policy algorithms can only make use of data from the *current* policy.

7.4.1 Introduction to policy gradient methods

In this section, we'll intuitively construct an iterative algorithm for improving the parameters. We'll build up to true policy gradient estimators by considering a series of update rules.

Remember that in RL, the primary goal is to find the *optimal policy* that achieves the highest total reward. Put simply, we want policies that *take better actions more often.* Breaking that down, we need

1. a way to shift the policy to make certain actions more likely, and
2. a way to measure how good an action is in a given state.

Let's tackle the first task: getting the policy to take action a in state s. Back in the tabular setting, for a deterministic policy π, we simply assigned $\pi(s) \leftarrow a$. Now that we're using a parameterized policy class, we can't directly assign a value to $\pi(s)$. Instead, we adjust the parameters θ of the policy to maximize the probability of taking a:

$$\hat{\theta} = \arg\max_{\theta} \pi_\theta(a \mid s). \tag{7.18}$$

Assuming the output of the policy is differentiable with respect to its parameters, we can apply gradient descent:

$$\begin{aligned} \theta^{t+1} &= \theta^t + \eta g_{s,a}(\theta^t) \\ \text{where} \quad g_{s,a}(\theta) &:= \nabla \pi_\theta(a \mid s). \end{aligned} \tag{7.19}$$

The notation g is chosen to remind you of a gradient estimator (eq. 7.15). We will draw the relationship between our intuitive approach and SGD more concretely later on.

Now let's approach the second task: how do we choose which actions to take more often? Suppose we have some random variable ψ (that's the Greek letter "psi") that is correlated with how "good" action a is in state s. Later in the chapter, we'll explore concrete choices for ψ, but for now, we'll leave its identity a mystery. Then, to update the policy, we could sample an action $a \sim \pi_{\theta^i}(\cdot \mid s)$, and weight the corresponding gradient by ψ. We will therefore call ψ the **gradient coefficient**.

$$g_{s,a}(\theta) := \psi \nabla \pi_\theta(a \mid s). \tag{7.20}$$

To illustrate this, suppose the policy takes one action a_0 and obtains a coefficient of $\psi_0 = 2$ and then takes a second action a_1 with a coefficient of $\psi_1 = -1$. Then we would update $\theta \leftarrow \theta + 2\eta \nabla \pi_\theta(a_0 \mid s)$ and then $\theta \leftarrow \theta - \eta \nabla \pi_\theta(a_1 \mid s)$.

Exercise 7.1 (An alternative approach)**.** Compare this with the policy iteration update (sec. 2.4.4.2), where we updated the deterministic policy according to $\pi(s) = \arg\max_{a \in \mathcal{A}} Q^\pi(s, a)$. In our current setting, why not just solve for $\theta^{t+1} = \arg\max_\theta \pi_\theta(a^\star \mid s)$, where $a^\star = \arg\max_{a \in \mathcal{A}} Q^{\pi_{\theta^i}}(a \mid s)$, instead of sampling an action from our policy? What type of action space does this approach assume? What does the added stochasticity grant you?

But the gradient estimator eq. 7.20 has an issue: the amount that we encourage action a depends on how *often* the policy takes it. This could lead to a positive feedback loop where the most common action becomes more and more likely, regardless of its quality. To cancel out this factor, we divide by the action's likelihood:

$$
\begin{aligned}
g_{s,a}(\theta) &:= \psi \frac{\nabla \pi_\theta(a \mid s)}{\pi_\theta(a \mid s)} \\
&= \psi \nabla \log \pi_\theta(a \mid s).
\end{aligned} \tag{7.21}
$$

Now we can extend this across the entire time horizon. Suppose we use π_{θ^i} to roll out a trajectory $\tau = (s_0, a_0, \dots, s_{H-1}, a_{H-1})$ and compute eq. 7.21 at each step of the trajectory. We compute a gradient coefficient at each timestep, so we denote each instance by $\psi_h(\tau)$.

$$
g_\tau(\theta) := \sum_{h=0}^{H-1} \psi_h(\tau) \nabla \log \pi_\theta(a_h \mid s_h). \tag{7.22}
$$

To reduce the variance, we could roll out multiple trajectories, and average the gradient steps across them. This gives us the general form of the **policy gradient algorithm**:

Definition 7.3 (General policy gradient algorithm)**.** Suppose we are given an expression for the gradient coefficients $\psi_h(\tau)$. Then we can perform policy gradient optimization as follows.

At each iteration $t = 0, \dots, T-1$ of the algorithm, we sample N trajectories $\tau^n = (s_0^n, a_0^n, r_0^n, \dots, s_{H-1}^n, a_{H-1}^n, r_{H-1}^n)$, and compute the update rule

$$
\begin{aligned}
\theta^{t+1} &= \theta^t + \eta \frac{1}{N} \sum_{n=1}^{N} g_{\tau^n}(\theta^t) \\
\text{where} \quad g_\tau(\theta) &= \sum_{h=0}^{H-1} \psi_h(\tau) \nabla \log \pi_\theta(a_h \mid s_h).
\end{aligned} \tag{7.23}
$$

This algorithm allows us to optimize a policy by sampling trajectories from it and computing the gradient-log-likelihoods (sometimes called the **scores**) of the chosen actions. Then we can update the parameters θ in the direction given by eq. 7.23 to obtain a new policy that chooses better actions more often.

Remark 7.6 (Summary of intuitive derivation). Let us review how we arrived at this expression:

1. We take the gradient of the policy to encourage certain actions (eq. 7.19).
2. We encourage actions proportionally to their advantage (eq. 7.20).
3. We correct for the policy's sampling distribution (eq. 7.21).
4. We extend this to each step of the trajectory (eq. 7.22).
5. We sample multiple trajectories to reduce variance (eq. 7.23).

The last piece is to figure out what ψ stands for.

function policy_gradient(env : gym.Env, $\pi, \theta : \mathbb{R}^D$, get_psi : (list$_{\text{"Transition"}}$) $\to \mathbb{R}^H$)
  "Estimate the policy gradient using REINFORCE."
  $g \leftarrow$ jnp.zeros_like(θ)
  $\tau \leftarrow$ sample_trajectory(env, $\pi(\theta)$)
  psis $\leftarrow$ get_psi($\pi(\theta), \tau$)
  for $((s, a, r), \psi) \in$ zip(τ, psis) **do**
    function policy_log_likelihood($\theta : \mathbb{R}^D$)
      return $\log \pi(\theta)\,(s, a)$
    end function
    $g \leftarrow g + \psi \cdot \nabla(\text{policy\_log\_likelihood})\,(\theta)$
  end for
  return g
end function

Figure 7.5: Pseudocode for the general policy gradient algorithm.

Exercise 7.2 (Brainstorming)**.** Can you think of any possibilities? $\psi_h(\tau)$ should correlate with the quality of the action taken at time h. It may depend on the current policy π_{θ^i} or any component of the trajectory $(s_0, a_0, r_0, \ldots, s_{H-1}, a_{H-1}, r_{H-1})$.

We won't keep you waiting: it turns out that if we set $\psi_h(\tau)$ to

1. $R(\tau)$ (the total reward of the trajectory) (eq. 7.25),
2. $\sum_{h'=h}^{H-1} r(s_{h'}, a_{h'})$ (the remaining reward in the trajectory),
3. $Q^{\pi_{\theta^i}}(s_h, a_h)$ (the current policy's Q-function), or
4. $A^{\pi_{\theta^i}}(s_h, a_h)$ (the current policy's advantage function),

among other possibilities, the gradient term of eq. 7.23 is actually an *unbiased* estimator (def. 7.2) of the true "policy gradient" $\nabla J(\theta)$ (see eq. 7.7). That is, for any of the ψ above, updating the parameters according to the general policy gradient algorithm eq. 7.23 *is* (minibatch) stochastic gradient descent on the expected total reward $J(\theta)$, with the gradient estimator

$$\begin{aligned} g_\tau(\theta) &= \sum_{h=0}^{H-1} \psi_h(\tau) \nabla \log \pi_\theta(a_h \mid s_h) \\ \text{where} \quad \mathop{\mathbb{E}}_{\tau \sim \rho^{\pi_\theta}}[g_\tau(\theta)] &= \nabla J(\theta). \end{aligned} \tag{7.24}$$

7.4.2 The REINFORCE policy gradient

We begin by showing that setting $\psi_h = \sum_{h'=0}^{H-1} r_{h'}$, i.e. the total reward of the trajectory, provides an unbiased gradient estimator.

Theorem 7.2 (Using the total reward is unbiased)**.** *Substituting*

$$\psi_h(\tau) := R(\tau) := \sum_{h'=0}^{H-1} r_{h'} \tag{7.25}$$

into the general policy gradient estimator eq. 7.23 gives an unbiased estimator. That is,

$$\begin{aligned} \nabla J(\theta) &= \mathop{\mathbb{E}}_{\tau\sim\rho^{\pi_\theta}}[g_\tau^R(\theta)] \\ \textit{where} \quad g_\tau^R(\theta) &:= \sum_{h=0}^{H-1} R(\tau)\nabla \log \pi_\theta(a_h \mid s_h). \end{aligned} \tag{7.26}$$

The "R" stands for REINFORCE, which stands for "REward Increment = Nonnegative Factor × Offset Reinforcement × Characteristic Eligibility" (Williams, 1992, p. 234). (We will not elaborate further on this etymology.)

Proof via calculus. As our first step towards constructing an unbiased policy gradient estimator, let us simplify the expression

$$\nabla J(\theta) = \nabla \mathop{\mathbb{E}}_{\tau\sim\rho^{\pi_\theta}}[R(\tau)]. \tag{7.27}$$

In supervised learning, we were able to swap the gradient and expectation. That was because the *function* being averaged depended on the parameters, not the distribution itself:

$$\nabla \mathop{\mathbb{E}}_{(x,y)\sim p}[L(f_\theta(x), y)] = \mathop{\mathbb{E}}_{(x,y)\sim p}[\nabla L(f_\theta(x), y)]. \tag{7.28}$$

Here, though, the distribution depends on the parameters, and the function being averaged does not. One way to compute this type of derivative is to use the identity

$$\nabla \rho^{\pi_\theta}(\tau) = \rho^{\pi_\theta}(\tau)\nabla \log \rho^{\pi_\theta}(\tau). \tag{7.29}$$

By expanding the definition of expected value, we can compute the correct value to be

$$\begin{aligned}\nabla J(\theta) &= \nabla \mathop{\mathbb{E}}_{\tau\sim\rho^{\pi_\theta}}[R(\tau)] \\ &= \int \nabla \rho^{\pi_\theta}(\tau) R(\tau) \\ &= \int \rho^{\pi_\theta}(\tau) \nabla \log \rho^{\pi_\theta}(\tau) R(\tau) \\ &= \mathop{\mathbb{E}}_{\tau\sim\rho^{\pi_\theta}}[\nabla \log \rho^{\pi_\theta}(\tau) R(\tau)].\end{aligned} \tag{7.30}$$

Now we deal with the $\nabla \log \rho^{\pi_\theta}(\tau)$ term, that is, the gradient-log-likelihood (aka **score**) of the trajectory. Recall Theorem 2.1, in which we showed that when the state transitions are Markov (i.e. s_h only depends on s_{h-1}, a_{h-1}) and the policy is history-independent (i.e. $a_h \sim \pi_\theta(\cdot \mid s_h)$), we can autoregressively write out the likelihood of a trajectory under the policy π_θ. Taking the log of the trajectory likelihood turns the products into sums:

$$\log \rho^{\pi_\theta}(\tau) = \log P_0(s_0) + \sum_{h=0}^{H-1} \Big(\log \pi_\theta(a_h \mid s_h) + \log P(s_{h+1} \mid s_h, a_h) \Big) \tag{7.31}$$

When we take the gradient with respect to the parameters θ, only the $\log \pi_\theta(a_h \mid s_h)$ terms depend on θ:

$$\nabla \log \rho^{\pi_\theta}(\tau) = \sum_{h=0}^{H-1} \nabla \log \pi_\theta(a_h \mid s_h). \tag{7.32}$$

Substituting this into eq. 7.30 gives

$$\begin{aligned}\nabla J(\theta) &= \mathop{\mathbb{E}}_{\tau\sim\rho^{\pi_\theta}} \left[\sum_{h=0}^{H-1} R(\tau) \nabla \log \pi_\theta(a_h \mid s_h) \right] \\ &= \mathop{\mathbb{E}}_{\tau\sim\rho^{\pi_\theta}} [g^{\mathrm{R}}_\theta(\tau)],\end{aligned}$$

showing that the REINFORCE policy gradient (eq. 7.25) is unbiased. □

Proof via importance sampling. Another way of deriving Theorem 7.2 involves a technique known as **importance sampling**. We'll demonstrate this approach here since it will come in handy later on. Importance sampling is useful when we want to estimate an expectation over a distribution that is hard to sample from. Instead, we can sample from a different distribution that supports the same values, and then reweight the samples according to the likelihood ratio between the two distributions.

Theorem 7.3 (Importance sampling)**.** *Consider some random variable $x \in \mathcal{X}$ with density function p. Let q be the density function of another distribution on $\mathcal{X}$ that supports all of p, that is, $q(x) = 0$ only if $p(x) = 0$. Then*

$$\mathop{\mathbb{E}}_{x\sim p}[f(x)] = \mathop{\mathbb{E}}_{x\sim q}\left[\frac{p(x)}{q(x)}f(x)\right]. \tag{7.33}$$

Proof. We expand the definition of expected value:

$$\begin{aligned}\mathop{\mathbb{E}}_{x\sim p}[f(x)] &= \sum_{x\in\mathcal{X}} f(x)p(x) \\ &= \sum_{x\in\mathcal{X}} f(x)\frac{p(x)}{q(x)}q(x) \\ &= \mathop{\mathbb{E}}_{x\sim q}\left[\frac{p(x)}{q(x)}f(x)\right].\end{aligned} \tag{7.34}$$

□

Exercise 7.3 (Importance sampling for a biased coin)**.** Suppose you are a student and you determine your study routine by flipping a biased coin. Let $x \in \{\text{heads}, \text{tails}\}$ be the result of the coin flip. The coin shows heads twice as often as it shows tails:

$$p(x) = \begin{cases} 2/3 & x = \text{heads} \\ 1/3 & x = \text{tails}. \end{cases} \tag{7.35}$$

Suppose you study for $f(x)$ hours, where

$$f(x) = \begin{cases} 1 & x = \text{heads} \\ 2 & x = \text{tails}. \end{cases} \tag{7.36}$$

One day, you lose your coin, and have to replace it with a fair one, i.e.

$$q(x) = 1/2, \tag{7.37}$$

but you want to study for the same amount on average. Suppose you decide to do this by importance sampling with the new coin. Now, upon flipping heads or tails, you study for

$$\begin{aligned} \frac{p(\text{heads})}{q(\text{heads})} f(\text{heads}) &= \frac{4}{3} \\ \frac{p(\text{tails})}{q(\text{tails})} f(\text{tails}) &= \frac{2}{3} \end{aligned} \tag{7.38}$$

hours respectively. Verify that your expected time spent studying is the same as before. Now compute the *variance* in the time you spend studying. Does it change?

Returning to the RL setting, we can compute the policy gradient by importance sampling from any trajectory distribution ρ. (All gradients are being taken with respect to θ.)

$$\begin{aligned} \nabla J(\theta) &= \nabla \mathop{\mathbb{E}}_{\tau \sim \rho^{\pi_\theta}}[R(\tau)] \\ &= \nabla \mathop{\mathbb{E}}_{\tau \sim \rho}\left[\frac{\rho^{\pi_\theta}(\tau)}{\rho(\tau)} R(\tau)\right] \\ &= \mathop{\mathbb{E}}_{\tau \sim \rho}\left[\frac{\nabla \rho^{\pi_\theta}(\tau)}{\rho(\tau)} R(\tau)\right]. \end{aligned}$$

Setting $\rho = \rho^{\pi_\theta}$ reveals eq. 7.30, and we can then proceed as we did in the previous proof. □

Let us reiterate some intuition into how this method works. Recall that we update our parameters according to

$$\begin{aligned} \theta_{t+1} &= \theta_t + \eta \nabla J(\theta_t) \\ &= \theta^t + \eta \mathop{\mathbb{E}}_{\tau \sim \rho^{\theta_t}}[\nabla \log \rho^{\theta^t}(\tau) R(\tau)]. \end{aligned} \tag{7.39}$$

Consider the "good" trajectories where $R(\tau)$ is large. Then θ gets updated so that these trajectories become more likely. To see why, recall that $\log \rho^\theta(\tau)$ is the log-likelihood of the trajectory τ under the policy π_θ, so the gradient points in the direction that makes τ more likely.

However, the REINFORCE gradient estimator $g^{\mathrm{R}}_\tau(\theta)$ has large variance. Intuitively, this is because it uses the total reward from the entire trajectory, which depends on the entire sequence of interactions with the environment, each step of which introducesrandomness. The rest of this chapter investigates ways to find *lower-variance* policy gradient estimators.

7.4.3 Baselines and advantages

A central idea from statistical learning is the **bias-variance decomposition**, which shows that the mean squared error of an estimator is the sum of its squared bias and its variance. All of the policy gradient estimators we will see in this chapter are already unbiased, i.e., their mean over trajectories equals the true policy gradient (def. 7.2). Can we construct estimators with lower *variance* as well?

As a first step, note that the action taken at step h does not causally affect the reward from previous timesteps, since they're already in the past. So we should only use the reward from the current timestep onwards to estimate the policy gradient.

Theorem 7.4 (Using the remaining reward is unbiased)**.** *Substituting the reamining reward* $\sum_{h'=h}^{H-1} r_{h'}$ *for* $\psi_h(\tau)$ *into the general policy gradient estimator eq. 7.23 gives an unbiased estimator. That is,*

$$\begin{aligned} \nabla J(\theta) &= \mathop{\mathbb{E}}_{\tau\sim\rho^{\pi_\theta}}[g_\tau^{rem}(\theta)] \\ \textit{where} \quad g_\tau^{rem}(\theta) &:= \sum_{h=0}^{H-1}\left(\sum_{h'=h}^{H-1} r_{h'}\right)\nabla \log \pi_\theta(a_h \mid s_h). \end{aligned} \tag{7.40}$$

Exercise 7.4 (Unbiasedness of remaining reward estimator)**.** We leave the proof of Theorem 7.4 as an exercise.

By a conditioning argument, we can replace the remaining reward with the policy's Q-function, evaluated at the current state. By the same reasoning as above, this also reduces the variance, since the only stochasticity in the expression $Q^{\pi_\theta}(s_h, a_h)$ comes from the current state and action.

Theorem 7.5 (Using the Q function is unbiased)**.** *Substituting* $Q^{\pi_\theta}(s_h, a_h)$ *for* $\psi_h(\tau)$ *into the general policy gradient estimator eq. 7.23 gives an unbiased estimator. That is,*

$$\begin{aligned} \nabla J(\theta) &= \mathop{\mathbb{E}}_{\tau\sim\rho^{\pi_\theta}}[g_\tau^{Q}(\theta)] \\ \textit{where} \quad g_\tau^{Q}(\theta) &:= \sum_{h=0}^{H-1} Q^{\pi_\theta}(s_h, a_h)\nabla \log \pi_\theta(a_h \mid s_h). \end{aligned} \tag{7.41}$$

Exercise 7.5 (Unbiasedness of remaining reward estimator)**.** We also leave the proof of Theorem 7.5 as an exercise.

We can further reduce variance by subtracting a **baseline function** $b_\theta : \mathcal{S} \to \mathbb{R}$. Note that this function could also depend on the current policy parameters.

Theorem 7.6 (Subtracting a baseline function preserves unbiasedness)**.** *Let $b_\theta : \mathcal{S} \to \mathbb{R}$ be some baseline function, and let ψ be a gradient coefficient function that yields an unbiased policy gradient estimator (e.g. eq. 7.25 or eq. 7.43). Substituting*

$$\psi_h^{bl}(\tau) := \psi_h(\tau) - b_\theta(s_h) \tag{7.42}$$

into the general policy gradient estimator eq. 7.23 gives an unbiased policy gradient estimator. That is,

$$\begin{aligned} \nabla J(\theta) &= \mathop{\mathbb{E}}_{\tau\sim\rho^{\pi_\theta}}[g_\tau^{bl}(\theta)] \\ \textit{where} \quad g_\tau^{bl}(\theta) &:= \sum_{h=0}^{H-1} (\psi_h(\tau) - b_\theta(s_h)) \nabla \log \pi_\theta(a_h \mid s_h). \end{aligned} \tag{7.43}$$

Exercise 7.6 (Unbiasedness of baseline estimator)**.** We leave the proof of Theorem 7.6 as an exercise as well.

For example, we might want b_h to estimate the average remaining reward at a given timestep:

$$b_\theta(s_h) = \mathop{\mathbb{E}}_{\tau\sim\rho^{\pi_\theta}} \left[\sum_{h'=h}^{H-1} r_{h'} \right]. \tag{7.44}$$

As a better baseline, we could instead choose the *value function* of π_θ. For any policy π, note that the random variable $Q_h^\pi(s,a) - V_h^\pi(s)$, where the randomness is taken over the action a, is centered around zero. (Recall $V_h^\pi(s) = \mathbb{E}_{a\sim\pi} Q_h^\pi(s,a)$.) This quantity matches the intuition given in sec. 7.4: it is *positive* for actions that are better than average (in state s), and *negative* for actions that are worse than average. In fact, it has a particular name: the **advantage function.**

Definition 7.4 (Advantage function)**.** For a policy π, its advantage function A^π at time h is given by

$$A_h^\pi(s,a) := Q_h^\pi(s,a) - V_h^\pi(s). \tag{7.45}$$

Note that for an optimal policy $\pi^\star$, the advantage of a given state-action pair is always zero or negative.

We can now use $A^{\pi_\theta}(s_h, a_h)$ for the gradient coefficients to obtain the ultimate unbiased policy gradient estimator.

Theorem 7.7 (Using the advnatage function is unbiased)**.** *Substituting*

$$\psi_h(\tau) := A^{\pi_\theta}(s_h, a_h) \tag{7.46}$$

into the general policy gradient estimator eq. 7.23 gives an unbiased estimator. That is,

$$\begin{aligned} \nabla J(\theta) &= \mathop{\mathbb{E}}_{\tau \sim \rho^{\pi_\theta}}[g_\tau^{adv}(\theta)] \\ \textit{where} \quad g_\tau^{adv}(\theta) &:= \sum_{h=0}^{H-1} A^{\pi_\theta}(s_h, a_h) \nabla \log \pi_\theta(a_h \mid s_h). \end{aligned} \tag{7.47}$$

Proof. This follows directly from Theorem 7.5 and Theorem 7.6. □

Note that to avoid correlations between the gradient estimator and the value estimator (i.e. baseline), we must estimate them with independently sampled trajectories:

function pg_with_learned_baseline(env : gym.Env, $\pi, \eta : \mathbb{R}, \theta_{\text{init}}, K : \mathbb{Z}, N : \mathbb{Z}$)
  $\theta \leftarrow \theta_{\text{init}}$
  for $k \in \text{range}(K)$ **do**
    trajectories $\leftarrow$ sample_trajectories(env, $\pi(\theta)$, N)
    $\widehat{V} \leftarrow$ fit_value(trajectories)
    $\tau \leftarrow$ sample_trajectories(env, $\pi(\theta)$, 1)
    $\widehat{\nabla} \leftarrow$ jnp.zeros_like(θ)
    for $(h, (s, a)) \in \text{enumerate}(\tau)$ **do**
      function log_likelihood(θ_{opt})
        return $\log \pi(\theta_{\text{opt}})\,(s, a)$
      end function
      $\widehat{\nabla} \leftarrow \widehat{\nabla} + \nabla(\text{log\_likelihood})\,(\theta) \cdot \left(\text{return\_to\_go}(\tau, h) - \widehat{V}(s)\right)$
    end for
    $\theta \leftarrow \theta + \eta \widehat{\nabla}$
  end for
  return θ
end function

Note that you could also generalize this by allowing the learning rate η to vary across steps, or take multiple trajectories τ and compute the sample mean of the gradient estimates.

The baseline estimation step `fit_value` can be done using any appropriate supervised learning algorithm. Note that the gradient estimator will be unbiased regardless of the baseline.

Example 7.5 (Policy gradient for the linear-in-features parameterization)**.** The gradient-log-likelihood for the linear parameterization ex. 7.2 is also quite elegant:

$$\begin{aligned}\nabla \log \pi_\theta(a|s) &= \nabla \left(\theta^\top \phi(s,a) - \log \left(\sum_{a'} \exp(\theta^\top \phi(s,a')) \right) \right) \\ &= \phi(s,a) - \mathop{\mathbb{E}}_{a' \sim \pi_\theta(s)} \phi(s,a')\end{aligned}$$

Plugging this into our policy gradient expression, we get

$$\begin{aligned}\nabla J(\theta) &= \mathop{\mathbb{E}}_{\tau \sim \rho^{\pi_\theta}} \left[\sum_{t=0}^{T-1} \nabla \log \pi_\theta(a_h|s_h) A_h^{\pi_\theta} \right] \\ &= \mathop{\mathbb{E}}_{\tau \sim \rho^{\pi_\theta}} \left[\sum_{t=0}^{T-1} \left(\phi(s_h,a_h) - \mathop{\mathbb{E}}_{a' \sim \pi(s_h)} \phi(s_h,a') \right) A_h^{\pi_\theta}(s_h,a_h) \right] \\ &= \mathop{\mathbb{E}}_{\tau \sim \rho^{\pi_\theta}} \left[\sum_{t=0}^{T-1} \phi(s_h,a_h) A_h^{\pi_\theta}(s_h,a_h) \right]\end{aligned}$$

Why can we drop the $\mathbb{E}\, \phi(s_h, a')$ term? By linearity of expectation, consider the dropped term at a single timestep: $\mathbb{E}_{\tau \sim \rho^{\pi_\theta}} \left[\left(\mathbb{E}_{a' \sim \pi(s_h)} \phi(s,a') \right) A_h^{\pi_\theta}(s_h,a_h) \right]$. By Adam's Law, we can wrap the advantage term in a conditional expectation on the state s_h. Then we already know that $\mathbb{E}_{a \sim \pi(s)} A_h^\pi(s,a) = 0$, and so this entire term vanishes.

7.5 Comparing policy gradient algorithms to policy iteration

What advantages do policy gradient algorithms have over the policy iteration algorithms covered in sec. 2.4.4.2?

Remark 7.7 (Policy iteration review). Recall that policy iteration is an algorithm for MDPs with unknown state transitions where we alternate between the following two steps:

- Estimating the Q-function (or advantage function) of the current policy;
- Updating the policy to be greedy with respect to this approximate Q-function (or advantage function).

To analyze the difference between them, we'll make use of the **performance difference lemma**, which provides an expression for comparing the difference between two value functions.

Theorem 7.8 (Performance difference lemma (S. Kakade & Langford, 2002, Lemma 6.1))**.** *Suppose Alice is playing a game (an MDP). Bob is spectating, and can evaluate how good an action is compared to his own strategy. (That is, Bob can compute his* advantage function $A_h^{Bob}(s_h, a_h)$*). The performance difference lemma says that Bob can now calculate exactly how much better or worse he is than Alice as follows:*

$$V_0^{Alice}(s) - V_0^{Bob}(s) = \mathop{\mathbb{E}}_{\tau \sim \rho^{Alice}} \left[\sum_{h=0}^{H-1} A_h^{Bob}(s_h, a_h) \mid s_0 = s \right] \tag{7.48}$$

where ρ^{Alice} denotes Alice's trajectory distribution (def. 2.7).

To see why, consider a specific step h in the trajectory. We compute how much better actions from Bob are than the actions from Alice, on average. But this is exactly the average Bob-advantage across actions from Alice, as described in the PDL!

Proof. Formally, this corresponds to a nice telescoping simplification when we expand out the definition of the advantage function. Note that

$$\begin{aligned} A_h^\pi(s_h, a_h) &= Q_h^\pi(s_h, a_h) - V_h^\pi(s_h) \\ &= r_h(s_h, a_h) + \mathop{\mathbb{E}}_{s_{h+1} \sim P(\cdot|s_h, a_h)} [V_{h+1}^\pi(s_{h+1})] - V_h^\pi(s_h) \end{aligned} \tag{7.49}$$

so expanding out the r.h.s. expression of eq. 7.48 and grouping terms together gives

$$\begin{aligned} \mathop{\mathbb{E}}_{\tau \sim \rho^{\text{Alice}}} \left[\sum_{h=0}^{H-1} A_h^{\text{Bob}}(s_h, a_h) \mid s_0 = s \right] &= \mathop{\mathbb{E}}_{\tau \sim \rho_{\text{Alice}}} \Bigg[\left(\sum_{h=0}^{H-1} r_h(s_h, a_h) \right) \\ &\quad + (V_1^{\text{Bob}}(s_1) + \cdots + V_H^{\text{Bob}}(s_H)) \\ &\quad - (V_0^{\text{Bob}}(s_0) + \cdots + V_{H-1}^{\text{Bob}}(s_{H-1})) \mid s_0 = s \Bigg] \\ &= V_0^{\text{Alice}}(s) - V_0^{\text{Bob}}(s). \end{aligned} \tag{7.50}$$

as desired. Note that the "inner" expectation from expanding the advantage function has the same distribution as the outer one, so omitting it here is valid. Also note that V_H^π, the value after reaching a terminal state, is always zero for any policy π. □

The PDL gives insight into why fitted approaches such as PI don't work as well in the "full" RL setting. To see why, let's consider a single iteration of policy iteration, where policy π gets updated to $\tilde{\pi}$. We'll assume these policies are deterministic. Define Δ_∞ to be the most negative advantage:

$$\Delta_\infty = \min_{s \in \mathcal{S}} A_h^\pi(s, \tilde{\pi}(s)). \tag{7.51}$$

Suppose $\Delta_\infty < 0$, i.e. there exists a state s such that

$$A^\pi(s, \tilde{\pi}(s)) < 0, \tag{7.52}$$

that is, if $\tilde{\pi}$ acts for just one turn from state s and then π acts thereafter, the result would be worse on average than allowing π to act. Plugging this into the PDL (Theorem 7.8) gives

$$\begin{aligned} V_0^{\tilde{\pi}}(s) - V_0^\pi(s) &= \mathop{\mathbb{E}}_{\tau \sim \rho^{\tilde{\pi}}} \left[\sum_{h=0}^{H-1} A_h^\pi(s_h, a_h) \mid s_0 = s \right] \\ &\geq H\Delta_\infty \\ V_0^{\tilde{\pi}}(s) &\geq V_0^\pi(s) - H|\Delta_\infty|. \end{aligned}$$

That is, for some state s, the lower bound on the performance of $\tilde{\pi}$ is *lower* than the performance of π. This doesn't state that $\tilde{\pi}$ *will* necessarily perform worse than π, only suggests that it might be possible. If these worst case states do exist, though, PI does not avoid situations where the new policy often visits them; It does not enforce that the trajectory distributions ρ^π and $\rho^{\tilde{\pi}}$ be close to each other. In other words, PI falls prey to **distributional shift**: the "training distribution" that our prediction rule is fitted on, ρ^π, may differ significantly from the "evaluation distribution" $\rho^{\tilde{\pi}}$.

On the other hand, policy gradient methods *do*, albeit implicitly, encourage ρ^π and $\rho^{\tilde{\pi}}$ to be similar. Suppose that the mapping from policy parameters to trajectory distributions is relatively smooth. Then, by adjusting the parameters only a small distance, the new policy will also have a similar trajectory distribution. But this is not very rigorous, and in practice the parameter-to-distribution mapping may not be so smooth. Can we constrain the distance between the resulting distributions more *explicitly*?

This brings us to the following local policy optimization methods:

1. **trust region policy optimization** (TRPO), which explicitly constrains the difference between the distributions before and after each step;
2. the **natural policy gradient** (NPG), a first-order approximation of TRPO;
3. **proximal policy optimization** (PPO-penalty), a "soft relaxation" of TRPO;
4. the **clipped surrogate objective** (PPO-clip), a version of PPO that is popular in practice.

Remark 7.8 (Ordering of algorithms). Chronologically, NPG was developed first, followed by TRPO and later PPO. We begin with TRPO since it sets up the intuition behind these constrained methods.

7.6 Trust region policy optimization

We saw above that policy gradient methods are effective because they implicitly constrain how much the policy changes at each iteration in terms of its trajectory distribution ρ^π. What happens if we *explicitly* constrain the distance between the new and old trajectory distributions? This requires some way to measure the *distance* between two trajectory distributions. For this, we introduce the **Kullback-Leibler divergence**.

Definition 7.5 (Kullback-Leibler divergence)**.** For two PDFs p, q,

$$\mathrm{KL}\left(p \parallel q\right) := \mathop{\mathbb{E}}_{x\sim p}\left[\log \frac{p(x)}{q(x)}\right]. \tag{7.53}$$

The Kullback-Leibler divergence can be interpreted in many different ways, many stemming from information theory. One such interpretation is that $\mathrm{KL}\left(p \parallel q\right)$ describes how much more surprised you are if you *think* data is being generated by q but it's actually generated by p, compared to someone who knows the true distribution p. (The **surprise** of an event with probability p is $-\log_2 p$.)

It can be shown that $\mathrm{KL}\left(p \parallel q\right) = 0$ if and only if $p = q$. Also note that it is generally *not* symmetric, that is, $\mathrm{KL}\left(p \parallel q\right) \neq \mathrm{KL}\left(q \parallel p\right)$. How can we interpret this asymmetry?

Remark 7.9 (Asymmetry of the Kullback-Leibler divergence)*.* Note that the KL divergence gets large if $p(x)/q(x)$ is very large for some x, that is, q assigns low probability to a common event under p. So if we minimize the KL divergence with respect to the second argument q, the "prediction" distribution, q will "spread out" to cover all common events under p. If we minimize the KL divergence with respect to the first argument p, the data generating distribution, p will "squeeze under" q, so that $p(x)$ is small wherever $q(x)$ is small.

For trajectory distributions $\rho^\pi_{\theta^i}$ and $\rho^\pi_{\theta'}$ (def. 2.7), the KL divergence can be broken down into a sum over timesteps:

$$\begin{aligned}
\mathrm{KL}\left(\rho^{\pi_{\theta^i}} \parallel \rho^{\pi_{\theta'}}\right) &= \mathop{\mathbb{E}}_{\tau\sim\rho^{\pi_{\theta^i}}}\left[\log \rho^{\pi_{\theta^i}}(\tau) - \log \rho^{\pi_{\theta'}}(\tau)\right] \\
&= \mathop{\mathbb{E}}_{\tau\sim\rho^{\pi_{\theta^i}}}\left[\sum_{h=0}^{H-1} \log \pi_{\theta^i}(a_h \mid s_h) - \log \pi_{\theta'}(a_h \mid s_h)\right]
\end{aligned} \tag{7.54}$$

since the terms corresponding to the state transitions and initial state distribution cancel out.

We can now use the KL divergence to explicitly constrain the distance between the new and old trajectory distributions:

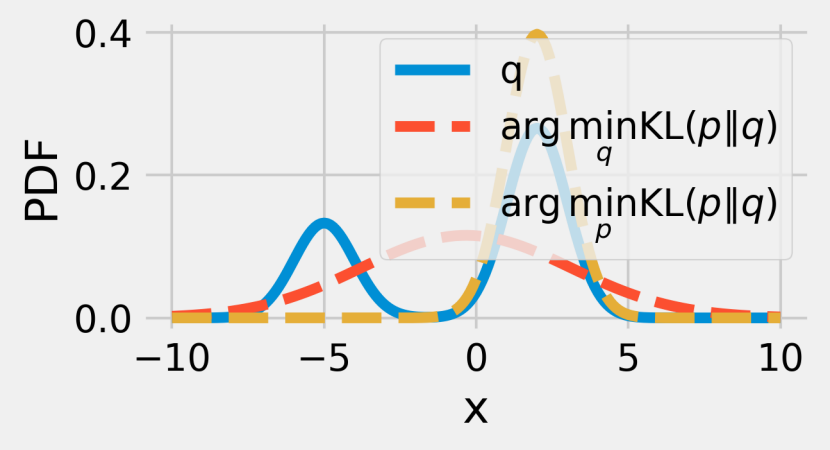


Figure 7.6: Minimizing the forward and backward Kullback-Leibler divergence against a bimodal distribution with respect to a unimodal distribution

$$\begin{aligned} \theta^{t+1} &\leftarrow \arg\max_{\theta' \in \mathbb{R}^D} J(\theta') \\ &\text{where } \mathrm{KL}\left(\rho^{\pi_{\theta^i}} \,\|\, \rho^{\pi_{\theta'}}\right) < \delta \end{aligned} \tag{7.55}$$

Note that we place $\rho^{\pi}_{\theta'}$ in the *second* argument to the KL divergence. This ensures that $\rho^{\pi}_{\theta'}$ supports all of the trajectories under $\rho^{\pi}_{\theta^i}$ (see Remark 7.9).

In place of J, if we use the performance difference lemma (Theorem 7.8) to compare the performance of the new policy to the old one, we obtain an

Definition 7.6 (TRPO update rule)**.** Let $\theta^i \in \mathbb{R}^D$ denote the current policy parameter vector. The TRPO update rule is

$$\begin{aligned} \theta^{k+1} &\leftarrow \arg\max_{\theta' \in \mathbb{R}^D} \mathop{\mathbb{E}}_{\tau \sim \rho^{\pi_{\theta^i}}} \left[\sum_{h=0}^{H-1} \mathop{\mathbb{E}}_{a'_h \sim \pi^{\theta'}(\cdot|s_h)} A^{\pi_{\theta^i}}(s_h, a'_h) \right] \\ &\text{where } \mathrm{KL}\left(\rho^{\pi_{\theta^i}} \,\|\, \rho^{\pi_{\theta'}}\right) < \delta \end{aligned} \tag{7.56}$$

Remark 7.10 (Drawing states from old policy). Note that we have made a small change to the eq. 7.48: we use the *current* policy's trajectory distribution, and re-sample *actions* from the *updated* policy. This allows us to reuse a single batch of trajectories from π_{θ^i} rather than sample new batches from $\pi_{\theta'}$ when solving the optimization problem eq. 7.56. This approximation also matches the r.h.s. of the PDL to first order in θ. (We will elaborate more on this later.)

The above isn't entirely complete: we still need to solve the actual optimization problem at each step. Unless we know additional properties of the problem, this is still a nonconvex

```
function trpo(env, δ, θ_init, n_interactions)
  θ ← θ_init
  for k ∈ range(K) do
    trajectories ← sample_trajectories(env, π(θ), n_interactions)
    Â ← fit_advantage(trajectories)
    function approximate_gain(θ_opt)
      A_total ← 0
      for τ ∈ trajectories do
        for (s, _a, _r) ∈ τ do
          for a ∈ env.action_space do
            A_total ← A_total + π(θ)(s, a) · Â(s, a)
          end for
        end for
      end for
      return A_total
    end function
    function constraint(θ_opt)
      return kl_div_trajectories(π, θ, θ_opt, trajectories) ≤ δ
    end function
    θ ← optimize(approximate_gain, constraint)
  end for
  return θ
end function
```

Figure 7.7: Pseudocode for trust region policy optimization

constrained optimization problem that might be inefficient to solve. Do we need to solve for the exact objective function, though? Instead, if we assume that both the objective function and the constraint are somewhat smooth in terms of the policy parameters, we can use their *Taylor expansions* to give us a simpler optimization problem with a closed-form solution. This brings us to the **natural policy gradient** algorithm.

7.7 Natural policy gradient

In some sense, the *natural policy gradient* algorithm (S. M. Kakade, 2001) is an *implementation* of trust region policy optimization (fig. 7.7). Recall that in each TRPO update, we seek to maximize the expected total reward while keeping the updated policy close to the current policy in terms of Kullback-Leibler divergence (def. 7.5):

$$\begin{aligned} &\theta^{k+1} \leftarrow \arg\max_{\theta' \in \mathbb{R}^D} J(\theta') \\ &\text{where } \mathrm{KL}\left(\theta^i \parallel \theta'\right) \leq \delta \end{aligned} \tag{7.57}$$

NPG uses the following simplification: we take local approximations of the objective and constraint functions, which results in a simple problem with a closed-form solution.

Concretely, we take a first-order Taylor approximation to the objective function about the current iterate θ^i:

$$J(\theta') = J(\theta^i) + (\theta' - \theta^i)^\top \nabla J(\theta^i) + O(\|\theta' - \theta^i\|^2). \tag{7.58}$$

We also take a second-order Taylor approximation to the constraint function:

Theorem 7.9 (Quadratic approximation to KL divergence.)**.** *The second-order approximation to* $d^i(\theta) := \mathrm{KL}\left(p_{\theta^i} \parallel p_\theta\right)$ *is given by*

$$d^i(\theta) = \frac{1}{2}(\theta - \theta^i)^\top F_{\theta^i}(\theta - \theta^i) + O(\|\theta - \theta^i\|^3), \tag{7.59}$$

where F_{θ^i} *is the* ***Fisher information matrix*** *(FIM) of the trajectory distribution* $\rho^{\pi_{\theta^i}}$*. (We define the FIM below in def. 7.7.)*

Proof. We leave the details as an exercise. Here is an outline:

1. Write down the Taylor expansion of $d^i(\theta)$ around θ^i.
2. Show that the zeroth-order term $d^i(\theta^i)$ is zero.
3. Show that the gradient $\nabla d^i(\theta)|_{\theta=\theta^i}$ is zero.
4. Show that the Hessian $\nabla^2 d^i(\theta)|_{\theta=\theta^i}$ equals F_{θ^i}.

□

Definition 7.7 (Fisher information matrix)**.** Let p_θ denote a distribution parameterized by θ. Its Fisher information matrix F_θ can be defined equivalently as:

$$\begin{aligned} F_\theta &= \mathop{\mathbb{E}}_{x\sim p_\theta}[(\nabla_\theta \log p_\theta(x))(\nabla_\theta \log p_\theta(x))^\top] \\ &= \mathop{\mathbb{E}}_{x\sim p_\theta}[-\nabla_\theta^2 \log p_\theta(x)]. \end{aligned} \tag{7.60}$$

Remark 7.11 (Interpretation of the Fisher information matrix)*.* The Fisher information matrix is an important quantity when working with parameterized distributions. It has many possible interpretations. The first expression in eq. 7.60 shows that it is the covariance matrix of the gradient-log-probability (also known as the **score**), and the second expression shows that it is the expected Hessian matrix of the negative-log-likelihood of the underlying distribution. It can also be shown that it is precisely the Hessian of the KL divergence (with respect to either one of the parameters).

Recall that the Hessian of a function describes its *curvature*. Concretely, suppose we have some parameter vector $\theta \in \mathbb{R}^D$ and we seek to measure how much p_θ changes if we shift θ by $\delta \in \mathbb{R}^D$. The quantity $\delta^\top F_\theta \delta$ describes how rapidly the negative log-likelihood changes if we move by δ. (The zeroth and first order terms are both zero in our case due to properties of the KL divergence.)

Putting this together results in the following update rule:

Definition 7.8 (Natural policy gradient update)**.** We aim to solve the constrained optimization problem eq. 7.57. Upon taking first- and second-order approximations of the objective function and constraint function respectively, we obtain the update rule

$$\begin{aligned} \theta^{k+1} &\leftarrow \theta^i + \arg\max_{\Delta\theta\in\mathbb{R}^D} \nabla J(\theta^i)^\top \Delta\theta \\ &\text{where } \frac{1}{2}\Delta\theta^\top F_{\theta^i}\Delta\theta \le \delta, \end{aligned} \tag{7.61}$$

where F_{θ^i} is the Fisher information matrix of $\rho^\pi_{\theta^i}$.

eq. 7.61 is a convex optimization problem with a closed-form solution. To see why, it helps to visualize the case where θ is two-dimensional: the constraint describes the inside of an ellipse, and the objective function is linear, so we can find the extreme point on the boundary of the ellipse by setting the gradient of the Lagrangian to zero:

$$\begin{aligned}
\mathcal{L}(\theta, \alpha) &= \nabla J(\theta^i)^\top (\theta - \theta^i) - \alpha \left[\frac{1}{2} (\theta - \theta^i)^\top F_{\theta^i} (\theta - \theta^i) - \delta \right] \\
\nabla \mathcal{L}(\theta^{k+1}, \alpha) &:= 0 \\
\implies \nabla J(\theta^i) &= \alpha F_{\theta^i} (\theta^{k+1} - \theta^i)
\end{aligned} \tag{7.62}$$

Rearranging gives the NPG update rule:

$$\begin{aligned}
\theta^{k+1} &= \theta^i + \eta F_{\theta^i}^{-1} \nabla J(\theta^i) \\
\text{where } \eta &= \sqrt{\frac{2\delta}{\nabla J(\theta^i)^\top F_{\theta^i}^{-1} \nabla J(\theta^i)}}
\end{aligned} \tag{7.63}$$

This gives us the closed-form update.

Remark 7.12 (Scalability of NPG). Having a closed-form solution might at first seem like brilliant news. Is there a catch? The challenge lies in computing the inverse Fisher information matrix (FIM) $F_{\theta^i}^{-1}$. Since it is an expectation over trajectories, computing it exactly is typically intractable. Instead, we could collect trajectories from the environment and approximate the expectation by a sample mean, since we can write the Fisher information matrix as

$$F_\theta = \mathop{\mathbb{E}}_{\tau \sim \rho^{\pi_\theta}} \left[\sum_{h=0}^{H-1} (\nabla \log \pi_\theta(a_h \mid s_h))(\nabla \log \pi_\theta(a_h \mid s_h))^\top \right]. \tag{7.64}$$

Note that we've used the Markov property to cancel out the cross terms corresponding to two different time steps.

It turns out that to estimate the FIM to a relative error of ϵ, we need $O(D/\epsilon^2)$ samples (Vershynin, 2018, Remark 4.7.2). In order for the estimated FIM to be accurate enough to be useful, this can be too large to be practical. Taking the inverse also takes $O(D^3)$ time, which can be expensive if the parameter space is large.

Remark 7.13 (NPG accounts for curvature in the parameter space). Let us compare the original policy gradient update (eq. 7.17) to the NPG update (eq. 7.63):

$$\begin{aligned}
\theta^{t+1} &= \theta^t + \eta \nabla J(\theta^t) && \text{Policy gradient} \\
\theta^{t+1} &= \theta^t + \eta F_{\theta^t}^{-1} \nabla J(\theta^t) && \text{Natural policy gradient}
\end{aligned} \tag{7.65}$$

The NPG update **preconditions** the gradient by the inverse Fisher information matrix. Speaking abstractly, this matrix accounts for the **geometry of the parameter space** in the following sense.

The typical gradient descent algorithm implicitly measures distances between parameters using the typical "flat" *Euclidean distance*:

$$\text{distance}(\theta_0, \theta_1) = \|\theta_0 - \theta_1\|. \tag{7.66}$$

The NPG update measures distance between parameters as the KL divergence (def. 7.5) between their induced trajectory distributions:

$$\text{distance}(\theta_0, \theta_1) = \text{KL}\left(\rho^{\pi_{\theta_0}} \parallel \rho^{\pi_{\theta_1}}\right). \tag{7.67}$$

Using a parameterized policy class is just a means to the end of optimizing the trajectory distribution, so it makes sense to optimize over the trajectory distributions directly. In fact, the NPG update is the only update that is invariant to reparameterizations of the policy space: if instead of θ, we used some transformation $\phi(\theta)$ to parameterize the policy, the NPG update would remain the same. This is why NPG is called a **coordinate-free** optimization algorithm.

We can illustrate this with the following example:

Example 7.6 (Natural gradient on a simple problem)**.** Let's step away from RL and consider a simple optimization problem over Bernoulli distributions $p_\theta \in \triangle(\{0, 1\})$. This distribution space is parameterized by the success probability $\theta \in [0, 1]$. The per-sample objective function is 100 if the Bernoulli trial succeeds and 1 if the trial fails.

$$f(x) = \begin{cases} 100 & x = 1 \\ 1 & x = 0. \end{cases} \tag{7.68}$$

The objective $J(\theta)$ is the average of f over $x \sim p_\theta$:

$$\begin{aligned} J(\theta) &= \mathop{\mathbb{E}}_{x \sim p_\theta}[f(x)] \\ &= 100\theta + 1(1 - \theta) \end{aligned} \tag{7.69}$$

We can think of the space of such distributions as the line between $(0, 1)$ to $(1, 0)$ on the Cartesian plane:

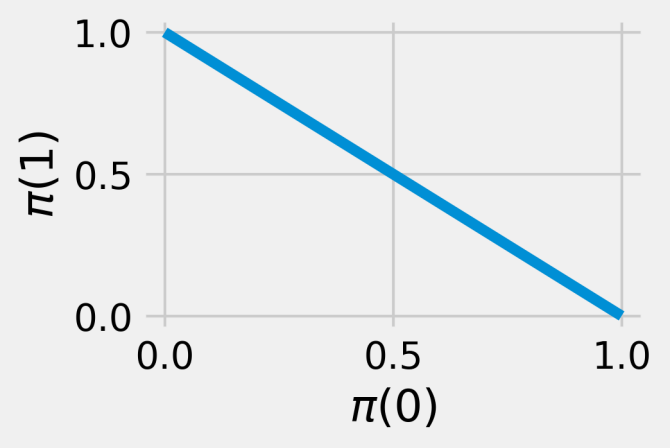


Figure 7.8: The space of Bernoulli distributions.

Clearly the optimal distribution is the constant one $\pi(1) = 1$. Suppose we optimize over the parameterized family $\pi_\theta(1) = \frac{\exp(\theta)}{1+\exp(\theta)}$. Then our optimization algorithm should set θ to be unboundedly large. Then the "parameter-space" gradient is

$$\nabla_\theta J(\pi_\theta) = \frac{99\exp(\theta)}{(1+\exp(\theta))^2}.$$

Note that as $\theta \to \infty$ that the increments get closer and closer to 0; the rate of increase becomes exponentially slow.

However, if we compute the Fisher information "matrix" (which is just a scalar in this case), we can account for the geometry induced by the parameterization.

$$\begin{aligned} F_\theta &= \mathop{\mathbb{E}}_{x\sim\pi_\theta}[(\nabla_\theta \log \pi_\theta(x))^2] \\ &= \frac{\exp(\theta)}{(1+\exp(\theta))^2}. \end{aligned}$$

This gives the natural gradient update

$$\begin{aligned} \theta^{k+1} &= \theta^i + \eta F_{\theta^i}^{-1} \nabla_\theta J(\theta^i) \\ &= \theta^i + 99\eta \end{aligned}$$

which increases at a constant rate, i.e. improves the objective more quickly than parameter-space gradient descent.

Though the NPG now gives a closed-form optimization step, it requires estimating and computing the inverse Fisher information matrix, which can be difficult or slow, especially as the parameter space gets large (Remark 7.12). Can we achieve a similar effect without the inverse Fisher information matrix? This brings us to the **proximal policy optimization** algorithm.

7.8 Penalty-based proximal policy optimization

We can relax the TRPO optimization problem (eq. 7.56) in a different way: Rather than imposing a *hard constraint*

$$\mathrm{KL}\left(\rho^{\pi_{\theta^i}} \,\|\, \rho^{\pi_{\theta'}}\right) < \delta, \tag{7.70}$$

we can instead impose a *soft* constraint by subtracting $\lambda \mathrm{KL}\left(\rho^{\pi_{\theta^i}} \,\|\, \rho^{\pi_{\theta'}}\right)$ from the function to be maximized. $\lambda > 0$ is a coefficient that controls the tradeoff between the two terms. This gives the following objective (Schulman et al., 2017):

$$\theta^{k+1} \leftarrow \arg\max_{\theta'} \mathop{\mathbb{E}}_{\tau\sim\rho^{\pi_{\theta^i}}} \left[\sum_{h=0}^{H-1} \mathop{\mathbb{E}}_{a'_h\sim\pi_\theta(s_h)} A^{\pi_{\theta^i}}(s_h, a'_h)\right] - \lambda \mathrm{KL}\left(\rho^{\pi_{\theta^i}} \,\|\, \rho^{\pi_{\theta'}}\right) \tag{7.71}$$

This optimization problem is also known as the **Lagrangian** formulation of eq. 7.56.

How do we solve this optimization? Let us begin by simplifying the $\mathrm{KL}\left(\rho^{\pi_{\theta^i}} \,\|\, \rho^{\pi_\theta}\right)$ term. The state transitions cancel as in eq. 7.54, which gives us

$$\begin{aligned}
\mathrm{KL}\left(\rho^{\pi_{\theta^i}} \,\|\, \rho^{\pi_\theta}\right) &= \mathop{\mathbb{E}}_{\tau\sim\rho^{\pi_{\theta^i}}} \left[\log \frac{\rho^{\pi_{\theta^i}}(\tau)}{\rho^{\pi_\theta}(\tau)}\right] \\
&= \mathop{\mathbb{E}}_{\tau\sim\rho^{\pi_{\theta^i}}} \left[\sum_{h=0}^{H-1} \log \frac{\pi_{\theta^i}(a_h \mid s_h)}{\pi_\theta(a_h \mid s_h)}\right] \\
&= \mathop{\mathbb{E}}_{\tau\sim\rho^{\pi_{\theta^i}}} \left[\sum_{h=0}^{H-1} \log \frac{1}{\pi_\theta(a_h \mid s_h)}\right] + c
\end{aligned}$$

where c doesn't depend on θ and can be ignored. This gives the objective

$$L^k(\theta) = \mathop{\mathbb{E}}_{s_0,\ldots,s_{H-1}\sim\rho^{\pi_{\theta^i}}} \left[\sum_{h=0}^{H-1} \mathop{\mathbb{E}}_{a_h\sim\pi_\theta(s_h)} A^{\pi_{\theta^i}}(s_h, a_h)\right] - \lambda \mathop{\mathbb{E}}_{\tau\sim\rho^{\pi_{\theta^i}}} \left[\sum_{h=0}^{H-1} \log \frac{1}{\pi_\theta(a_h \mid s_h)}\right]$$

Once again, this takes an expectation over trajectories. But here we cannot directly sample trajectories from π_{θ^i}, since in the first term, the actions actually come from π_θ. To make

this term line up with the other expectation, we would need the actions to also come from π_{θ^i}. This should sound familiar: we want to estimate an expectation over one distribution by sampling from another. We can use importance sampling (Theorem 7.3) to rewrite the inner expectation:

$$\mathop{\mathbb{E}}_{a_h \sim \pi_{\theta'}(s_h)} A^{\pi_{\theta^i}}(s_h, a_h) = \mathop{\mathbb{E}}_{a_h \sim \pi_{\theta^i}(s_h)} \left[\frac{\pi_{\theta'}(a_h \mid s_h)}{\pi_{\theta^i}(a_h \mid s_h)} A^{\pi_{\theta^i}}(s_h, a_h) \right] \tag{7.72}$$

Remark 7.14 (Interpretation of likelihood ratio). Suppose a^+ is a "good" action for π_{θ^i} in state s_h, i.e. $A^{\pi}_{\theta^i}(s_h, a^+) > 0$. Then maximizing eq. 7.72 encourages θ' to *increase* the probability ratio $\pi_{\theta'}(a_h \mid s_h)/\pi_{\theta^i}(a_h \mid s_h)$.

Otherwise, if a^- is a "bad" action for π_{θ^i} in state s_h (i.e. $A^{\pi}_{\theta^i}(s_h, a^-) < 0$), then maximizing eq. 7.72 encourages θ' to *decrease* the probability ratio $\pi_{\theta'}(a_h \mid s_h)/\pi_{\theta^i}(a_h \mid s_h)$.

Now we can combine the expectations together to get the objective

Definition 7.9 (Penalty objective)**.**

$$L^k(\theta) = \mathop{\mathbb{E}}_{\tau \sim \rho^{\pi_{\theta^i}}} \left[\sum_{h=0}^{H-1} \left(\frac{\pi_\theta(a_h \mid s_h)}{\pi_{\theta^i}(a_h \mid s_h)} A^{\pi_{\theta^i}}(s_h, a_h) - \lambda \log \frac{1}{\pi_\theta(a_h \mid s_h)} \right) \right]$$

Now we can estimate this function by a sample mean over trajectories from π_{θ^i}. Remember that to complete a single iteration of PPO, we execute

$$\theta^{k+1} \leftarrow \arg\max_\theta L^k(\theta).$$

If L^k is differentiable, we can optimize it by gradient descent, completing a single iteration of PPO.

```
function ppo_proximal(env, π : (ℝ^D) → (State × Action) → ℝ, λ : ℝ, θ_init : ℝ^D, n_iters : ℤ, n_fit_trajectories : ℤ
  θ ← θ_init
  for k ∈ range(n_iters) do
    fit_trajectories ← sample_trajectories(env, π(θ), n_fit_trajectories)
    Â ← fit(fit_trajectories)
    sample_trajectories ← sample_trajectories(env, π(θ), n_sample_trajectories)
    function objective(θ_opt)
      total_objective ← 0
      for τ ∈ sample_trajectories do
        for (s, a, _r) ∈ τ do
          total_objective ← total_objective + (π(θ_opt)(s,a) / π(θ)(s,a)) Â(s, a) + λ · log π(θ_opt)(s, a)
        end for
      end for
      return total_objective / n_sample_trajectories
    end function
    θ ← optimize(objective, θ)
  end for
  return θ
end function
```

7.9 Advantage clipping

Recall that the main point of proximal policy optimization methods (TRPO, NPG, PPO) is to encourage the updated policy (after taking a gradient step) to remain similar to the current one. These methods used the KL divergence to measure the distance between policies. Schulman et al. (2017) proposed an alternative way to constrain the step size based on *clipping* a certain objective function. This method, known as "PPO-clip" or the "clipped surrogate" objective function, is the most widely used proximal policy optimization algorithm in practice.

Above, in eq. 7.72, we constructed an objective function by applying importance sampling to the performance difference lemma (Theorem 7.8). Without the KL divergence penalty, this becomes

$$L^k(\theta') = \mathop{\mathbb{E}}_{\tau\sim\rho^{\pi_{\theta^i}}}\left[\sum_{h=0}^{H-1} \frac{\pi_{\theta'}(a_h \mid s_h)}{\pi_{\theta^i}(a_h \mid s_h)} A^{\pi_{\theta^i}}(s_h, a_h)\right] \approx V(\pi_{\theta'}) - V(\pi_{\theta^i}). \tag{7.73}$$

In the following part, define the policy ratio at time $h \in [H]$ as

$$\Lambda_h(\theta', \theta^i) = \frac{\pi_{\theta'}(a_h \mid s_h)}{\pi_{\theta^i}(a_h \mid s_h)}. \tag{7.74}$$

The clipped surrogate objective function modifies eq. 7.73 to remove incentives for $\pi_{\theta'}$ to differ greatly from π_{θ^i}. Specifically, we choose some small $\epsilon > 0$, and constrain $\Lambda_h(\theta', \theta^i) \in (1-\epsilon, 1+\epsilon)$. Formally,

$$\Lambda_h^{\text{clipped}}(\theta', \theta^i) = \text{clip}(\Lambda_h(\theta', \theta^i), 1-\epsilon, 1+\epsilon), \tag{7.75}$$

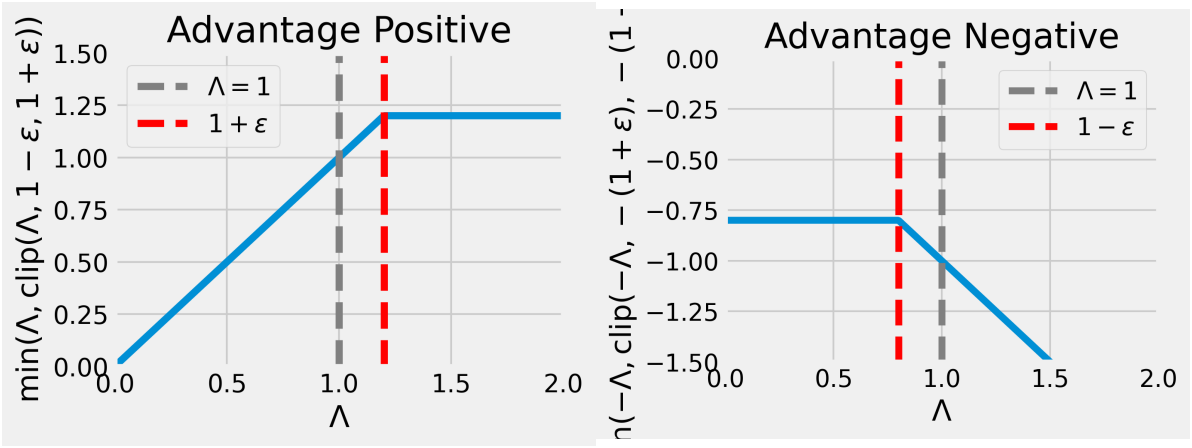
where

$$\text{clip}(x, a, b) := \max\{a, \min\{x, b\}\}. \tag{7.76}$$

Remark 7.15 (Interpretation). As in Remark 7.14, suppose a_h^+ and a_h^- are a "good" and "bad" action in s_h, i.e. $A^\pi_{\theta^i}(s_h, a_h^+) > 0$ and $A^\pi_{\theta^i}(s_h, a_h^-) < 0$. By clipping $\Lambda_h(\theta', \theta^i)$, no additional benefit is gained by increasing $\pi_{\theta'}(a_h^+ \mid s_h)$ above $(1+\epsilon)\pi_{\theta^i}(a_h^+ \mid s_h)$, or by decreasing $\pi_{\theta^i}(a_h^- \mid s_h)$ under $(1-\epsilon)\pi_{\theta^i}(a_h^- \mid s_h)$.

As a final step, we only use this clipped objective if it is smaller than the original objective. For example, if $A^\pi_{\theta^i}(s_h, a_h) > 0$ and $\Lambda_h(\theta', \theta^i) \ll 1-\epsilon$, then the clipped objective would disproportionately incentivize taking action a_h. The same thing happens if $A^\pi_{\theta^i}(s_h, a_h) < 0$ and $\Lambda_h(\theta', \theta^i) > 1+\epsilon$.

Putting these together, this results in only clipping the policy ratio on the side that it would normally move towards:



(a) For a "good" action a_h^+ with positive advantage. (b) For a "poor" action a_h^- with negative advantage.

Figure 7.9: Clipped surrogate objective function

Definition 7.10 (Clipped surrogate objective)**.** The PPO-Clip objective is

$$L^i(\theta) = \mathop{\mathbb{E}}_{\tau \sim \rho^{\pi_{\theta^i}}} \left[\sum_{h=0}^{H-1} \min \left\{ \Lambda_h(\theta, \theta^i) A^{\pi_{\theta^i}}(s_h, a_h), \text{clip}(\Lambda_h(\theta, \theta^i), 1-\epsilon, 1+\epsilon) A^{\pi}_{\theta^i}(s_h, a_h) \right\} \right] \tag{7.77}$$

Remark 7.16 (Convergence of the clipped surrogate objective). From a traditional optimization perspective, this objective function seems challenging to analyze: the clip operation (eq. 7.76) is not differentiable

7.10 Key takeaways

Policy gradient methods are a powerful family of algorithms that optimize the expected total reward by iteratively updating the policy parameters. Precisely, we estimate the gradient of the expected total reward (with respect to the parameters), and update the parameters in that direction. But estimating the gradient is a tricky task! We saw many ways to reduce the variance of the gradient estimator, culminating in the advantage-based expression eq. 7.47.

But updating the parameters doesn't entirely solve the problem: Sometimes, a small step in the parameters might lead to a big step in the policy. To avoid changing the policy too much at each step, we must account for the curvature in the parameter space. We first did this explicitly with sec. 7.6, and then saw ways to relax the constraint via the natural policy gradient update (sec. 7.7), the PPO-KL update (sec. 7.8), and the PPO-Clip update (sec. 7.9).

These are still popular methods to this day, especially because they efficiently integrate with *deep neural networks* for representing complex functions.

7.11 Bibliographic notes and further reading

Sutton et al. (1999) popularized the term "policy gradient methods". Policy gradient methods, including REINFORCE (Williams, 1992), were some of the earliest reinforcement learning methods (Barto et al., 1983; Sutton, 1984; Witten, 1977).

S. Kakade & Langford (2002) introduced the performance difference lemma, which was used to prove theoretical guarantees in several later works.

The expression of the policy gradient in terms of the Q-function was simultaneously observed by Marbach & Tsitsiklis (2001) and Sutton et al. (1999).

The natural gradient was suggested by Amari (1998) for general optimization problems and then applied to the policy gradient by S. M. Kakade (2001). The **Natural Actor-Critic**

algorithm combines actor-critic learning with the natural policy gradient estimator (Bhatnagar et al., 2009; Peters et al., 2005; Peters & Schaal, 2008).

Natural gradient descent is effective because it accounts for the *geometry* induced by the parameterization and thereby optimizes over the *distributions* themselves. Another optimization algorithm that accounts for parameter-space geometry is **mirror descent** (Beck & Teboulle, 2003; Nemirovskij et al., 1983). At each optimization step, this maps the parameter vector to a *dual space* defined to capture the desired geometry. It then takes a gradient descent step on the dual parameters and maps the result back to an updated "primal" parameter vector. One can also apply mirror descent to policy gradient algorithms (Mahadevan et al., 2013; Mahadevan & Liu, 2012).

A key reason for the popularity of policy gradient methods is that they are easily *parallelizable* across multiple computing threads. Each thread collects rollouts and computes its own gradient estimates, and every once in a while, these are accumulated across threads and used to perform an update of the parameters, which are then synchronized across the different threads. Mnih et al. (2016) present multi-threaded variants of policy gradient methods as well as several algorithms from Chapter 6. Schulman et al. (2015) proposed TRPO (def. 7.6). Schulman et al. (2016) introduced **generalized advantage estimation** (GAE), which accepts some bias in the gradient estimate in exchange for lower variance. This tradeoff is controlled by the parameter λ, the same parameter as in TD(λ). Schulman et al. (2017) introduced the PPO-KL and PPO-Clip algorithms. Wu et al. (2017) introduced the ACKTR algorithm, which seeks to scale NPG up to deep learning models where naively approximating the Fisher information matrix is impractical. It does so using the Kronecker-factored approximation proposed in Martens & Grosse (2015). Shao et al. (2024) introduce a variant of PPO known as **Group Relative Policy Optimization** (GRPO).

Policy gradient methods, being on-policy algorithms, inherently have worse sample efficiency than off-policy algorithms, since naively each sample can only be used to update at the value of the parameter vector that it was sampled at. Wang et al. (2017) introduce the **actor-critic with experience replay** (ACER) algorithm, which builds on the **Retrace** Q function estimator (Munos et al., 2016) to estimate policy gradients using previously collected experience (Lin, 1992).

The **soft Q-learning** and **soft actor-critic** algorithms introduced in Haarnoja et al. (2017) and Haarnoja et al. (2018) respectively add the **entropy** of the policy to the objective function. This practice is known as **maximum entropy reinforcement learning** since we seek to maximize the policy entropy alongside the expected total reward. This encourages policies that "act more randomly", which aids exploration.

8 Imitation Learning

8.1 Introduction

Imagine you are tasked with learning how to drive. How do, or did, you go about it? At first, this task might seem insurmountable: there are a vast array of controls, and the cost of making a single mistake could be extremely high, making it hard to explore by trial and error. Luckily, there are already people in the world who know how to drive who can get you started. In almost every challenge we face, we "stand on the shoulders of giants" and learn skills from experts who have already mastered them.

Figure 8.1: A robot imitating the pose of a young child. Image from Danilyuk (2021).

In machine learning, we often try to teach machines to accomplish tasks that humans are already proficient at. In such cases, the machine learning algorithm is the one learning the new skill, and humans are the "experts" that can demonstrate how to perform the task. **Imitation learning** is an approach to sequential decision-making where we aim to learn a policy that performs at least as well as the expert. It is often used as a first step for complex tasks where it is too challenging to learn from scratch or difficult to specify a reward function that captures the desired behaviour.

We'll see that the most naive form of imitation learning, called **behaviour cloning**, is really an application of supervised learning to interactive tasks. We'll then explore **dataset aggregation** (DAgger) as a way to query an expert and learn even more effectively.

8.2 Behaviour cloning

This notion of "learning from human-provided data" may remind you of the basic premise of Chapter 5. In supervised learning, there is some mapping from *inputs* to *outputs*, such as the task of assigning the correct label to an image, that humans can implicitly compute. To teach a machine to calculate this mapping, we first collect a large *training dataset* by getting people to label a lot of inputs, and then use some optimization algorithm to produce a predictor that maps from the inputs to the outputs as closely as possible.

How does this relate to interactive tasks? Here, the input is the observation seen by the agent and the output is the action it selects, so the mapping is the agent's *policy*. What's stopping us from applying supervised learning techniques to mimic the expert's policy? In principle, nothing! This is called **behaviour cloning**.

Definition 8.1 (Behaviour cloning)**.**

1. Collect a training dataset of trajectories $\mathcal{D} = (s^n, a^n)_{n=1}^N$ generated by an **expert policy** π_{expert}. (For example, if the dataset contains M trajectories, each with a finite horizon H, then $N = M \times H$.)
2. Use a supervised learning algorithm $\texttt{fit} : \mathcal{D} \mapsto \tilde{\pi}$ to extract a policy $\tilde{\pi}$ that approximates the expert policy.

Typically, this second task can be framed as **empirical risk minimization** (which we previously saw in sec. 5.3.2):

$$\tilde{\pi} = \arg\min_{\pi \in \Pi} \sum_{n=0}^{N-1} \text{loss}(\pi(s^n), a^n) \tag{8.1}$$

where Π is some class of possible policies, loss is the loss function to measure how different the policy's prediction is from the true observed action, and the supervised learning algorithm itself, also known as the **fitting method**, tells us how to compute this arg min.

How should we choose the loss function? In supervised learning, we saw that the **mean squared error** is a good choice for continuous outputs. However, how should we measure the difference between two actions in a *discrete* action space? In this setting, the policy acts more like a *classifier* that picks the best action in a given state. Rather than considering a deterministic policy that just outputs a single action, we'll consider a stochastic policy π that outputs a *distribution* over actions. This allows us to assign a *likelihood* to observing the entire dataset $\mathcal{D}$ under the policy π, as if the state-action pairs are independent:

$$\mathbb{P}_\pi(\mathcal{D}) = \prod_{n=1}^{N} \pi(a_n \mid s_n) \tag{8.2}$$

Note that the states and actions are *not*, however, actually independent! A key property of interactive tasks is that the agent's output – the action that it takes – may influence its next observation. We want to find a policy under which the training dataset $\mathcal{D}$ is the most likely. This is called the **maximum likelihood estimate** of the policy that generated the dataset:

$$\tilde{\pi} = \arg\max_{\pi \in \Pi} \mathbb{P}_\pi(\mathcal{D}) \tag{8.3}$$

This is also equivalent to doing empirical risk minimization with the **negative log likelihood** as the loss function:

$$\begin{aligned} \tilde{\pi} &= \arg\min_{\pi \in \Pi} -\log \mathbb{P}_\pi(\mathcal{D}) \\ &= \arg\min_{\pi \in \Pi} \sum_{n=1}^{N} -\log \pi(a_n \mid s_n) \end{aligned} \tag{8.4}$$

Can we quantify how well this algorithm works? For simplicity, let's consider the case where the action space is *finite* and both the expert policy and learned policy are deterministic.

Theorem 8.1 (Performance of behaviour cloning)**.** *Suppose the learned policy obtains ε* classification error*. That is, for trajectories drawn from the expert policy, the learned policy chooses a different action at most ε of the time:*

$$\mathbb{E}_{\tau \sim \rho^{\pi_{expert}}} \left[\frac{1}{H} \sum_{h=0}^{H-1} \mathbf{1} \left\{ \tilde{\pi}(s_h) \neq \pi_{expert}(s_h) \right\} \right] \leq \varepsilon \tag{8.5}$$

Then, their value functions differ by

$$|V^{\pi_{expert}} - V^{\widetilde{\pi}}| \leq H^2\varepsilon$$

where H is the horizon of the problem.

Proof. Recall the Performance Difference Lemma (Theorem 7.8). The Performance Difference Lemma allows us to express the difference between $\pi -$ expert and $\tilde{\pi}$ as

$$V_0^{\pi_{\text{expert}}}(s) - V_0^{\widetilde{\pi}}(s) = \mathop{\mathbb{E}}_{\tau \sim \rho^{\pi_{\text{expert}}} | s_0 = s} \left[\sum_{h=0}^{H-1} A_h^{\widetilde{\pi}}(s_h, a_h) \right]. \tag{8.6}$$

Now since the expert policy is deterministic, we can substitute $a_h = \pi_{\text{expert}}(s_h)$. This allows us to make a further simplification: since π_{expert} is deterministic, the advantage of the chosen action is exactly zero:

$$A^{\pi_{\text{expert}}}(s, \pi_{\text{expert}}(s)) = Q^{\pi_{\text{expert}}}(s, \pi_{\text{expert}}(s)) - V^{\pi_{\text{expert}}}(s) = 0. \tag{8.7}$$

But the right-hand-side of eq. 8.6 uses $A^{\widetilde{\pi}}$, not $A^{\pi - \text{expert}}$. To bridge this gap, we now use the assumption that $\tilde{\pi}$ obtains ε classification error. Note that $A_h^{\widetilde{\pi}}(s_h, \pi_{\text{expert}}(s_h)) = 0$ when $\pi_{\text{expert}}(s_h) = \tilde{\pi}(s_h)$. In the case where the two policies differ on s_h, which occurs with probability ε, the advantage is naively upper bounded by H (assuming rewards are bounded between 0 and 1). Taking the final sum gives the desired bound. □

8.3 Distribution shift

Let us return to the driving analogy. Suppose you have taken some driving lessons and now feel comfortable in your neighbourhood. But today you have to travel to an area you haven't visited before, such as a highway, where it would be dangerous to try and apply the techniques you've already learned. This is the issue of *distribution shift*: a policy learned under a certain distribution of states may perform poorly if the distribution of states changes.

This is already a common issue in supervised learning, where the training dataset for a model might not resemble the environment where it gets deployed. In interactive environments, this issue is further exacerbated by the dependency between the observations and the agent's behaviour; if you take a wrong turn early on, it may be difficult or impossible to recover in that trajectory.

How could you learn a strategy for these new settings? In the driving example, you might decide to install a dashcam to record the car's surroundings. That way, once you make it back to safety, you can show the recording to an expert, who can provide feedback at each step of the way. Then the next time you go for a drive, you can remember the expert's advice,

and take a safer route. You could then repeat this training as many times as desired, thereby collecting the expert's feedback over a diverse range of locations. This is the key idea behind *dataset aggregation*.

8.4 Dataset aggregation (DAgger)

The DAgger algorithm assumes that we have *query access* to the expert policy. That is, for a given state s, we can ask for the expert's action $\pi_{\text{expert}}(s)$ in that state. We also need access to the environment for rolling out policies. This makes DAgger an **online** algorithm, as opposed to pure behaviour cloning, which is **offline** since we don't need to act in the environment at all.

You can think of DAgger as a specific way of collecting the dataset $\mathcal{D}$.

Definition 8.2 (DAgger algorithm)**.** Inputs: π_{expert}, an initial policy π_{init}, the number of iterations T, and the number of trajectories N to collect per iteration.

1. Initialize $\mathcal{D} = \{\}$ (the empty set) and $\pi = \pi_{\text{init}}$.
2. For $i = 1, \dots, I$:
   - Collect T trajectories $\tau_0, \dots, \tau_{T-1}$ using the current policy π.
   - For each trajectory τ_n:
     - Replace each action a_h in τ_n with the **expert action** $\pi_{\text{expert}}(s_h)$.
     - Call the resulting trajectory τ_n^{expert}.
   - $\mathcal{D} \leftarrow \mathcal{D} \cup \{\tau_1^{\text{expert}}, \dots, \tau_n^{\text{expert}}\}$.
   - Let $\pi \leftarrow \texttt{fit}(\mathcal{D})$, where `fit` is a behaviour cloning algorithm.
3. Return π.

We leave the implementation as an exercise. How well does DAgger perform?

Theorem 8.2 (Performance of DAgger)**.** *Let π_{expert} be the expert policy and π_{DAgger} be the policy resulting from DAgger. In $I = \widetilde{O}(H^2)$ iterations, with high probability,*

$$|V^{\pi_{expert}} - V^{\pi_{DAgger}}| \leq H\varepsilon, \tag{8.8}$$

where ε is the "classification error" guaranteed by the supervised learning algorithm.

8.5 Key takeaways

Given a task where learning from scratch is too challenging, if we have access to *expert data*, we can use supervised learning to find a policy that imitates the expert demonstrations.

The simplest way to do this is to apply a supervised learning algorithm to an already-collected dataset of expert state-action pairs. This is called **behaviour cloning**. However, given query access to the expert policy, we can do better by integrating its feedback in an online loop. The **DAgger** algorithm is one way of doing this, where we use the expert policy to augment trajectories and then learn from this augmented dataset using behaviour cloning.

8.6 Bibliographic notes and further reading

Earlier interest in imitation learning arose in the context of autonomous driving (Pomerleau, 1991). This task is suitable for imitation learning since expert (or near-expert) driving data is readily available. It is also challenging to express a reward function that captures exactly what we mean by "good driving". Imitation learning methods sidestep this issue by directly training the algorithm to imitate expert demonstrations. The DAgger algorithm (def. 8.2) is due to Ross et al. (2010). The performance guarantee is stated as Ross et al. (2010, thm. 3.4).

Another approach is to infer the reward function from the expert trajectories. This is known as the **inverse reinforcement learning** (IRL) problem (Abbeel & Ng, 2004; Ng & Russell, 2000; S. Russell, 1998). The typical RL problem is going from a reward function to an optimal policy, so the inverse problem is going from an optimal policy to the underlying reward function. One can then use typical RL techniques to optimize for the inferred reward function. This tends to generalize better than direct behaviour cloning. The challenge is that this problem is not well-defined, since any policy is optimal for infinitely many possible reward functions. This means that the researcher must come up with a useful "regularization" assumption to select which of the possible reward functions is the most plausible. Three common modelling assumptions are **maximum-margin IRL** (Ng & Russell, 2000; Ratliff et al., 2006), **Bayesian IRL** (Ramachandran & Amir, 2007), and **maximum-entropy IRL** (Ziebart et al., 2008, 2010).

Another framework for learning behaviour from expert demonstrations is **generative adversarial imitation learning** (Ho & Ermon, 2016; Orsini et al., 2021), inspired by generative adversarial networks (GANs) (Goodfellow et al., 2020). Rather than first learning a reward function from expert data and then optimizing the policy in a separate phase, generative adversarial imitation learning simultaneously learns the reward function and the optimal policy, leading to improved performance.

We can infer a reward function from other expert data besides demonstrations. Expert demonstrations provide a lot of signal but can be prohibitively expensive to collect for some tasks. It

is often easier to *recognize* a good solution than to *generate* one. This leads to the related approach of **reinforcement learning from human feedback** (RLHF). Instead of inferring a reward function from expert demonstrations, we instead infer a reward function from a dataset of expert *rankings* of trajectories. This is the dominant approach used in RL finetuning of large language models (Ouyang et al., 2022; Ziegler et al., 2020). We recommend Lambert (2024) for a comprehensive treatment of RLHF.

Piot et al. (2017) unifies imitation learning and IRL in the **set-policy** framework. Gleave et al. (2022) is a library of modular implementations of imitation learning algorithms in PyTorch. We recommend Zare et al. (2024) for a more comprehensive survey of imitation learning.

9 Tree Search Methods

9.1 Introduction

Have you ever lost a strategy game against a skilled opponent? It probably seemed like they were *ahead of you at every turn*. They might have been *planning ahead* and anticipating your actions, then formulating their strategy to counter yours. If this opponent was a computer, they might have been using one of the strategies that we are about to explore.

9.2 Deterministic, zero sum, fully observable two-player games

In this chapter, we will focus on games that are:

- *deterministic,*
- *zero sum* (one player wins and the other loses),
- *fully observable,* that is, the state of the game is perfectly known by both players,
- for *two players* that alternate turns,

We can represent such a game as a *complete game tree* that describes every possible match. Each possible state is a node in the tree, and since we only consider deterministic games, we can represent actions as edges leading from the current state to the next. Each path through the tree, from root to leaf, represents a single game.

(In games where one can return to a previous board state, to avoid introducing cycles, we might modify the state by also including the number of moves that have been made. This ensures that the complete game tree indeed has no cycles.)

If you could store the complete game tree on a computer, you would be able to win every potentially winnable game by searching all paths from your current state and taking a winning move. We will see an explicit algorithm for this in sec. 9.3. However, as games become more complex, it becomes computationally impossible to search every possible path.

For instance, a chess player has roughly 30 actions to choose from at each turn, and each game takes roughly 40 moves per player, so trying to solve chess exactly using minimax would take somewhere on the order of $30^{80} \approx 10^{118}$ operations. That's 10 billion billion billion billion billion billion billion billion billion billion billion billion billion billion billion operations. As of the time of

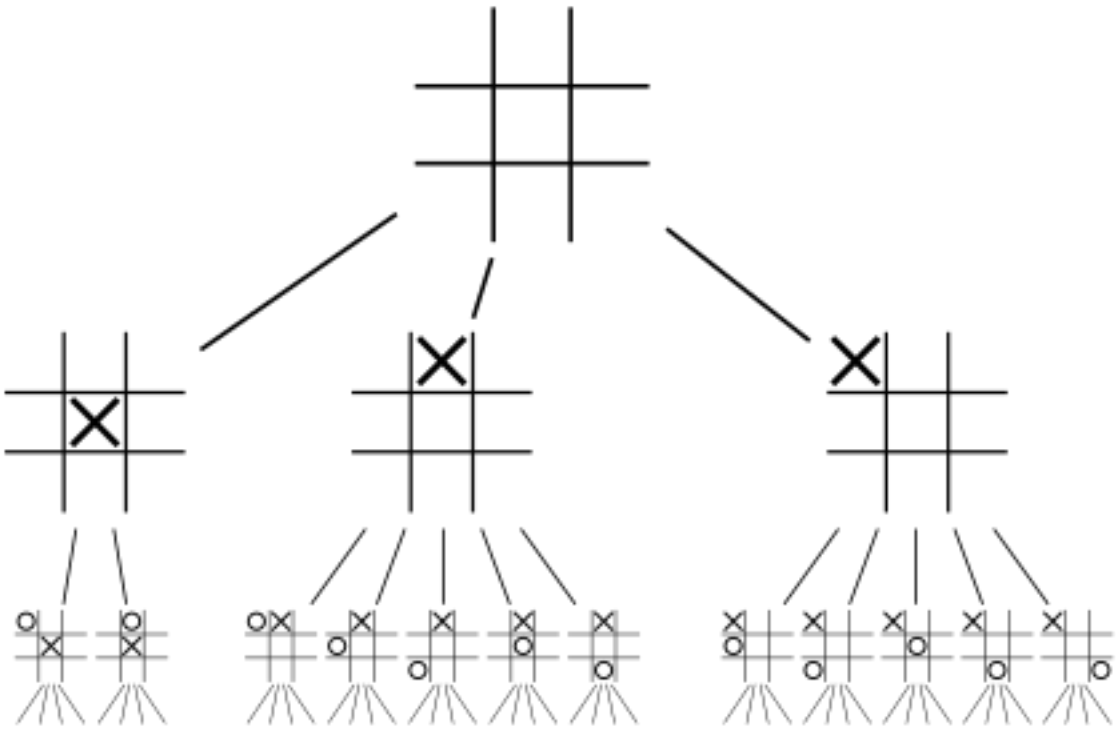


Figure 9.1: The first two layers of the complete game tree of tic-tac-toe.

writing, the fastest processor can achieve almost 10 GHz (10 billion operations per second), so to fully solve chess using minimax is many, many orders of magnitude out of reach.

It is thus intractable, in any realistic setting, to solve the complete game tree exactly. Luckily, only a small fraction of those games ever occur in reality. Later in this chapter, we will explore ways to *prune away* parts of the tree that we know we can safely ignore. We can also *approximate* the value of a state without fully evaluating it. Using these approximations, we can no longer *guarantee* winning the game, but we can come up with strategies that will do well against most opponents.

9.2.1 Notation

Let us now describe these games formally. We'll call the first player Max and the second player Min. Max seeks to maximize the final game score, while Min seeks to minimize the final game score.

- We'll use $\mathcal{S}$ to denote the set of all possible game states.
- The game begins in some **initial state** $s_0 \in \mathcal{S}$.
- Max moves on even turn numbers $h = 2n$, and Min moves on odd turn numbers $h = 2n + 1$, where n is a natural number.
- The space of possible actions, $\mathcal{A}_h(s)$, depends on the state itself, as well as whose turn it is. (For example, in tic-tac-toe, Max can only play `X`s while Min can only play `O`s.)
- The game ends after H total moves (which might be even or odd). We call the final state a **terminal state**.
- P denotes the **state transitions**, that is, $P(s, a)$ denotes the resulting state when taking action $a \in \mathcal{A}(s)$ in state s. We'll assume that this function is time-homogeneous (a.k.a. stationary) and doesn't change across timesteps.
- $r(s)$ denotes the **game score** of the terminal state s. Note that this is some positive or negative value seen by both players: A positive value indicates Max winning, a negative value indicates Min winning, and a value of 0 indicates a tie.

We also call the sequence of states and actions a **trajectory**.

Exercise 9.1 (Variable length games)**.** Above, we suppose that the game ends after H total moves. But most real games have a *variable* length. How would you describe this?

Example 9.1 (Tic-tac-toe)**.** Let us frame tic-tac-toe in this setting.

- Each of the 9 squares is either empty, marked X, or marked O. So there are $|\mathcal{S}| = 3^9$ potential states. Not all of these may be reachable!
- The initial state s_0 is the empty board.

- The set of possible actions for Max in state s, $\mathcal{A}_{2n}(s)$, is the set of tuples ("X", i) where i refers to an empty square in s. Similarly, $\mathcal{A}_{2n+1}(s)$ is the set of tuples ("O", i) where i refers to an empty square in s.
- We can take $H = 9$ as the longest possible game length.
- $P(s, a)$ for a *nonterminal* state s is simply the board with the symbol and square specified by a marked into s. Otherwise, if s is a *terminal* state, i.e. it already has three symbols in a row, the state no longer changes.
- $r(s)$ at a *terminal* state is $+1$ if there are three Xs in a row, -1 if there are three Os in a row, and 0 otherwise.

Our notation may remind you of Chapter 2. Given that these games also involve a sequence of states and actions, can we formulate them as finite-horizon MDPs? The two settings are not exactly analogous, since in MDPs we only consider a *single* policy, while these games involve two distinct players with opposite objectives. Since we want to analyze the behaviour of *both* players at the same time, describing such a game as an MDP is more trouble than it's worth.

9.3 Min-max search

In the introduction, we claimed that we could win any potentially winnable game by looking ahead and predicting the opponent's actions. This would mean that each *nonterminal* state already has some predetermined game score. That is, in each state, it is already possible to determine which player is going to win.

Let $V_h^\star(s)$ denote the game score under optimal play from both players starting in state s at time h.

Definition 9.1 (Min-max search algorithm)**.** The best move for Max is the one that leads to the maximum value. Correspondingly, the best move for Min is the one that leads to the minimum value. This naturally gives rise to a recursive definition of the value of each state under optimal play:

$$V_h^\star(s) = \begin{cases} r(s) & h = H \\ \max_{a \in \mathcal{A}_h(s)} V_{h+1}^\star(P(s,a)) & h \text{ is even and } h < H \\ \min_{a \in \mathcal{A}_h(s)} V_{h+1}^\star(P(s,a)) & h \text{ is odd and } h < H \end{cases}.$$

Recall that $P(s, a)$ denotes the next state after taking action a in state s.

We can compute this by **dynamic programming**. We start at the terminal states, where the game's outcome is known, and work backwards. This might remind you of policy evaluation in finite-horizon MDPs (sec. 2.3.1).

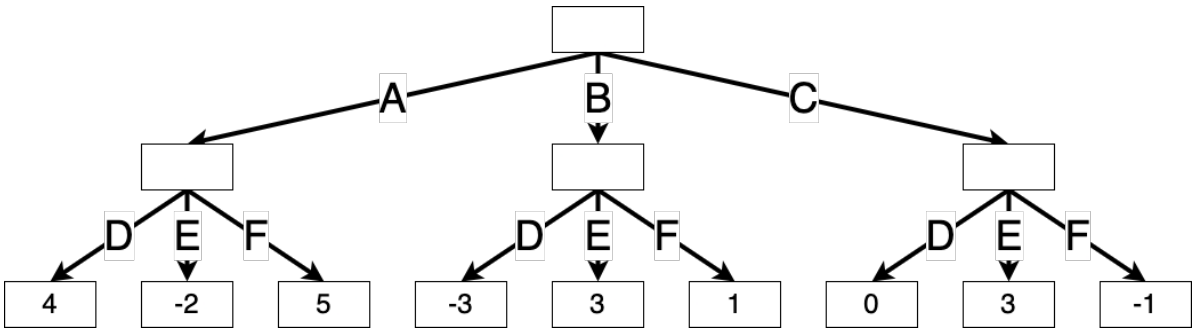
This translates directly into a recursive depth-first search algorithm for searching the complete game tree.

```
function minimax_search(s, player)
  if env.is_terminal(s)
    return (None, env.winner(s))
  end if
  if player ≡ max
    (a_max, v_max) ← (None, None)
    for a ∈ A(s) do
      (_, v) ← minimax_search(P(s, a), min)
      if v > v_max
        (a_max, v_max) ← (a, v)
      end if
    end for
    return (a_max, v_max)
  else
    (a_min, v_min) ← (None, None)
    for a ∈ A(s) do
      (_, v) ← minimax_search(P(s, a), max)
      if v < v_min
        (a_min, v_min) ← (a, v)
      end if
    end for
    return (a_min, v_min)
  end if
end function
```

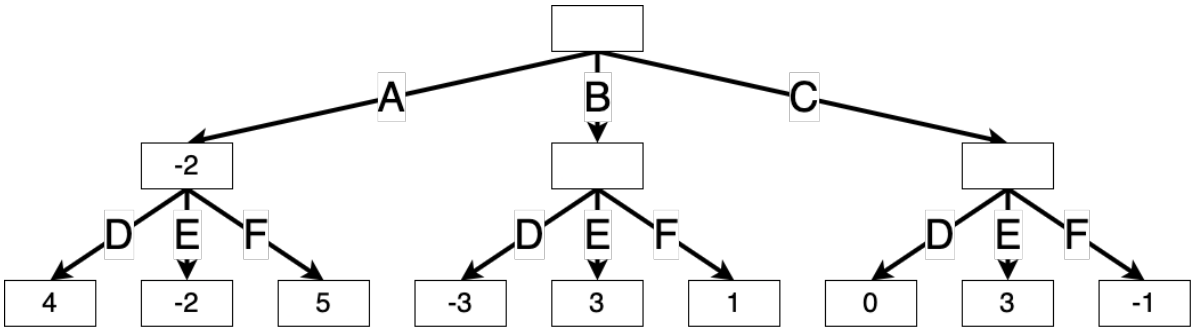
Example 9.2 (Min-max search for a simple game)**.** Consider a simple game with just two steps: Max chooses one of three possible actions (A, B, C), and then Min chooses one of three possible actions (D, E, F). The combination leads to a certain integer outcome, shown in the table below:

	D	E	F
A	4	-2	5
B	-3	3	1
C	0	3	-1

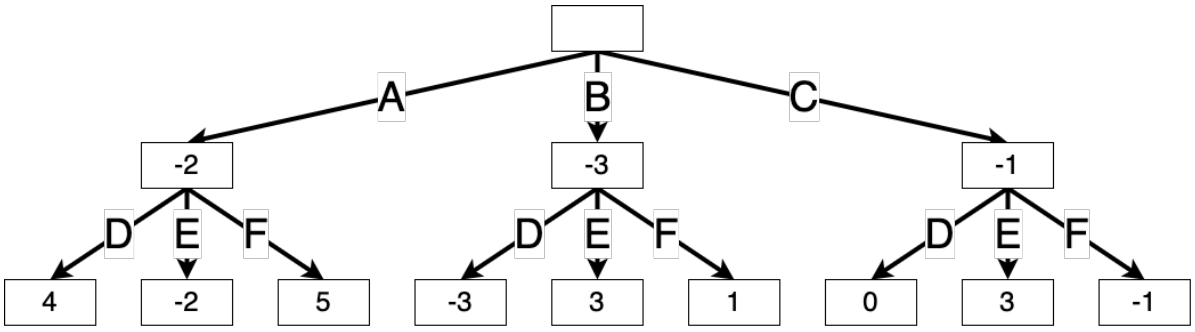
We can visualize this as the following complete game tree, where each box contains the value $V_h^\star(s)$ of that node. The min-max values of the terminal states are already known:



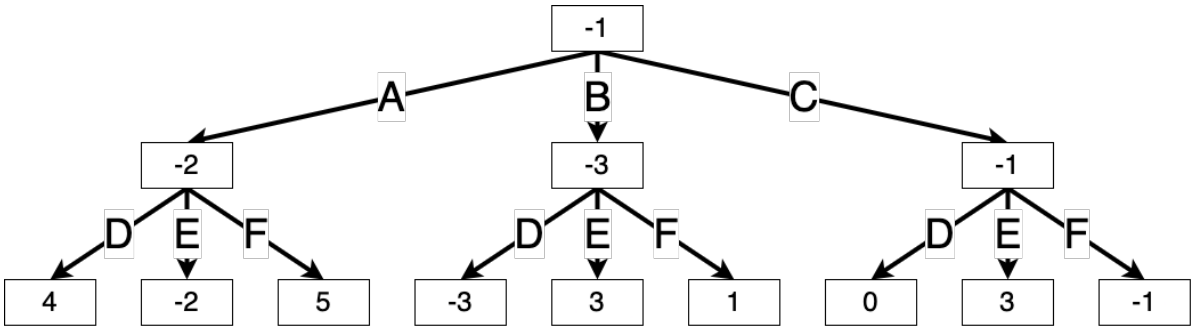
We begin min-max search at the root, exploring each of Max's actions. Suppose Max chooses action A. Then Min will choose action E to minimize the game score, making the value of this game node $\min(4, -2, 5) = -2$.



Similarly, if Max chooses action B, then Min will choose action D, and if Max chooses action C, then Min will choose action F. We can fill in the values of these nodes accordingly:



Thus, Max's best move is to take action C, resulting in a game score of $\max(-2, -3, -1) = -1$.



9.3.1 Complexity of min-max search

At each of the H timesteps, this algorithm iterates through the entire action space at that state, and therefore has a time complexity of H^{n_A} (where n_A is the largest number of actions possibly available at once). This makes the min-max algorithm impractical for even moderately sized games.

But do we need to compute the exact value of *every* possible state? Instead, is there some way we could "ignore" certain actions and their subtrees if we already know of better options? The **alpha-beta search** makes use of this intuition.

9.4 Alpha-beta pruning

For a given deterministic, zero-sum, fully observable two-player game (sec. 9.2), we have seen that it is possible to "solve" the game, that is, determine the best move in every situation, using min-max search (sec. 9.3). However, the time complexity of min-max search makes it infeasible for most scenarios. Alpha-beta pruning improves min-max search by *pruning* down the search tree.

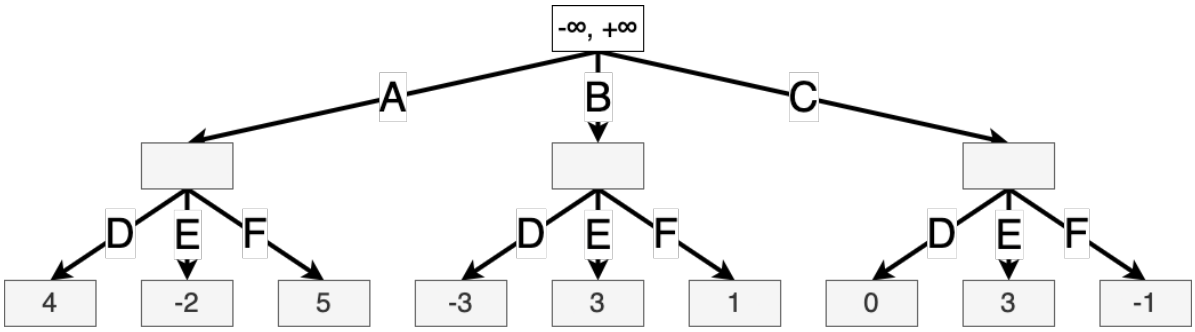
Suppose Max is in state s is deciding between action a and a'. If at any point Max finds out that action a' is no better than action a, she doesn't need to evaluate action a' any further.

Concretely, we run min-max search as above, except now we keep track of two additional parameters $\alpha(s)$ and $\beta(s)$ while evaluating each state:

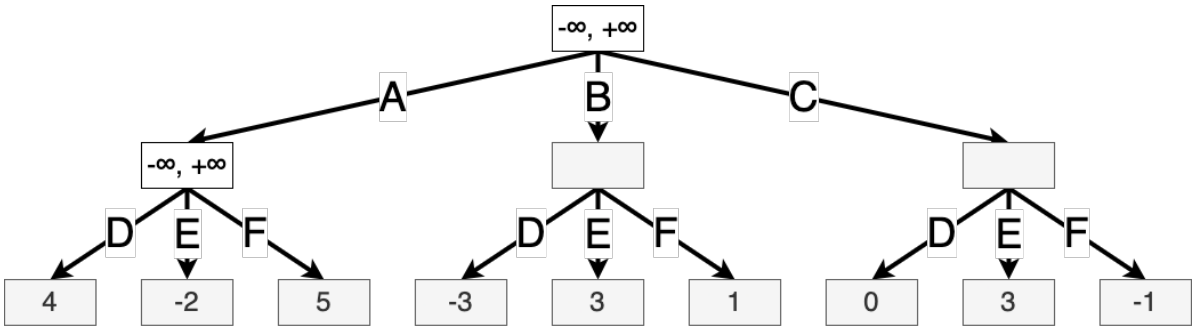
- Starting in state s, Max can achieve a game score of *at least* $\alpha(s)$ assuming Min plays optimally. That is, $V_h^\star(s) \geq \alpha(s)$ at all points.
- Analogously, starting in state s, Min can ensure a game score of *at most* $\beta(s)$ assuming Max plays optimally. That is, $V_h^\star(s) \leq \beta(s)$ at all points.

Suppose we are evaluating $V_h^\star(s)$, where it is Max's turn (h is even). We update $\alpha(s)$ to be the *highest* minimax value achievable from s so far. That is, the value of s is *at least* $\alpha(s)$. Suppose Max chooses action a, which leads to state s', in which it is Min's turn. If any of Min's actions in s' achieve a value $V_{h+1}^\star(s') \leq \alpha(s)$, we know that Max would not choose action a, since they know that it is *worse* than whichever action gave the value $\alpha(s)$. Similarly, to evaluate a state on Min's turn, we update $\beta(s)$ to be the *lowest* value achievable from s so far. That is, the value of s is *at most* $\beta(s)$. Suppose Min chooses action a, which leads to state s' for Max. If Max has any actions that do *better* than $\beta(s)$, they would take it, making action a a suboptimal choice for Min.

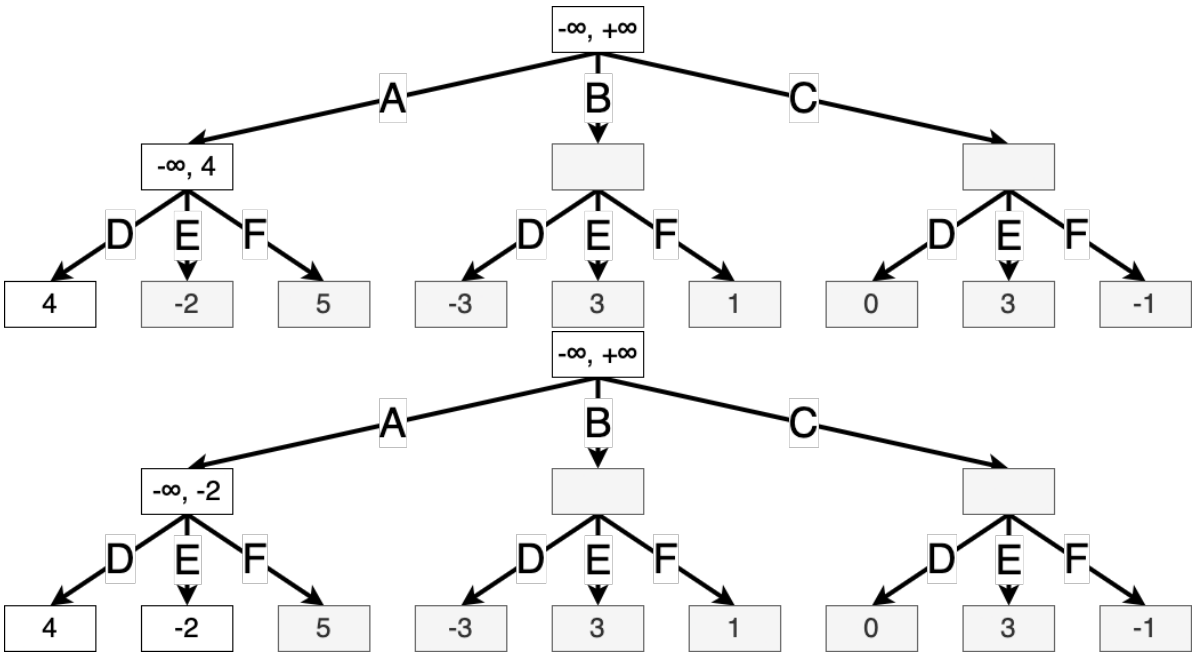
Example 9.3 (Alpha-beta search for a simple game)**.** Let us use the same simple game from ex. 9.2. We list the values of $\alpha(s), \beta(s)$ in each node throughout the algorithm. These values are initialized to $-\infty, +\infty$ respectively. We shade any squares that have not been visited by the algorithm, and we assume that actions are evaluated from left to right.



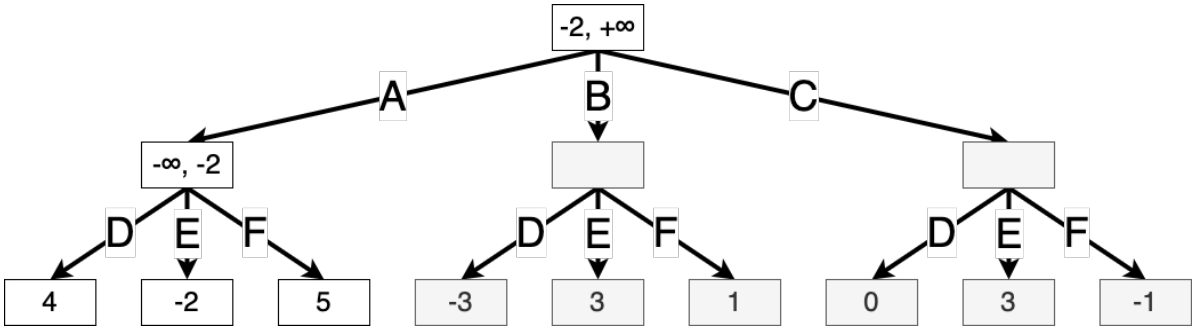
Suppose Max takes action A. Let s' be the resulting game state. The values of $\alpha(s')$ and $\beta(s')$ are initialized at the same values as the root state, since we want to prune a subtree if there exists a better action at any step higher in the tree.



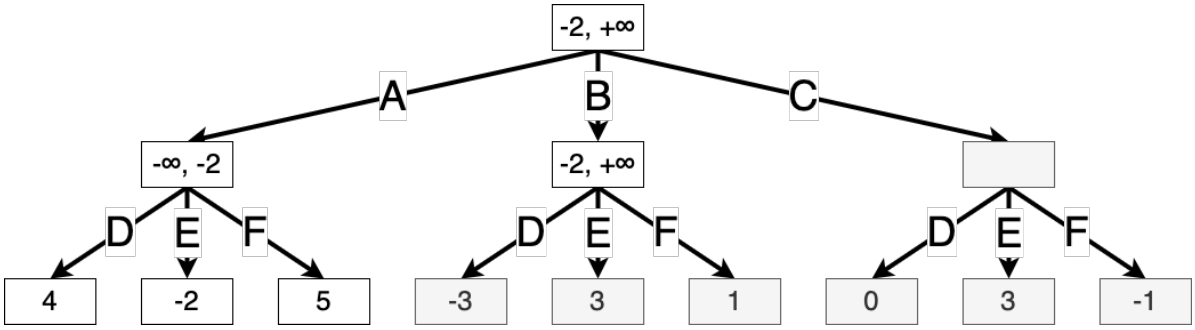
Then we iterate through Min's possible actions, updating the value of $\beta(s')$ as we go.



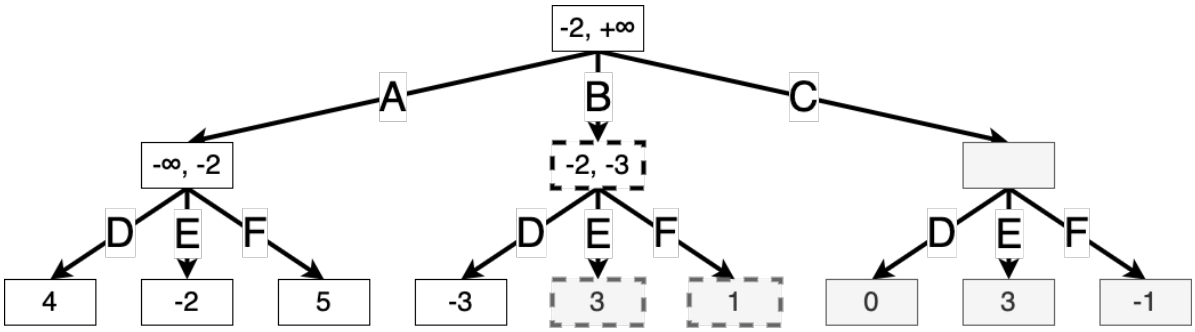
Once the value of state s' is fully evaluated, we know that Max can achieve a value of *at least* -2 starting from the root, and so we update $\alpha(s)$, where s is the root state:



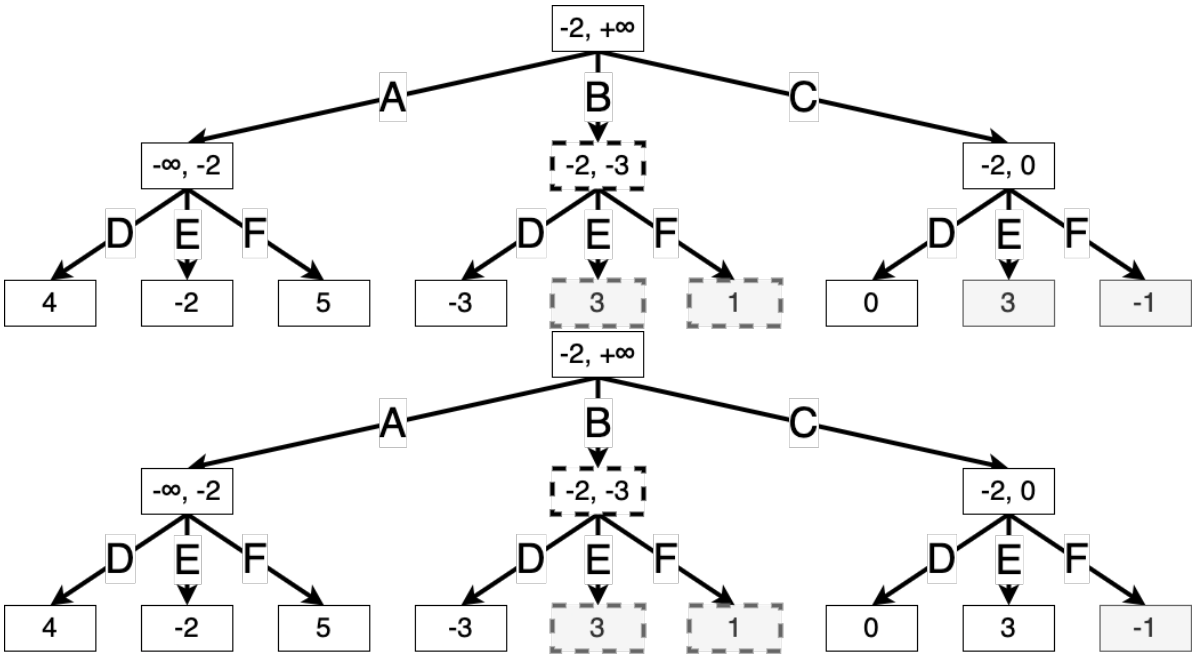
Then Max imagines taking action B. Again, let s' denote the resulting game state. We initialize $\alpha(s')$ and $\beta(s')$ from the root:



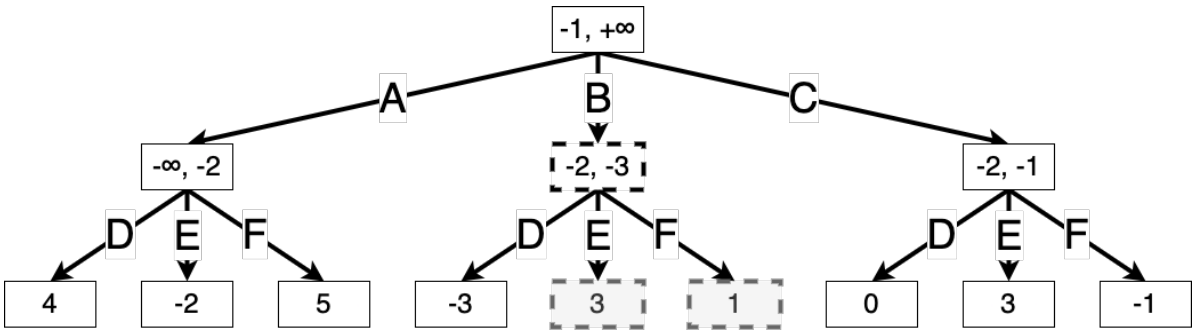
Now suppose Min takes action D, resulting in a value of -3. We see that $V_h^\star(s') = \min(-3, x, y)$, where x and y are the values of the remaining two actions. But since $\min(-3, x, y) \leq -3$, we know that the value of s' is at most -3. But Max can achieve a better value of $\alpha(s') = -2$ by taking action A, and so Max will never take action B, and we can prune the search here. We will use dotted lines to indicate states that have been ruled out from the search:



Finally, suppose Max takes action C. For Min's actions D and E, there is still a chance that action C might outperform action A, so we continue expanding:



Finally, we see that Min taking action F achieves the minimum value at this state. This shows that optimal play is for Max to take action C, and Min to take action F.



function alpha_beta_search(s, player, α, β)
  if env.is_terminal(s)
    return (None, env.winner(s))
  end if
  if player $\equiv$ max
    $(a_{\max}, v_{\max}) \leftarrow$ (None, None)
    for $a \in$ actions **do**
      $(\_, v) \leftarrow$ minimax_search(env.step(s, a), min, α, β)
      if $v > v_{\max}$
        $(a_{\max}, v_{\max}) \leftarrow (a, v)$
        $\alpha \leftarrow \max(\alpha, v)$
      end if
      if $v_{\max} \geq \beta$
        return $(a_{\max}, v_{\max})$
      end if
    end for
    return $(a_{\max}, v_{\max})$
  else
    $(a_{\min}, v_{\min}) \leftarrow$ (None, None)
    for $a \in$ actions **do**
      $(\_, v) \leftarrow$ minimax_search(env.step(s, a), max)
      if $v < v_{\min}$
        $(a_{\min}, v_{\min}) \leftarrow (a, v)$
        $\beta \leftarrow \min(\beta, v)$
      end if
      if $v_{\min} \leq \alpha$
        return $(a_{\min}, v_{\min})$
      end if
    end for
    return $(a_{\min}, v_{\min})$
  end if
end function

How do we choose what *order* to explore the branches? As you can tell, this significantly affects the efficiency of the pruning algorithm. If Max explores the possible actions in order from worst to best, they will not be able to prune any branches at all! Additionally, to verify that an action is suboptimal, we must run the search recursively from that action, which ultimately requires traversing the tree all the way to a leaf node. The longer the game might possibly last, the more computation we have to run.

In practice, we can often use background information about the game to develop a **heuristic** for evaluating possible actions. If a technique is based on background information or intuition, especially if it isn't rigorously justified, we call it a heuristic.

Can we develop *heuristic methods* for tree exploration that works for all sorts of games?

9.5 Monte Carlo Tree Search

The task of *evaluating actions* in a complex environment might seem familiar. We've encountered this problem before in both multi-armed bandits (Chapter 4) and Markov decision processes (Chapter 2). Now we'll see how to combine concepts from these to form a more general and efficient tree search heuristic called **Monte Carlo Tree Search** (MCTS).

When a problem is intractable to solve *exactly*, we often turn to *approximate* algorithms that sacrifice some accuracy in exchange for computational efficiency. MCTS also improves on alpha-beta search in this sense. As the name suggests, MCTS uses *Monte Carlo* simulation, that is, collecting random samples and computing the sample statistics, in order to *approximate* the value of each action.

As before, we imagine a game tree in which each path represents an *entire game*. MCTS assigns values to only the game states that are *relevant* to the *current game*. That is, we maintain a *search tree* that we gradually expand at each move. For comparison, in alpha-beta search, the entire tree only needs to be solved *once*, and from then on, choosing an action is as simple as taking a maximum over the previously computed values.

The crux of MCTS is approximating the win probability of a state by a *sample probability*. In practice, MCTS is used for games with *binary outcomes* where $r(s) \in \{+1, -1\}$, and so this is equivalent to approximating the final game score. To approximate the win probability from state s, MCTS samples random games starting in s and computes the sample proportion of those that the player wins.

Note that, for a given state s, choosing the best action a can be framed as a multi-armed bandit problem, where each action corresponds to an arm, and the reward distribution of arm k is the distribution of the game score over random games after choosing that arm. The most commonly used bandit algorithm in practice for MCTS is the upper confidence bound algorithm (sec. 4.7).

Remark 9.1 (Summary of UCB). Let us quickly review the UCB algorithm for a multi-armed bandit problem (sec. 4.7) For each arm k, we track the sample mean

$$\hat{\mu}_t^k = \frac{1}{N_t^k} \sum_{\tau=0}^{t-1} \mathbf{1}\left\{a_\tau = k\right\} r_\tau$$

of all rewards from that arm up to time t. Then we construct a *confidence interval*

$$C_t^k = [\hat{\mu}_t^k - B_t^k, \hat{\mu}_t^k + B_t^k],$$

where $B_t^k = \sqrt{\frac{\ln(2t/\delta)}{2N_t^k}}$ is given by Hoeffding's inequality, so that with probability δ (some fixed parameter we choose), the true mean μ^k lies within C_t^k. Note that B_t^k scales like $\sqrt{1/N_t^k}$, i.e. the more we have visited that arm, the more confident we get about it, and the narrower the confidence interval.

To select an arm, we pick the arm with the highest *upper confidence bound*.

This means that, for each edge in the game tree, which corresponds to a state-action pair (s, a), we keep track of the statistics required to compute its UCB:

- How many times it has been "visited" ($N_t^{s,a}$)
- How many of those visits resulted in victory ($\sum_{\tau=0}^{t-1} \mathbf{1}\left\{(s_\tau, a_\tau) = (s, a)\right\} r_\tau$). Let us call this latter value $W_t^{s,a}$ (for number of "wins").

What does t refer to in the above expressions? Recall t refers to the number of time steps elapsed in the *bandit environment*. As mentioned above, each state s corresponds to its own bandit environment, and so t refers to N^s, that is, how many actions have been taken from state s. This term, N^s, gets incremented as the algorithm runs; for simplicity, we won't introduce another index to track how it changes.

Definition 9.2 (Monte Carlo tree search algorithm)**.** Here we describe how to perform a Monte Carlo tree search for choosing a single action in state s_{start}.

Inputs:

- T, the number of iterations per move
- π_{rollout}, the **rollout policy** for randomly sampling games
- c, a positive value that encourages exploration

To choose a single move starting at state s_{start}, MCTS first tries to estimate the UCB values for each of the possible actions $\mathcal{A}(s_{\text{start}})$, and then chooses the best one. To estimate the UCB values, it repeats the following four steps T times:

1. **Selection**: We start at $s = s_{\text{start}}$. Let τ be an empty list that we will use to track states and actions.

- Choose $a \leftarrow \arg\max_k \mathrm{UCB}^{s,k}$, where

$$\mathrm{UCB}^{s,a} = \frac{W^{s,a}}{N^{s,a}} + c\sqrt{\frac{\ln N^s}{N^{s,a}}} \tag{9.1}$$

  - Append (s, a) to τ
  - If s has at least one action that hasn't been taken, move onto the next step. Otherwise, move to the next state $s \leftarrow P(s, a)$ and repeat.

2. **Expansion**: Let s_{new} denote the final state in τ (that has at least one action that hasn't been taken). Choose one of these unexplored actions from s_{new}. Call it a_{new}. Add it to τ.
3. **Simulation**: Simulate a complete game episode by starting with the action a_{new} and then playing according to π_{rollout}. This results in the outcome $r \in \{+1, -1\}$.
4. **Backup**: For each $(s, a) \in \tau$:

   - Set $N^{s,a} \leftarrow N^{s,a} + 1$
   - $W^{s,a} \leftarrow W^{s,a} + r$
   - Set $N^s \leftarrow N^s + 1$

After T repeats of the above, we return the action with the highest UCB value eq. 9.1. Then play continues.

Between turns, we can keep the subtree whose statistics we have visited so far. However, the rest of the tree for the actions we did *not* end up taking gets discarded.

The application which brought the MCTS algorithm to fame was DeepMind's **AlphaGo** Silver et al. (2016). Since then, it has been used in numerous applications ranging from games to automated theorem proving.

How accurate is this Monte Carlo estimation? It depends heavily on the rollout policy π_{rollout}. If the distribution π_{rollout} induces over games is very different from the distribution seen during real gameplay, we might end up with a poor value approximation.

9.5.1 Incorporating value functions and policies

To remedy this, we might make use of a value function $v : \mathcal{S} \to \mathbb{R}$ that more efficiently approximates the value of a state. Then, we can replace the simulation step of def. 9.2 with evaluating $r = v(s - \text{next})$, where $s_{\text{next}} = P(s_{\text{new}}, a_{\text{new}})$.

We might also make use of a **"guiding" policy** $\pi_{\text{guide}} : \mathcal{S} \to \triangle(\mathcal{A})$ that provides "intuition" as to which actions are more valuable in a given state. We can scale the exploration term of eq. 9.1 according to the policy's outputs.

Putting these together, we can describe an updated version of MCTS that makes use of these value functions and policy:

Definition 9.3 (Monte Carlo tree search with policy and value functions)**.** Inputs: - T, the number of iterations per move - v, a value function that evaluates how good a state is - π_{guide}, a guiding policy that encourages certain actions - c, a positive value that encourages exploration

To select a move in state s_{start}, we repeat the following four steps T times:

1. **Selection**: We start at $s = s_{\text{start}}$. Let τ be an empty list that we will use to track states and actions.
   - Until s has at least one action that hasn't been taken:
     - Choose $a \leftarrow \arg\max_k \text{UCB}^{s,k}$, where

$$\text{UCB}^{s,a} = \frac{W^{s,a}}{N^s} + c \cdot \pi_{\text{guide}}(a \mid s)\sqrt{\frac{\ln N^s}{N^{s,a}}} \tag{9.2}$$

     - Append (s, a) to τ
     - Set $s \leftarrow P(s, a)$
2. **Expansion**: Let s_{new} denote the final state in τ (that has at least one action that hasn't been taken). Choose one of these unexplored actions from s_{new}. Call it a_{new}. Add it to τ.
3. **Simulation**: Let $s_{\text{next}} = P(s_{\text{new}}, a_{\text{new}})$. Evaluate $r = v(s_{\text{next}})$. This approximates the value of the game after taking the action a_{new}.
4. **Backup**: For each $(s, a) \in \tau$:
   - $N^{s,a} \leftarrow N^{s,a} + 1$
   - $W^{s,a} \leftarrow W^{s,a} + r$
   - $N^s \leftarrow N^s + 1$

We finally return the action with the highest UCB value eq. 9.2. Then play continues. As before, we can reuse the tree across timesteps.

How do we actually compute a useful π_{guide} and v? If we have some existing dataset of trajectories, we could use Chapter 8 (that is, imitation learning) to generate a policy π_{guide} via behaviour cloning and learn v by regressing the game outcomes onto states. Then, plugging these into def. 9.3 results in a stronger policy by using tree search to "think ahead".

But we don't have to stop at just one improvement step; we could iterate this process via **self-play**.

9.5.2 Self-play

Recall the policy iteration algorithm for solving an MDP (sec. 2.4.4.2). Policy iteration alternates between **policy evaluation** (taking π and computing V^π) and **policy improvement** (setting π to be greedy with respect to V^π). We can think of MCTS as a "policy improvement"

operation: for a given policy π^0, we can use it to guide MCTS. This results in an algorithm that is *itself* a policy that maps from states to actions. This improved policy (using MCTS) is usually called the **search policy**. Denote it by π^0_{MCTS}. Now, we can use imitation learning techniques (Chapter 8) to obtain a new policy π^1 that imitates π^0_{MCTS}. We can now use π^1 to guide MCTS, and repeat.

Definition 9.4 (MCTS with self-play)**.** Input:

- A parameterized policy class $\pi_\theta : \mathcal{S} \to \triangle(\mathcal{A})$
- A parameterized value function class $v_\lambda : \mathcal{S} \to \mathbb{R}$
- A number of trajectories M to generate
- The initial parameters θ^0, λ^0

For $t = 0, \dots, T-1$:

- **Policy improvement**: Let π^t_{MCTS} denote the policy obtained by def. 9.3 with π_{θ^t} and $v - \lambda^t$. We use $\pi^t -$ MCTS to play against itself M times. This generates M trajectories $\tau - 0, \dots, \tau - M - 1$.
- **Policy evaluation**: Use behaviour cloning to find a set of policy parameters θ^{t+1} that mimic the behaviour of π^t_{MCTS} and a set of value function parameters λ^{t+1} that approximate its value function. That is,

$$\theta^{t+1} \leftarrow \arg\min_\theta \sum_{m=0}^{M-1} \sum_{h=0}^{H-1} -\log \pi_\theta(a^m_h \mid s^m_h)$$

$$\lambda^{t+1} \leftarrow \arg\min_\lambda \sum_{m=0}^{M-1} \sum_{h=0}^{H-1} (v_\lambda(s^m_h) - R(\tau_m))^2$$

Note that in implementation, the policy and value are typically both returned by a single deep neural network, that is, with a single set of parameters, and the two loss functions are added together.

This algorithm was brought to fame by AlphaGo Zero (Silver et al., 2017).

9.5.2.1 Extending to continuous rewards (*)

In the search algorithm above, we used $W^{s,a}$ to track the number of times the policy wins after taking action a in state s. This binary outcome can easily be generalized to a *continuous* reward at each state-action pair. This is the reward function we assumed when discussing MDPs (Chapter 2).

9.6 Key takeaways

In this chapter, we explored tree search-based algorithms for deterministic, zero sum, fully observable two-player games. We began with min-max search (sec. 9.3), an algorithm for exactly solving the game value of every possible state. However, this is impossible to execute in practice, and so we must resort to various ways to reduce the number of states and actions that we must explore. Alpha-beta search (sec. 9.4) does this by *pruning* away states that we already know to be suboptimal, and MCTS (sec. 9.5) *approximates* the value of states instead of evaluating them exactly.

9.7 Bibliographic notes and further reading

S. J. Russell & Norvig (2021, ch. 5) provides an excellent overview of search methods in games. The original AlphaGo paper Silver et al. (2016) was a groundbreaking application of these technologies. Silver et al. (2017) removed the imitation learning phase, learning the optimal policy from scratch using self-play. AlphaZero (Silver et al., 2018) then extended to other games beyond Go, namely shogi and chess, also learning from scratch. In MuZero (Schrittwieser et al., 2020), this was further extended by *learning* a model of the game dynamics. EfficientZero (Ye et al., 2021) presented a more sample-efficient algorithm based on MuZero. Gumbel MuZero (Danihelka et al., 2021) greatly improved the computational efficiency of MuZero by reducing the number of rollouts required. Stochastic MuZero (Antonoglou et al., 2021) extends MuZero to stochastic environments.

While search methods are extremely powerful, they are also computationally intensive, and have therefore historically been written in lower-level languages such as C or C++ rather than Python. The development of the JAX framework addresses this issue by providing a readable high-level Python library that compiles to code optimized for specific hardware. In particular, the Mctx library (Babuschkin et al., 2020) provides usable implementations of MuZero and its variants above.

10 Exploration in MDPs

10.1 Introduction

One of the key challenges of reinforcement learning is the *exploration-exploitation tradeoff*. Should we *exploit* actions we know will give high reward, or should we *explore* different actions to discover potentially better strategies? An algorithm that doesn't explore effectively might easily *overfit* to certain areas of the state space, and fail to generalize once they enter a region they haven't yet seen.

In the multi-armed bandit setting (Chapter 4), we studied the upper confidence bound (UCB) algorithm (sec. 4.7) that incentivizes the learner to explore arms that it is uncertain about. In particular, UCB relies on **optimism in the face of uncertainty**: it chooses arms based on an *overestimate* of the arm's true mean reward. In this chapter, we will see how to generalize this idea to the MDP setting.

10.1.1 Sparse reward

Exploration is crucial in **sparse reward** problems where $r(s, a) = 0$ for most (or nearly all) states and actions. Often, the agent must take a specific sequence of actions before any reward is observed.

Example 10.1 (Chain MDP)**.** Here's a simple example of an MDP with sparse rewards:

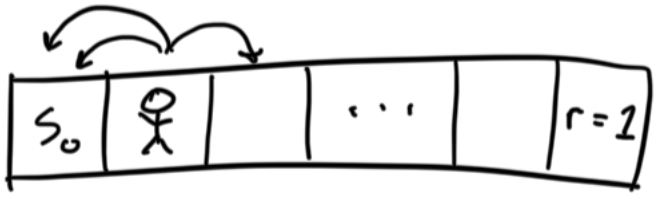


Figure 10.1: An illustration of the chain MDP environment.

There are $|\mathcal{S}|$ cells arranged in a chain. The agent starts in the leftmost cell. The rightmost state is a terminal state. In every state, there are three possible actions, two of which move the agent left and one which moves the agent right. (The two "left" actions do nothing in the

leftmost cell.) The reward function gives a reward of 1 for taking the action that enters the rightmost cell, and zero otherwise.

The problem of sparse rewards is especially prevalent in RL compared to supervised learning. In most supervised learning tasks, every labelled sample provides some useful signal. However, in RL, algorithms that don't *systematically* explore new states may fail to learn anything meaningful within a reasonable amount of time.

Consider the algorithms we've covered so far for unknown environments: policy gradient methods (Chapter 7) and fitted DP methods (Chapter 6). How would these do on this problem?

Remark 10.1 (Policy gradient methods fail on sparse reward). Policy gradient algorithms require the gradient to be *nonzero* in order to learn. If we never observe any reward, the gradient will always be zero, and the policy will never change or improve. If we think of the expected total reward as a function $J(\theta)$ of the policy parameters, we can visualize the *graph* of J as being mostly flat, making it impossible to "climb the hill" from almost every random initialization.

Remark 10.2 (Fitted DP methods fail on sparse reward). Fitted DP algorithms run into a similar issue: as we randomly interact with the environment, we never observe any reward, and so the reward model simply gives zero for every state-action pair. In expectation, it would take a computationally infeasible number of rollouts to observe the reward by chance.

This is quite disheartening! The sophisticated methods we've developed, which can exceed human-level performance on a wide variety of tasks, fail on this problem that seems almost trivial.

Of course, a simple way to solve the "chain MDP" in ex. 10.1 is to actively visit unseen states. For a policy that visits a new state in each rollout, the final cell can be reached in $O(|\mathcal{S}|)$ rollouts (i.e. $O(|\mathcal{S}|^2)$ time). The rest of this chapter will consider ways to *explicitly* explore unknown states.

10.1.2 Reward shaping

One workaround to sparse reward problems in practice is to *shape* the reward function using domain knowledge. For example, in ex. 10.1, we (that is, the practitioners) know that travelling to the right is the correct action, so we could design a reward function that provides a reward of 0.1 for the action that moves to the right. A similar strategy is used in practice for many chess or board game algorithms where capturing the opponent's pieces earns some positive reward.

Though this might seem obvious, designing a useful reward function can be challenging in practice. The agent may learn to exploit the intermediate rewards rather than solve the original goal. A famous example is the agent trained to play the CoastRunners game, in

which players race boats around a racetrack. However, the algorithm found that it could achieve higher reward (i.e. in-game score) by refusing to go around the racetrack and instead collecting bonus points in a loop!

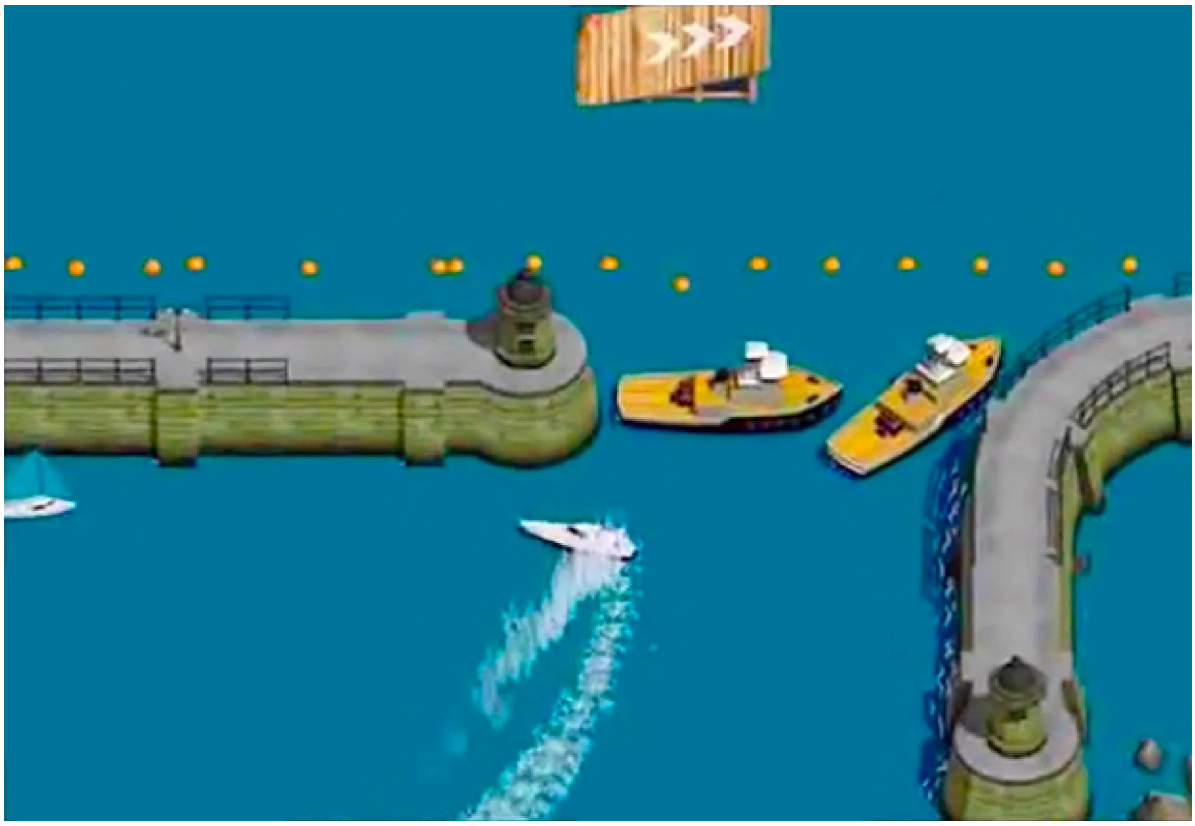


Figure 10.2: An RL agent collects bonus points instead of participating in the race. Image from Clark & Amodei (2024).

This phenomenon is known as **reward hacking** or **Goodhart's law**. Reward hacking is essentially a special case of "finding loopholes" around the written guidelines (or in this case, the reward signal used for training); think of folk stories such as King Midas or the Monkey's Paw. When RL algorithms are deployed in high-stakes scenarios, it is crucial to verify the learned policy's behaviour and ensure that it is aligned to the designer's intentions.

10.2 Exploration in deterministic MDPs

Let us address the exploration problem in a *deterministic* MDP, that is, where taking action a in state s always leads to the state $P(s, a) \in \mathcal{S}$. How can we methodically visit every single state-action pair?

In the multi-armed bandit setting (Chapter 4), there are no states, so it's trivial to visit every "state-action pair": just pull each arm once. But in the MDP setting, in order to achieve a particular state-action pair (s, a), one must plan out a path from the initial state.

We can do this by constructing an MDP where only unseen state-action pairs are rewarded, and using value iteration/dynamic programming (sec. 2.3.2) to reach the unknown states in $M_{\mathcal{D}}$. Concretely, we keep a set $\mathcal{D}$ of all the (s, a, r, s') tuples we've observed. Each episode, we use $\mathcal{D}$ to construct a fully known MDP, $M_{\mathcal{D}}$, in which only unseen state-action pairs are rewarded.

Definition 10.1 (Explore-then-exploit algorithm)**.** Suppose that every state can be reached from the initial state within a single episode.

1. $\mathcal{D} \leftarrow \emptyset$
2. For $T = 0, 1, 2, ...$ (until the entire MDP has been explored):
    1. Construct $M_{\mathcal{D}}$ using $\mathcal{D}$. That is, the state transitions are set to those observed in $\mathcal{D}$, and the reward is set to 0 for all state-action pairs in $\mathcal{D}$, and 1 otherwise.
    2. Execute DP (sec. 2.3.2) on the known MDP $M_{\mathcal{D}}$ to compute the optimal policy $\pi^\star_{\mathcal{D}}$.
    3. Execute $\pi^\star_{\mathcal{D}}$ in $M_{\mathcal{D}}$. This will visit some (s, a) not yet in $\mathcal{D}$, and observe the reward $r(s, a)$ and next state $P(s, a)$.
    4. $\mathcal{D} \leftarrow \mathcal{D} \cup \{(s, a, r, s')\}$, where $s' = P(s, a), r = r(s, a)$ are the observed state transition and reward.

Remark 10.3 (Path planning is graph traversal). Review the dynamic programming algorithm for a finite-horizon MDP (sec. 2.3.2). Note that in the constructed MDP $M_{\mathcal{D}}$, this is identical to a **breadth-first search** beginning from the desired state at the final timestep: each state-timestep pair is a node in the graph, and the state transitions determine the (directed) edges. Each state-timestep pair from which it is possible to reach the desired state is assigned a value of 1. The policy serves to backtrack through these state-timestep pairs, returning to the root node of the search: the desired state.

We can easily measure the **per-episode regret** of this algorithm.

Definition 10.2 (Per-episode regret)**.** We aim to evaluate some iterative policy optimization algorithm. Let π^t be the policy returned by the algorithm after t iterations. The per-episode regret across T iterations is given by

$$\text{Regret}_T = \mathop{\mathbb{E}}_{s_0 \sim P_0} \left[\sum_{t=0}^{T-1} V_0^\star(s_0) - V_0^{\pi^t}(s_0) \right] \tag{10.1}$$

where the randomness is in the initial state distribution.

Remark 10.4 (MDP policies as MAB arms). What does this have to do with the definition of regret in the MAB setting (def. 4.3)? Here, policies are arms, and the "mean reward" is the expected total reward of a trajectory. We'll make this connection more explicit in sec. 10.3.

Theorem 10.1 (Performance of explore-then-exploit)**.** *The regret of the explore-then-exploit algorithm (def. 10.1) can be upper-bounded by*

$$\sum_{t=0}^{T-1} V_0^\star - V_0^{\pi^t} \leq |\mathcal{S}||\mathcal{A}|H. \tag{10.2}$$

(This MDP and algorithm are deterministic, assuming there is a single starting state, so the regret is not random.)

Proof. As long as every state can be reached from s_0 within a single episode, i.e. $|\mathcal{S}| \leq H$, def. 10.2 will eventually be able to explore all $|\mathcal{S}||\mathcal{A}|$ state-action pairs, adding one new transition per episode.

Let M denote the original MDP that we aim to solve. We know it will take at most $|\mathcal{S}||\mathcal{A}|$ iterations to explore the entire MDP, after which $M_\mathcal{D} = M$ and $\pi^\star_\mathcal{D}$ is the optimal policy in M, incurring no additional regret. For each "shortest-path" policy $\pi^\star_\mathcal{D}$ up until then, its value will differ from that of $\pi^\star$ by at most H, since the policies will differ by at most 1 reward at each timestep. □

10.3 Treating an unknown MDP as a MAB

We explored the exploration-exploitation tradeoff in the multi-armed bandits setting (Chapter 4). Can we apply the MAB algorithms we discovered to MDPs as well? Let us formally describe an unknown MDP as an MAB problem.

In a MAB problem, we want to find the *arm* with the highest mean reward. In an MDP, we want to find the *policy* that achieves the highest expected total reward. So if we want to apply MAB techniques to solving an MDP, it makes sense to draw an equivalence between *arms* and *policies*. We can summarize this equivalence in the following table:

Table 10.1: Treating an MDP with finite states and actions as a MAB.

MAB	MDP
K arms	$(\lvert\mathcal{A}\rvert^{\lvert\mathcal{S}\rvert})^H$ deterministic policies
unknown reward distributions ν^k	unknown trajectory distributions ρ^π
$k^\star = \arg\max_{k\in[K]} \mathbb{E}_{r\sim\nu^k}[r]$	$\pi^\star = \arg\max_{\pi\in\Pi} \mathbb{E}_{\tau\sim\rho^\pi}\left[\sum_{h=0}^{H-1} r(s_h, a_h)\right]$

MAB	MDP
pull arm k and observe reward	roll out with π and observe total reward

(For the sake of this example, assume that the MDP's reward function is stochastic, so that the MAB reward distributions are nondegenerate.)

Recall that UCB incurs regret $\widetilde{O}(\sqrt{TK})$, where T is the number of pulls and K is the number of arms. So in the MDP-as-MAB problem, using UCB for T episodes would achieve regret

$$\widetilde{O}\left(\sqrt{|\mathcal{A}|^{|\mathcal{S}|H}T}\right) \tag{10.3}$$

This scales *exponentially* in $|\mathcal{S}|$ and H, which quickly becomes intractable. Notably, this method treats each policy as entirely independent from the others, but the performance of different policies are typically correlated. We can illustrate this with the following example:

Example 10.2 (Treating an MDP as a MAB)**.** Consider a "coin MDP" with two states "heads" and "tails", two actions "Y" and "N", and a time horizon of $H = 2$. The state transition flips the coin, and doesn't depend on the action. The reward only depends on the action: Taking action Y gives reward 1, and taking action N gives reward 0.

Suppose we collect data from the two constant policies $\pi_{\mathrm{Y}}(s) = \mathrm{Y}$ and $\pi_{\mathrm{N}}(s) = \mathrm{N}$. Now we want to learn about the policy $\tilde{\pi}$ that takes action Y and then N. Do we need to collect data from $\tilde{\pi}$ to evaluate it? No: Since the reward only depends on the action, we can infer its value from our data on the policies π_{Y} and π_{N}. However, if we treat the MDP as a bandit in which $\tilde{\pi}$ is a new, unknown arm, we ignore the known correlation between the action and the reward.

10.4 Upper confidence bound value iteration

We shouldn't need to consider all $|\mathcal{A}|^{|\mathcal{S}|H}$ deterministic policies to achieve low regret. Rather, all we need to describe the optimal policy is $Q^\star$, which has $H|\mathcal{S}||\mathcal{A}|$ entries to be learned. In this section, we'll study the upper confidence bound value iteration (UCBVI) algorithm (Azar et al., 2017), which indeed achieves *polynomial* regret in $|\mathcal{S}|$, $|\mathcal{A}|$, and H.

As its name suggests, UCBVI combines the upper confidence bound (UCB) algorithm from the multi-armed bandits setting (sec. 4.7) with value iteration (VI) from the MDP setting (sec. 2.3.2):

- UCB strikes a good exploration-exploitation tradeoff in an (unknown) MAB;
- VI (what we simply call DP in the finite-horizon setting) computes the optimal value function in a known MDP (with finite states and actions).

Let us briefly review these two algorithms:

Remark 10.5 (Review of UCB). At each iteration t, for each arm k, we construct a *confidence interval* for the mean of arm k's reward distribution. We then choose the arm with the highest upper confidence bound:

$$\begin{aligned} k_{t+1} &\leftarrow \arg\max_{k\in[K]} \text{ucb}_t^k \\ \text{where } \text{ucb}_t^k &= \frac{R_t^k}{N_t^k} + \sqrt{\frac{\ln(2t/\delta)}{2N_t^k}} \end{aligned} \tag{10.4}$$

where N_t^k indicates the number of times arm k has been pulled up until time t, R_t^k indicates the total reward obtained by pulling arm k up until time t, and $\delta > 0$ controls the width of the confidence interval.

We can treat the upper confidence bound as a "proxy reward" that is the estimated mean reward plus a bonus **exploration term**. Since the size of the bonus term is proportional to our uncertainty (i.e. predicted variance) about that arm's mean, this is called an **optimistic** bonus. In UCBVI, we will extend this idea to the case of an unknown MDP $\mathcal{M}$ by adding an *exploration term* to the reward function. Then, we will use DP to solve for the optimal policy in $\widetilde{\mathcal{M}}$.

Remark 10.6 (Review of VI/DP). Value iteration (VI) is a dynamic programming (DP) algorithm for computing the optimal policy and value function in an MDP where the state transitions and reward function are known. We begin at the final timestep, where $V_H^\star(s) = 0$, and work backwards using Bellman's optimality equations (Theorem 2.5):

For $h = H-1, \dots, 0$:

$$\begin{aligned} Q_h^\star(s,a) &= r(s,a) + \mathop{\mathbb{E}}_{s'\sim P(\cdot|s,a)}[V_{h+1}^\star(s')] \\ \pi_h^\star(s) &= \arg\max_{a\in\mathcal{A}} Q_h^\star(s,a) \\ V_h^\star(s) &= Q_h^\star(s, \pi_h^\star(s)). \end{aligned} \tag{10.5}$$

Assumptions: We will consider the general case of a **time-varying** MDP where the transition and reward functions may change over time. Recall our convention that P_h is the distribution of $s_{h+1} \mid s_h, a_h$ and r_h is applied to s_h, a_h.

Definition 10.3 (UCBVI)**.** At a high level, the UCBVI algorithm can be described as follows: For $i = 0, \dots, I-1$:

1. **Modeling:** Use previously collected data to model the state transitions $\widehat{P}_0, \dots, \widehat{P}_{H-1}$ and reward functions $\hat{r}_0, \dots, \hat{r}_{H-1}$.

2. **Reward bonus:** Design a reward bonus $b_h(s,a) \in \mathbb{R}$ to encourage exploration, analogous to the UCB term.
3. **Optimistic planning:** Use VI (i.e. DP) to compute the optimal policy $\hat{\pi}$ in the modelled MDP

$$\widetilde{\mathcal{M}} = (\mathcal{S}, \mathcal{A}, \{\widehat{P}_h\}_{h\in[H]}, \{\hat{r}_h + b_h\}_{h\in[H]}, H).$$

4. **Execution:** Use $\hat{\pi}$ to collect a new trajectory.

We detail each of these steps below.

10.4.1 Modeling the transitions

Recall that we *don't know* the state transitions or reward function of the MDP we aim to solve. We seek to approximate

$$P_h(s_{h+1} \mid s_h, a_h) = \frac{\mathbb{P}(s_h, a_h, s_{h+1})}{\mathbb{P}(s_h, a_h)}, \tag{10.6}$$

where $\mathbb{P}$ denotes the true joint probabilities. We can estimate these using their sample probabilities across a set of collected transitions. That is, define

$$\begin{aligned} N_h^i(s,a,s') &:= \sum_{i'=0}^{i-1} \mathbf{1}\left\{(s_h^{i'}, a_h^{i'}, s_{h+1}^{i'}) = (s,a,s')\right\} \\ N_h^i(s,a) &:= \sum_{i'=0}^{i-1} \mathbf{1}\left\{(s_h^{i'}, a_h^{i'}) = (s,a)\right\} \end{aligned} \tag{10.7}$$

to be the number of times the tuple s, a, s' appears in the collected data, and similar for the state-action pair s, a. Then we can model

$$\widehat{P}_h^t(s' \mid s,a) = \frac{N_h^t(s,a,s')}{N_h^t(s,a)}. \tag{10.8}$$

Similarly, we can model the rewards by the sample mean in each state-action pair:

$$\hat{r}_h^t(s,a) = \frac{N_h^t(s,a)}{\sum_{t'=0}^{t-1} \mathbf{1}\left\{(s_h^i, a_h^i) = (s,a)\right\} r_h^i}. \tag{10.9}$$

This is a fairly naive, nonparametric estimator that doesn't assume any underlying structure of the MDP. We'll see how to incorporate assumptions about the MDP in the following section.

10.4.2 Reward bonus

To motivate the reward bonus term, recall how we designed the reward bonus term for UCB (sec. 4.7):

1. We used Hoeffding's inequality to bound, with high probability, how far the sample mean $\hat{\mu}_t^k$ deviated from the true mean μ^k.
2. By inverting this inequality, we obtained a $(1-\delta)$-confidence interval for the true mean, centered at our estimate.
3. To make this bound *uniform* across all timesteps $t \in [T]$, we applied the union bound and multiplied δ by a factor of T.

We'd like to do the same for UCBVI, and construct the bonus term such that $V_h^\star(s) \leq \widehat{V}_h^t(s)$ with high probability. However, our construction will be more complex than the MAB case, since $\widehat{V}_h^t(s)$ depends on the bonus $b_h^t(s,a)$ implicitly via DP. We claim that the bonus term that gives the proper bound is

$$b_h^i(s,a) = 2H\sqrt{\frac{\log(|\mathcal{S}||\mathcal{A}|HI/\delta)}{N_h^t(s,a)}}. \tag{10.10}$$

We provide a heuristic sketch of the proof in sec. B.2; see A. Agarwal et al. (2022), Section 7.3 for a full proof.

10.4.3 Performance of UCBVI

How exactly does UCBVI strike a good balance between exploration and exploitation? In UCB for MABs, the bonus exploration term is simple to interpret: It encourages the learner to take actions with a high exploration term. Here, the policy depends on the bonus term indirectly: The policy is obtained by planning in an MDP where the bonus term is added to the reward function. Note that the bonuses *propagate backwards* in DP, effectively enabling the learner to *plan to explore* unknown states. This effect takes some further interpretation.

Recall we constructed b_h^t so that, with high probability, $V_h^\star(s) \leq \widehat{V}_h^t(s)$ and so

$$V_h^\star(s) - V_h^{\pi^t}(s) \leq \widehat{V}_h^t(s) - V_h^{\pi^t}(s).$$

That is, the l.h.s. measures how suboptimal policy π^t is in the true environment, while the r.h.s. is the difference in the policy's value when acting in the modelled MDP $\widetilde{\mathcal{M}}^t$ instead of the true one $\mathcal{M}$.

If the r.h.s. is *small*, this implies that the l.h.s. difference is also small, i.e. that π^t is *exploiting* actions that are giving high reward.

If the r.h.s. is *large*, then we have overestimated the value: π^t, the optimal policy of $\widetilde{\mathcal{M}}^t$, does not perform well in the true environment $\mathcal{M}$. This indicates that one of the $b_h^t(s,a)$ terms must be large, or some $\widehat{P}_h^t(\cdot \mid s,a)$ must be inaccurate, indicating a state-action pair with a low visit count $N_h^t(s,a)$ that the learner was encouraged to explore.

It turns out that UCBVI achieves a regret of

Theorem 10.2 (UCBVI regret)**.** *The expected regret of UCBVI satisfies*

$$\mathbb{E}\left[\sum_{t=0}^{T-1}\left(V_0^\star(s_0) - V_0^{\pi^t}(s_0)\right)\right] = \widetilde{O}(H^2\sqrt{|\mathcal{S}||\mathcal{A}|I})$$

Comparing this to the UCB regret bound $\widetilde{O}(\sqrt{TK})$, where K is the number of arms of the MAB, we see that we've reduced the number of effective arms from $|\mathcal{A}|^{|\mathcal{S}|H}$ (in eq. 10.3) to $H^4|\mathcal{S}||\mathcal{A}|$, which is indeed polynomial in $|\mathcal{S}|$, $|\mathcal{A}|$, and H, as desired. This is also roughly the number of episodes it takes to achieve constant-order average regret:

$$\frac{1}{T}\,\mathbb{E}[\text{Regret}_T] = \widetilde{O}\left(\sqrt{\frac{H^4|\mathcal{S}||\mathcal{A}|}{T}}\right)$$

Note that the time-dependent transition matrix has $H|\mathcal{S}|^2|\mathcal{A}|$ entries. Assuming $H \ll |\mathcal{S}|$, this shows that it's possible to achieve low regret, and achieve a near-optimal policy, while only understanding a $1/|\mathcal{S}|$ fraction of the world's dynamics.

10.5 Linear MDPs

A polynomial dependency on $|\mathcal{S}|$ and $|\mathcal{A}|$ is manageable when the state and action spaces are small. But for large or continuous state and action spaces, even this polynomial factor will become intractable. Can we find algorithms that don't depend on $|\mathcal{S}|$ or $|\mathcal{A}|$ at all, effectively reducing the dimensionality of the MDP? In this section, we'll explore **linear MDPs**: an example of a *parameterized* MDP where the rewards and state transitions depend only on some parameter space of dimension d that is independent from $|\mathcal{S}|$ or $|\mathcal{A}|$.

Definition 10.4 (Linear MDP)**.** We assume that the transition probabilities and rewards are *linear* in some feature vector

$\phi(s,a) \in \mathbb{R}^d$:

$$P_h(s' \mid s,a) = \phi(s,a)^\top \mu_h^\star(s')$$
$$r_h(s,a) = \phi(s,a)^\top \theta_h^\star$$

Note that we can also think of $P_h(\cdot \mid s,a) = \mu_h^\star$ as an $|\mathcal{S}| \times d$ matrix, and think of $\mu_h^\star(s')$ as indexing into the s'-th row of this matrix (treating it as a column vector). Thinking of $V_{h+1}^\star$ as an $|\mathcal{S}|$-dimensional vector, this allows us to write

$$\mathop{\mathbb{E}}_{s' \sim P_h(\cdot|s,a)}[V_{h+1}^\star(s)] = (\mu_h^\star \phi(s,a))^\top V_{h+1}^\star.$$

The ϕ feature mapping can be designed to capture interactions between the state s and action a. In this book, we'll assume that the feature map $\phi : \mathcal{S} \times \mathcal{A} \to \mathbb{R}^d$ and the reward function (described by $\theta_h^\star$) are known to the learner.

10.5.1 Planning in a linear MDP

It turns out that $Q_h^\star$ is also linear with respect to this feature mapping. We can prove this by simply computing it using DP. We initialize the value function at the end of the time horizon by setting $V_H^\star(s) = 0$ for all states s. Then we iterate:

$$\begin{aligned}
Q_h^\star(s,a) &= r_h(s,a) + \mathop{\mathbb{E}}_{s' \sim P_h(\cdot|s,a)}[V_{h+1}^\star(s')] \\
&= \phi(s,a)^\top \theta_h^\star + (\mu_h^\star \phi(s,a))^\top V_{h+1}^\star \\
&= \phi(s,a)^\top \underbrace{(\theta_h^\star + (\mu_h^\star)^\top V_{h+1}^\star)}_{w_h} \\
V_h^\star(s) &= \max_a Q_h^\star(s,a) \\
\pi_h^\star(s) &= \arg\max_a Q_h^\star(s,a)
\end{aligned}$$

Exercise 10.1 (Action-value function is linear in features)**.** Show that Q_h^π is also linear with respect to $\phi(s,a)$ for any policy π.

10.5.2 UCBVI in a linear MDP

10.5.2.1 Modeling the transitions

This linear assumption on the MDP will also allow us to model the unknown dynamics $P_h^?(s' \mid s,a)$ with techniques from **supervised learning** (SL). Recall that SL is useful for estimating conditional expectations by minimizing mean squared error. We can rephrase the estimation of $P_h^?(s' \mid s,a)$ as a least-squares problem as follows: Write δ_s to denote a one-hot vector in $\mathbb{R}^{|\mathcal{S}|}$, with a 1 in the s-th entry and 0 everywhere else. Note that

$$\mathop{\mathbb{E}}_{s' \sim P_h(\cdot|s,a)}[\delta_{s'}] = P_h(\cdot \mid s,a) = \mu_h^\star \phi(s,a).$$

Furthermore, since the expectation here is linear with respect to $\phi(s,a)$, we can directly apply least-squares multi-target linear regression to construct the estimate

$$\hat{\mu} = \arg\min_{\mu \in \mathbb{R}^{|\mathcal{S}| \times d}} \sum_{t=0}^{T-1} \|\mu\phi(s_h^i, a_h^i) - \delta_{s_{h+1}^i}\|_2^2.$$

This has a well-known closed-form solution:

$$\hat{\mu}^\top = (A_h^t)^{-1} \sum_{i=0}^{t-1} \phi(s_h^i, a_h^i)\delta_{s_{h+1}^i}^\top$$
$$\text{where} \quad A_h^t = \sum_{i=0}^{t-1} \phi(s_h^i, a_h^i)\phi(s_h^i, a_h^i)^\top + \lambda I$$

where we include a λI term to ensure that the matrix A_h^t is invertible. (This can also be derived by adding a $\lambda\|\mu\|_{\mathrm{F}}^2$ regularization term to the objective.) We can directly plug in this estimate into $\widehat{P}_h^t(\cdot \mid s,a) = \hat{\mu}_h^t\phi(s,a)$.

10.5.2.2 Reward bonus

Now, to design the reward bonus, we can't apply Hoeffding's inequality anymore, since the terms no longer involve sample means of bounded random variables; Instead, we're incorporating information across different states and actions. Rather, we can construct an upper bound using *Chebyshev's inequality* in the same way we did for the LinUCB algorithm in the MAB setting sec. 4.9.1:

$$b_h^t(s,a) = \beta\sqrt{\phi(s,a)^\top (A_h^t)^{-1}\phi(s,a)}, \quad \beta = \widetilde{O}(dH).$$

Note that this isn't explicitly inversely proportional to $N_h^t(s,a)$ as in the original UCBVI bonus term eq. 10.10. Rather, it is inversely proportional to the amount that the direction $\phi(s,a)$ has been explored in the history. That is, if $A - h^t$ has a large component in the direction $\phi(s,a)$, implying that this direction is well explored, then the bonus term will be small, and vice versa.

We can now plug in these transition estimates and reward bonuses into the UCBVI algorithm def. 10.3.

Theorem 10.3 (LinUCBVI regret)**.** *The LinUCBVI algorithm achieves expected regret*

$$\mathbb{E}[Regret_T] = \mathbb{E}\left[\sum_{t=0}^{T-1} V_0^\star(s_0) - V_0^{\pi^t}(s_0)\right] \leq \widetilde{O}(H^2 d^{1.5}\sqrt{T})$$

Comparing this to our bound for UCBVI in an environment without this linear assumption, we see that we go from a sample complexity of $\widetilde{\Omega}(H^4|\mathcal{S}||\mathcal{A}|)$ to $\widetilde{\Omega}(H^4d^3)$. This new sample complexity only depends on the feature dimension and not on the state or action space of the MDP!

10.6 Key takeaways

We first discussed the explore-then-exploit algorithm (def. 10.1), a simple way to explore a deterministic MDP by visiting all state-action pairs. This is essentially a graph traversal algorithm, where each state represents an edge of the graph. We then discussed how to treat an unknown MDP as a MAB (sec. 10.3), and how this approach is inefficient since it doesn't make use of correlations between different policies. We then introduced the UCBVI algorithm (def. 10.3), the key algorithm of this chapter, which models the unknown MDP by a proxy MDP with a reward bonus term that encourages exploration. Finally, assuming that the transitions and rewards are linear with respect to a feature transformation of the state and action, we introduced the LinUCBVI algorithm (sec. 10.5.2), which has a sample complexity independent of the size of the state and action spaces. This makes it possible to scale up UCBVI to large problems that have a simple underlying structure.

10.7 Bibliographic notes and further reading

Sparse reward problems are frequent throughout reinforcement learning. The chain MDP example is from Thrun (1992). One of the most famous sparse reward problems is **Montezuma's Revenge**, one of the tasks in the popular **arcade learning environment** (ALE) benchmark of Atari 2600 games (Bellemare et al., 2013; Machado et al., 2018). These were first solved by algorithms that explicitly encourage exploration (Bellemare et al., 2016; Burda et al., 2018).

The issue of **reward hacking** is one of many possible concerns relating to AI safety. We refer the reader to Amodei et al. (2016) for an overview of such risks. Reward hacking has been empirically demonstrated in large language model training (Gao et al., 2023).

The UCBVI algorithm was first presented in Azar et al. (2017). UCBVI extends the UCB algorithm from multi-armed bandits to the MDP by estimating a model of the environment. Later work by Drago et al. (2025) improved the regret bound on UCBVI. Other model-based methods for strategic exploration have been studied at least since Schmidhuber (1991) and Meyer & Wilson (1991). UCBVI computes the reward bonus using the *count* of the number of times that state-action pair has been visited. Tang et al. (2017) surveys other such count-based exploration algorithms.

It is also possible to encourage model-free algorithms to strategically explore. Badia et al. (2020) designed a Q-learning algorithm with exploration incentives that surpassed the human baseline on the challenging Atari tasks.

Intrinsic motivation is another family of approaches to strategic exploration. In some sense, intrinsic motivation approaches are to RL as self-supervised approaches are to unsupervised learning: typically, we add some *intrinsic reward* to the objective function that encourages the policy to explore. See Schmidhuber (2010) and Aubret et al. (2019) for a recent survey on this family of methods.

We refer the reader to the survey article Ladosz et al. (2022) for further reading on exploration in RL.

References

Abbeel, P., & Ng, A. Y. (2004, July 4). Apprenticeship learning via inverse reinforcement learning. *Proceedings of the Twenty-First International Conference on Machine Learning*. International Conference on Machine Learning. https://doi.org/10.1145/1015330.1015430

Agarwal, A., Jiang, N., Kakade, S. M., & Sun, W. (2022). *Reinforcement learning: Theory and algorithms*. https://rltheorybook.github.io/rltheorybook_AJKS.pdf

Agarwal, R., Schwarzer, M., Castro, P. S., Courville, A. C., & Bellemare, M. (2021). Deep reinforcement learning at the edge of the statistical precipice. *Advances in Neural Information Processing Systems*, *34*, 29304–29320. https://proceedings.neurips.cc/paper_files/paper/2021/hash/f514cec81cb148559cf475e7426eed5e-Abstract.html

Albrecht, S. V., Christianos, F., & Schäfer, L. (2023). *Multi-agent reinforcement learning: Foundations and modern approaches*. MIT Press. https://www.marl-book.com

Allaire, J. J., Teague, C., Scheidegger, C., Xie, Y., & Dervieux, C. (2024). *Quarto* (Version 1.4) [Computer software]. https://doi.org/10.5281/zenodo.5960048

Amari, S.-I. (1998). Natural gradient works efficiently in learning. *Neural Comput.*, *10*(2), 251–276. https://doi.org/10.1162/089976698300017746

Amodei, D., Olah, C., Steinhardt, J., Christiano, P., Schulman, J., & Mané, D. (2016). *Concrete problems in AI safety*. http://arxiv.org/abs/1606.06565

Ansel, J., Yang, E., He, H., Gimelshein, N., Jain, A., Voznesensky, M., Bao, B., Bell, P., Berard, D., Burovski, E., Chauhan, G., Chourdia, A., Constable, W., Desmaison, A., DeVito, Z., Ellison, E., Feng, W., Gong, J., Gschwind, M., … Chintala, S. (2024, April). PyTorch 2: Faster machine learning through dynamic python bytecode transformation and graph compilation. *29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (ASPLOS '24)*. https://doi.org/10.1145/3620665.3640366

Antonoglou, I., Schrittwieser, J., Ozair, S., Hubert, T. K., & Silver, D. (2021, October 6). Planning in stochastic environments with a learned model. *The Tenth International Conference on Learning Representations*. International Conference on Learning Representations. https://openreview.net/forum?id=X6D9bAHhBQ1

Athans, M., & Falb, P. L. (1966). *Optimal control: An introduction to the theory and its applications*. McGraw-Hill. https://books.google.com?id=pfJHAQAAIAAJ

Aubret, A., Matignon, L., & Hassas, S. (2019, November 19). *A survey on intrinsic motivation in reinforcement learning*. https://doi.org/10.48550/arXiv.1908.06976

Auer, P. (2002). Using confidence bounds for exploitation-exploration trade-offs. *Journal of Machine Learning Research*, *3*, 397–422. https://www.jmlr.org/papers/v3/auer02a.html

Azar, M. G., Osband, I., & Munos, R. (2017). Minimax regret bounds for reinforcement learning. *Proceedings of the 34th International Conference on Machine Learning*, 263–272. https://proceedings.mlr.press/v70/azar17a.html

Babuschkin, I., Baumli, K., Bell, A., Bhupatiraju, S., Bruce, J., Buchlovsky, P., Budden, D., Cai, T., Clark, A., Danihelka, I., Dedieu, A., Fantacci, C., Godwin, J., Jones, C., Hemsley, R., Hennigan, T., Hessel, M., Hou, S., Kapturowski, S., … Viola, F. (2020). *The DeepMind JAX ecosystem* [Computer software]. http://github.com/deepmind

Badia, A. P., Piot, B., Kapturowski, S., Sprechmann, P., Vitvitskyi, A., Guo, D., & Blundell, C. (2020). Agent57: Outperforming the atari human benchmark. *ICML 2020*, 507–517. https://doi.org/10.48550/arXiv.2003.13350

Barto, A. G., Sutton, R. S., & Anderson, C. W. (1983). Neuronlike adaptive elements that can solve difficult learning control problems. *IEEE Transactions on Systems, Man, and Cybernetics*, *SMC-13*(5), 834–846. IEEE Transactions on Systems, Man, and Cybernetics. https://doi.org/10.1109/TSMC.1983.6313077

Baydin, A. G., Pearlmutter, B. A., Radul, A. A., & Siskind, J. M. (2018, February 5). *Automatic differentiation in machine learning: A survey*. https://doi.org/10.48550/arXiv.1502.05767

Beck, A., & Teboulle, M. (2003). Mirror descent and nonlinear projected subgradient methods for convex optimization. *Operations Research Letters*, *31*(3), 167–175. https://doi.org/10.1016/S0167-6377(02)00231-6

Bellemare, M. G., Naddaf, Y., Veness, J., & Bowling, M. (2013). The arcade learning environment: An evaluation platform for general agents. *Journal of Artificial Intelligence Research*, *47*, 253–279. https://doi.org/10.1613/jair.3912

Bellemare, M. G., Srinivasan, S., Ostrovski, G., Schaul, T., Saxton, D., & Munos, R. (2016). Unifying count-based exploration and intrinsic motivation. *Proceedings of the 30th International Conference on Neural Information Processing Systems*, 1479–1487.

Bellman, R. (1957). *Dynamic programming*. Princeton University Press. https://books.google.com?id=rZW4ugAACAAJ

Bellman, R. (1961). *Adaptive control processes: A guided tour*. Princeton University Press. https://books.google.com?id=POAmAAAAMAAJ

Berkovitz, L. D. (1974). *Optimal control theory*. Springer Science+Business Media LLC.

Berry, D. A., & Fristedt, B. (1985). *Bandit problems*. Springer Netherlands. https://doi.org/10.1007/978-94-015-3711-7

Bertsekas, D. P., & Tsitsiklis, J. N. (1996). *Neuro-dynamic programming*. Athena scientific.

Bhatnagar, S., Sutton, R. S., Ghavamzadeh, M., & Lee, M. (2009). Natural actor–critic algorithms. *Automatica*, *45*(11), 2471–2482. https://doi.org/10.1016/j.automatica.2009.07.008

Bishop, C. M. (2006). *Pattern recognition and machine learning*. Springer.

Boyd, S., & Vandenberghe, L. (2004). *Convex optimization*. Cambridge University Press. https://web.stanford.edu/~boyd/cvxbook/

Bradbury, J., Frostig, R., Hawkins, P., Johnson, M. J., Leary, C., Maclaurin, D., Necula, G., Paszke, A., VanderPlas, J., Wanderman-Milne, S., & Zhang, Q. (2018). *JAX: Composable transformations of python+NumPy programs* (Version 0.3.13) [Computer software]. http:

//github.com/google/jax
Burda, Y., Edwards, H., Storkey, A., & Klimov, O. (2018, September 27). Exploration by random network distillation. *The Seventh International Conference on Learning Representations*. International Conference on Learning Representations. https://openreview.net/forum?id=H1lJJnR5Ym
Clark, J., & Amodei, D. (2024, February 14). *Faulty reward functions in the wild*. OpenAI. https://openai.com/index/faulty-reward-functions/
Danihelka, I., Guez, A., Schrittwieser, J., & Silver, D. (2021, October 6). Policy improvement by planning with gumbel. *The Tenth International Conference on Learning Representations*. https://openreview.net/forum?id=bERaNdoegnO
Danilyuk, P. (2021). *A robot imitating a girl's movement* [Graphic]. https://www.pexels.com/photo/a-robot-imitating-a-girl-s-movement-8294811/
Deng, L. (2012). The MNIST database of handwritten digit images for machine learning research. *IEEE Signal Processing Magazine*, *29*(6), 141–142. IEEE Signal Processing Magazine. https://doi.org/10.1109/MSP.2012.2211477
Drago, S., Mussi, M., & Metelli, A. M. (2025, February 24). *A refined analysis of UCBVI*. https://doi.org/10.48550/arXiv.2502.17370
Fisher, R. A. (1925). *Statistical methods for research workers, 11th ed. rev.* Edinburgh.
Frans Berkelaar. (2009). *Container ship MSC davos - westerschelde - zeeland* [Graphic]. https://www.flickr.com/photos/28169156@N03/52957948820/
Gao, L., Schulman, J., & Hilton, J. (2023). Scaling laws for reward model overoptimization. *Proceedings of the 40th International Conference on Machine Learning*, *202*, 10835–10866.
Gittins, J. C. (2011). *Multi-armed bandit allocation indices* (2nd ed). Wiley.
Gleave, A., Taufeeque, M., Rocamonde, J., Jenner, E., Wang, S. H., Toyer, S., Ernestus, M., Belrose, N., Emmons, S., & Russell, S. (2022, November 22). *imitation: Clean imitation learning implementations*. https://doi.org/10.48550/arXiv.2211.11972
Goodfellow, I., Pouget-Abadie, J., Mirza, M., Xu, B., Warde-Farley, D., Ozair, S., Courville, A., & Bengio, Y. (2020). Generative adversarial networks. *Commun. ACM*, *63*(11), 139–144. https://doi.org/10.1145/3422622
GPA Photo Archive. (2017). *Robotic arm* [Graphic]. https://www.flickr.com/photos/iip-photo-archive/36123310136/
Guy, R. (2006). *Chess* [Graphic]. https://www.flickr.com/photos/romainguy/230416692/
Haarnoja, T., Tang, H., Abbeel, P., & Levine, S. (2017). Reinforcement learning with deep energy-based policies. *Proceedings of the 34th International Conference on Machine Learning - Volume 70*, 1352–1361.
Haarnoja, T., Zhou, A., Abbeel, P., & Levine, S. (2018). Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. *Proceedings of the 35th International Conference on Machine Learning*, 1861–1870. https://proceedings.mlr.press/v80/haarnoja18b.html
Hasselt, H. (2010). Double q-learning. *Advances in Neural Information Processing Systems*, *23*. https://proceedings.neurips.cc/paper_files/paper/2010/hash/091d584fced301b442654dd8c23b3fc9-Abstract.html
Hasselt, H. van, Guez, A., & Silver, D. (2016). Deep reinforcement learning with double

q-learning. *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence*, 2094–2100.

Hastie, T., Tibshirani, R., & Friedman, J. (2013). *The elements of statistical learning: Data mining, inference, and prediction*. Springer Science & Business Media. https://books.google.com?id=yPfZBwAAQBAJ

Hausknecht, M., & Stone, P. (2017, January 11). *Deep recurrent q-learning for partially observable mdps*. https://doi.org/10.48550/arXiv.1507.06527

Hessel, M., Modayil, J., van Hasselt, H., Schaul, T., Ostrovski, G., Dabney, W., Horgan, D., Piot, B., Azar, M., & Silver, D. (2018). Rainbow: Combining improvements in deep reinforcement learning. *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence and Thirtieth Innovative Applications of Artificial Intelligence Conference and Eighth AAAI Symposium on Educational Advances in Artificial Intelligence*, 3215–3222.

Ho, J., & Ermon, S. (2016). Generative adversarial imitation learning. *Proceedings of the 30th International Conference on Neural Information Processing Systems*, 4572–4580.

Ivanov, S., & D'yakonov, A. (2019, July 6). *Modern deep reinforcement learning algorithms*. https://doi.org/10.48550/arXiv.1906.10025

James, G., Witten, D., Hastie, T., Tibshirani, R., & Taylor, J. (2023). *An introduction to statistical learning: With applications in python*. Springer International Publishing. https://doi.org/10.1007/978-3-031-38747-0

Kakade, S. M. (2001). A natural policy gradient. *Advances in Neural Information Processing Systems*, *14*. https://proceedings.neurips.cc/paper_files/paper/2001/hash/4b86abe48d358ecf194c56c69108433e-Abstract.html

Kakade, S., & Langford, J. (2002). Approximately optimal approximate reinforcement learning. *Proceedings of the Nineteenth International Conference on Machine Learning*, 267–274.

Keviczky, L., Bars, R., Hetthéssy, J., & Bányász, C. (2019). *Control engineering*. Springer. https://doi.org/10.1007/978-981-10-8297-9

Kochenderfer, M. J., Wheeler, T. A., & Wray, K. H. (2022). *Algorithms for decision making*. https://mitpress.mit.edu/9780262047012/algorithms-for-decision-making/

Ladosz, P., Weng, L., Kim, M., & Oh, H. (2022). Exploration in deep reinforcement learning: A survey. *Inf. Fusion*, *85*(C), 1–22. https://doi.org/10.1016/j.inffus.2022.03.003

Lai, T. L., & Robbins, H. (1985). Asymptotically efficient adaptive allocation rules. *Advances in Applied Mathematics*, *6*(1), 4–22. https://doi.org/10.1016/0196-8858(85)90002-8

Lambert, N. (2024). *Reinforcement learning from human feedback*. Online. https://rlhfbook.com

Lewis, F. L., Vrabie, D. L., & Syrmos, V. L. (2012). *Optimal control* (3rd ed). John Wiley & Sons. https://doi.org/10.1002/9781118122631

Li, L., Chu, W., Langford, J., & Schapire, R. E. (2010). A contextual-bandit approach to personalized news article recommendation. *Proceedings of the 19th International Conference on World Wide Web*, 661–670. https://doi.org/10.1145/1772690.1772758

Li, S. E. (2023). *Reinforcement learning for sequential decision and optimal control*. Springer Nature. https://doi.org/10.1007/978-981-19-7784-8

Li, Y. (2018). *Deep reinforcement learning*. https://doi.org/10.48550/arXiv.1810.06339

Lin, L.-J. (1992). Self-improving reactive agents based on reinforcement learning, planning

and teaching. *Machine Learning*, *8*(3), 293–321. https://doi.org/10.1007/BF00992699
Ljapunov, A. M., & Fuller, A. T. (1992). *The general problem of the stability of motion*. Taylor & Francis.
Lyapunov, A. M. (1892). *The general problem of the stability of motion*. University of Kharkov.
MacFarlane, A. (1979). The development of frequency-response methods in automatic control [perspectives]. *IEEE Transactions on Automatic Control*, *24*(2), 250–265. IEEE Transactions on Automatic Control. https://doi.org/10.1109/TAC.1979.1101978
Machado, M. C., Bellemare, M. G., Talvitie, E., Veness, J., Hausknecht, M., & Bowling, M. (2018). Revisiting the arcade learning environment: Evaluation protocols and open problems for general agents. *Journal of Artificial Intelligence Research*, *61*, 523–562. https://doi.org/10.1613/jair.5699
Maei, H., Szepesvári, C., Bhatnagar, S., Precup, D., Silver, D., & Sutton, R. S. (2009). Convergent temporal-difference learning with arbitrary smooth function approximation. *Advances in Neural Information Processing Systems*, *22*. https://papers.nips.cc/paper_files/paper/2009/hash/3a15c7d0bbe60300a39f76f8a5ba6896-Abstract.html
Mahadevan, S., Giguere, S., & Jacek, N. (2013). Basis adaptation for sparse nonlinear reinforcement learning. *Proceedings of the AAAI Conference on Artificial Intelligence*, *27*(1, 1), 654–660. https://doi.org/10.1609/aaai.v27i1.8665
Mahadevan, S., & Liu, B. (2012). Sparse q-learning with mirror descent. *Proceedings of the Twenty-Eighth Conference on Uncertainty in Artificial Intelligence*, 564–573.
Mannor, S., Mansour, Y., & Tamar, A. (2024). *Reinforcement learning: foundations*. https://sites.google.com/view/rlfoundations/home
Marbach, P., & Tsitsiklis, J. N. (2001). Simulation-based optimization of markov reward processes. *IEEE Transactions on Automatic Control*, *46*(2), 191–209. IEEE Transactions on Automatic Control. https://doi.org/10.1109/9.905687
Martens, J., & Grosse, R. (2015). Optimizing neural networks with kronecker-factored approximate curvature. *Proceedings of the 32nd International Conference on Machine Learning*, 2408–2417. https://proceedings.mlr.press/v37/martens15.html
Maxwell, J. C. (1867). On governors. *Proceedings of the Royal Society of London*, *16*, 270–283. https://www.jstor.org/stable/112510
Mayr, E. (1970). *Populations, species and evolution: An abridgment of animal species and evolution*. Belknap Press of Harvard University Press.
Meurer, A., Smith, C. P., Paprocki, M., Čertík, O., Kirpichev, S. B., Rocklin, M., Kumar, A., Ivanov, S., Moore, J. K., Singh, S., Rathnayake, T., Vig, S., Granger, B. E., Muller, R. P., Bonazzi, F., Gupta, H., Vats, S., Johansson, F., Pedregosa, F., … Scopatz, A. (2017). SymPy: Symbolic computing in python. *PeerJ Computer Science*, *3*, e103. https://doi.org/10.7717/peerj-cs.103
Meyer, J.-A., & Wilson, S. W. (1991). A possibility for implementing curiosity and boredom in model-building neural controllers. In *From Animals to Animats: Proceedings of the First International Conference on Simulation of Adaptive Behavior* (pp. 222–227). From Animals to Animats: Proceedings of the First International Conference on Simulation of Adaptive Behavior. MIT Press. https://ieeexplore.ieee.org/document/6294131
Mnih, V., Badia, A. P., Mirza, M., Graves, A., Lillicrap, T., Harley, T., Silver, D., &

Kavukcuoglu, K. (2016). Asynchronous methods for deep reinforcement learning. *Proceedings of The 33rd International Conference on Machine Learning*, 1928–1937. https://proceedings.mlr.press/v48/mniha16.html

Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., & Riedmiller, M. A. (2013). Playing atari with deep reinforcement learning. *CoRR*, *abs/1312.5602*. http://arxiv.org/abs/1312.5602

Munos, R., Stepleton, T., Harutyunyan, A., & Bellemare, M. G. (2016). Safe and efficient off-policy reinforcement learning. *Proceedings of the 30th International Conference on Neural Information Processing Systems*, 1054–1062.

Murphy, K. (2025, March 24). *Reinforcement learning: A comprehensive overview*. https://doi.org/10.48550/arXiv.2412.05265

Negative Space. (2015). *Photo of commercial district during dawn* [Graphic]. https://www.pexels.com/photo/photo-of-commercial-district-during-dawn-34639/

Nemirovskij, A. S., Judin, D. B., Dawson, E. R., & Nemirovskij, A. S. (1983). *Problem complexity and method efficiency in optimization*. Wiley.

Ng, A. Y., & Russell, S. J. (2000). Algorithms for inverse reinforcement learning. *Proceedings of the Seventeenth International Conference on Machine Learning*, 663–670. https://ai.stanford.edu/~ang/papers/icml00-irl.pdf

Nielsen, M. A. (2015). *Neural networks and deep learning*. Determination Press. http://neuralnetworksanddeeplearning.com/

Nocedal, J., & Wright, S. J. (2006). *Numerical optimization* (2nd ed). Springer.

OpenAI. (2022, November 30). *Introducing ChatGPT*. OpenAI News. https://openai.com/index/chatgpt/

Orsini, M., Raichuk, A., Hussenot, L., Vincent, D., Dadashi, R., Girgin, S., Geist, M., Bachem, O., Pietquin, O., & Andrychowicz, M. (2021). What matters for adversarial imitation learning? *Proceedings of the 35th International Conference on Neural Information Processing Systems*, 14656–14668.

Ouyang, L., Wu, J., Jiang, X., Almeida, D., Wainwright, C. L., Mishkin, P., Zhang, C., Agarwal, S., Slama, K., Ray, A., Schulman, J., Hilton, J., Kelton, F., Miller, L., Simens, M., Askell, A., Welinder, P., Christiano, P., Leike, J., & Lowe, R. (2022). Training language models to follow instructions with human feedback. *Proceedings of the 36th International Conference on Neural Information Processing Systems*, 27730–27744.

Peters, J., & Schaal, S. (2008). Natural actor-critic. *Neurocomputing*, *71*(7), 1180–1190. https://doi.org/10.1016/j.neucom.2007.11.026

Peters, J., Vijayakumar, S., & Schaal, S. (2005). Natural actor-critic. In J. Gama, R. Camacho, P. B. Brazdil, A. M. Jorge, & L. Torgo (Eds.), *Machine Learning: ECML 2005* (pp. 280–291). Springer. https://doi.org/10.1007/11564096_29

Piot, B., Geist, M., & Pietquin, O. (2017). Bridging the gap between imitation learning and inverse reinforcement learning. *IEEE Transactions on Neural Networks and Learning Systems*, *28*(8), 1814–1826. IEEE Transactions on Neural Networks and Learning Systems. https://doi.org/10.1109/TNNLS.2016.2543000

Pixabay. (2016a). *20 mg label blister pack* [Graphic]. https://www.pexels.com/photo/20-mg-label-blister-pack-208512/

Pixabay. (2016b). *Coins on brown wood* [Graphic]. https://www.pexels.com/photo/coins-on-brown-wood-210600/

Plaat, A. (2022). *Deep reinforcement learning*. Springer Nature. https://doi.org/10.1007/978-981-19-0638-1

Pomerleau, D. A. (1991). Efficient training of artificial neural networks for autonomous navigation. *Neural Computation*, *3*(1), 88–97. Neural Computation. https://doi.org/10.1162/neco.1991.3.1.88

Powell, W. B. (2022). *Reinforcement learning and stochastic optimization: A unified framework for sequential decisions*. Wiley.

Puterman, M. L. (1994). *Markov decision processes: Discrete stochastic dynamic programming*. Wiley.

Ramachandran, D., & Amir, E. (2007). Bayesian inverse reinforcement learning. *Proceedings of the 20th International Joint Conference on Artifical Intelligence*, 2586–2591.

Rao, A., & Jelvis, T. (2022). *Foundations of reinforcement learning with applications in finance*. Chapman and Hall/CRC. https://doi.org/10.1201/9781003229193

Ratliff, N. D., Bagnell, J. A., & Zinkevich, M. A. (2006). Maximum margin planning. *Proceedings of the 23rd International Conference on Machine Learning*, 729–736. https://doi.org/10.1145/1143844.1143936

Robbins, H. (1952). Some aspects of the sequential design of experiments. *Bulletin of the American Mathematical Society*, *58*(5), 527–535. https://projecteuclid.org/journals/bulletin-of-the-american-mathematical-society/volume-58/issue-5/Some-aspects-of-the-sequential-design-of-experiments/bams/1183517370.full

Ross, S., Gordon, G. J., & Bagnell, J. (2010, November 2). *A reduction of imitation learning and structured prediction to no-regret online learning*. International Conference on Artificial Intelligence and Statistics. https://www.semanticscholar.org/paper/A-Reduction-of-Imitation-Learning-and-Structured-to-Ross-Gordon/79ab3c49903ec8cb339437ccf5cf998607fc313e

Russell, S. (1998). Learning agents for uncertain environments (extended abstract). *Proceedings of the Eleventh Annual Conference on Computational Learning Theory*, 101–103. https://doi.org/10.1145/279943.279964

Russell, S. J., & Norvig, P. (2021). *Artificial intelligence: A modern approach* (Fourth edition). Pearson.

Schmidhuber, J. (1991). Curious model-building control systems. *[Proceedings] 1991 IEEE International Joint Conference on Neural Networks*, 1458–1463 vol.2. https://doi.org/10.1109/IJCNN.1991.170605

Schmidhuber, J. (2010). Formal theory of creativity, fun, and intrinsic motivation (1990–2010). *IEEE Transactions on Autonomous Mental Development*, *2*(3), 230–247. IEEE Transactions on Autonomous Mental Development. https://doi.org/10.1109/TAMD.2010.2056368

Schrittwieser, J., Antonoglou, I., Hubert, T., Simonyan, K., Sifre, L., Schmitt, S., Guez, A., Lockhart, E., Hassabis, D., Graepel, T., Lillicrap, T., & Silver, D. (2020). Mastering atari, go, chess and shogi by planning with a learned model. *Nature*, *588*(7839, 7839), 604–609. https://doi.org/10.1038/s41586-020-03051-4

Schulman, J., Levine, S., Abbeel, P., Jordan, M., & Moritz, P. (2015). Trust region policy optimization. *Proceedings of the 32nd International Conference on Machine Learning*, 1889–1897. https://proceedings.mlr.press/v37/schulman15.html

Schulman, J., Moritz, P., Levine, S., Jordan, M. I., & Abbeel, P. (2016). High-dimensional continuous control using generalized advantage estimation. In Y. Bengio & Y. LeCun (Eds.), *4th international conference on learning representations*. http://arxiv.org/abs/1506.02438

Schulman, J., Wolski, F., Dhariwal, P., Radford, A., & Klimov, O. (2017, August 28). *Proximal policy optimization algorithms*. https://doi.org/10.48550/arXiv.1707.06347

Schultz, W., Dayan, P., & Montague, P. R. (1997). A neural substrate of prediction and reward. *Science*, *275*(5306), 1593–1599. https://doi.org/10.1126/science.275.5306.1593

Shao, Z., Wang, P., Zhu, Q., Xu, R., Song, J., Bi, X., Zhang, H., Zhang, M., Li, Y. K., Wu, Y., & Guo, D. (2024, April 27). *DeepSeekMath: Pushing the limits of mathematical reasoning in open language models*. https://doi.org/10.48550/arXiv.2402.03300

Silver, D., Huang, A., Maddison, C. J., Guez, A., Sifre, L., van den Driessche, G., Schrittwieser, J., Antonoglou, I., Panneershelvam, V., Lanctot, M., Dieleman, S., Grewe, D., Nham, J., Kalchbrenner, N., Sutskever, I., Lillicrap, T., Leach, M., Kavukcuoglu, K., Graepel, T., & Hassabis, D. (2016). Mastering the game of go with deep neural networks and tree search. *Nature*, *529*(7587, 7587), 484–489. https://doi.org/10.1038/nature16961

Silver, D., Hubert, T., Schrittwieser, J., Antonoglou, I., Lai, M., Guez, A., Lanctot, M., Sifre, L., Kumaran, D., Graepel, T., Lillicrap, T., Simonyan, K., & Hassabis, D. (2018). A general reinforcement learning algorithm that masters chess, shogi, and go through self-play. *Science*, *362*(6419), 1140–1144. https://doi.org/10.1126/science.aar6404

Silver, D., Lever, G., Heess, N., Degris, T., Wierstra, D., & Riedmiller, M. (2014). Deterministic policy gradient algorithms. *Proceedings of the 31st International Conference on Machine Learning*, 387–395. https://proceedings.mlr.press/v32/silver14.html

Silver, D., Schrittwieser, J., Simonyan, K., Antonoglou, I., Huang, A., Guez, A., Hubert, T., Baker, L., Lai, M., Bolton, A., Chen, Y., Lillicrap, T., Hui, F., Sifre, L., van den Driessche, G., Graepel, T., & Hassabis, D. (2017). Mastering the game of go without human knowledge. *Nature*, *550*(7676, 7676), 354–359. https://doi.org/10.1038/nature24270

Silver, D., Singh, S., Precup, D., & Sutton, R. S. (2021). Reward is enough. *Artificial Intelligence*, *299*, 103535. https://doi.org/10.1016/j.artint.2021.103535

Stigler, S. M. (2003). *The history of statistics: The measurement of uncertainty before 1900* (9. print). Belknap Pr. of Harvard Univ. Pr.

Sussman, G. J., Wisdom, J., & Farr, W. (2013). *Functional differential geometry*. The MIT Press.

Sutton, R. S. (1984). *Temporal credit assignment in reinforcement learning* [PhD thesis]. University of Massachusetts Amherst.

Sutton, R. S. (1988). Learning to predict by the methods of temporal differences. *Machine Learning*, *3*(1), 9–44. https://doi.org/10.1007/BF00115009

Sutton, R. S., & Barto, A. G. (2018). *Reinforcement learning: An introduction* (Second edition). The MIT Press. http://incompleteideas.net/book/RLbook2020trimmed.pdf

Sutton, R. S., McAllester, D., Singh, S., & Mansour, Y. (1999). Policy gradient methods for

reinforcement learning with function approximation. *Proceedings of the 13th International Conference on Neural Information Processing Systems*, 1057–1063.

Szepesvári, C. (2010). *Algorithms for reinforcement learning*. Springer International Publishing. https://doi.org/10.1007/978-3-031-01551-9

Tang, H., Houthooft, R., Foote, D., Stooke, A., Xi Chen, O., Duan, Y., Schulman, J., DeTurck, F., & Abbeel, P. (2017). #Exploration: A study of count-based exploration for deep reinforcement learning. *Advances in Neural Information Processing Systems*, *30*. https://proceedings.neurips.cc/paper_files/paper/2017/hash/3a20f62a0af1aa152670bab3c602feed-Abstract.html

Thompson, W. R. (1933). On the likelihood that one unknown probability exceeds another in view of the evidence of two samples. *Biometrika*, *25*(3/4), 285–294. https://doi.org/10.2307/2332286

Thompson, W. R. (1935). On the theory of apportionment. *American Journal of Mathematics*, *57*(2), 450–456. https://doi.org/10.2307/2371219

Thorndike, E. L. (1911). *Animal intelligence: Experimental studies* (pp. viii, 297). Macmillan Press. https://doi.org/10.5962/bhl.title.55072

Thrun, S. B. (1992). *Efficient exploration in reinforcement learning* [Technical Report]. Carnegie Mellon University.

Turing, A. (1948). Intelligent machinery. *National Physical Laboratory*. https://weightagnostic.github.io/papers/turing1948.pdf

van Hasselt, H., Doron, Y., Strub, F., Hessel, M., Sonnerat, N., & Modayil, J. (2018, December 6). *Deep reinforcement learning and the deadly triad*. https://doi.org/10.48550/arXiv.1812.02648

Vapnik, V. N. (2000). *The nature of statistical learning theory*. Springer. https://doi.org/10.1007/978-1-4757-3264-1

Vershynin, R. (2018). *High-dimensional probability: An introduction with applications in data science*. Cambridge University Press. https://books.google.com?id=NDdqDwAAQBAJ

Wald, A. (1949). Statistical decision functions. *The Annals of Mathematical Statistics*, *20*(2), 165–205. https://doi.org/10.1214/aoms/1177730030

Wang, Z., Bapst, V., Heess, N., Mnih, V., Munos, R., Kavukcuoglu, K., & Freitas, N. de. (2017, February 6). Sample efficient actor-critic with experience replay. *5th International Conference on Learning Representations*. https://openreview.net/forum?id=HyM25Mqel

Wang, Z., Schaul, T., Hessel, M., Hasselt, H., Lanctot, M., & Freitas, N. (2016). Dueling network architectures for deep reinforcement learning. *Proceedings of The 33rd International Conference on Machine Learning*, 1995–2003. https://proceedings.mlr.press/v48/wangf16.html

Williams, R. J. (1992). Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine Learning*, *8*(3), 229–256. https://doi.org/10.1007/BF00992696

Witten, I. H. (1977). An adaptive optimal controller for discrete-time markov environments. *Information and Control*, *34*(4), 286–295. https://doi.org/10.1016/S0019-9958(77)90354-0

Wu, Y., Mansimov, E., Grosse, R. B., Liao, S., & Ba, J. (2017). Scalable trust-region method for deep reinforcement learning using kronecker-factored approximation. *Advances in Neu-*

ral Information Processing Systems, *30*. https://proceedings.neurips.cc/paper/2017/hash/361440528766bbaaaa1901845cf4152b-Abstract.html

Ye, W., Liu, S., Kurutach, T., Abbeel, P., & Gao, Y. (2021). Mastering atari games with limited data. *NeurIPS 2021*, 25476–25488. https://doi.org/10.48550/arXiv.2111.00210

Zare, M., Kebria, P. M., Khosravi, A., & Nahavandi, S. (2024). A survey of imitation learning: Algorithms, recent developments, and challenges. *IEEE Transactions on Cybernetics*, *54*(12), 7173–7186. IEEE Transactions on Cybernetics. https://doi.org/10.1109/TCYB.2024.3395626

Ziebart, B. D., Bagnell, J. A., & Dey, A. K. (2010). Modeling interaction via the principle of maximum causal entropy. *Proceedings of the 27th International Conference on International Conference on Machine Learning*, 1255–1262.

Ziebart, B. D., Maas, A., Bagnell, J. A., & Dey, A. K. (2008). Maximum entropy inverse reinforcement learning. *Proceedings of the 23rd National Conference on Artificial Intelligence - Volume 3*, 1433–1438.

Ziegler, D. M., Stiennon, N., Wu, J., Brown, T. B., Radford, A., Amodei, D., Christiano, P., & Irving, G. (2020, January 8). *Fine-Tuning Language Models from Human Preferences*. https://doi.org/10.48550/arXiv.1909.08593

A Background

A.1 O notation

Throughout this chapter and the rest of the book, we will describe the asymptotic behaviour of a function using O notation.

For two functions $f(t)$ and $g(t)$, we say that $f(t) \leq O(g(t))$ if f is asymptotically upper bounded by g. Formally, this means that there exists some constant $C > 0$ such that $f(t) \leq C \cdot g(t)$ for all t past some point t_0.

We say $f(t) < o(g(t))$ if asymptotically f grows strictly slower than g. Formally, this means that for *any* scalar $C > 0$, there exists some t_0 such that $f(t) \leq C \cdot g(t)$ for all $t > t_0$. Equivalently, we say $f(t) < o(g(t))$ if $\lim_{t\to\infty} f(t)/g(t) = 0$.

$f(t) = \Theta(g(t))$ means that f and g grow at the same rate asymptotically. That is, $f(t) \leq O(g(t))$ and $g(t) \leq O(f(t))$.

Finally, we use $f(t) \geq \Omega(g(t))$ to mean that $g(t) \leq O(f(t))$, and $f(t) > \omega(g(t))$ to mean that $g(t) < o(f(t))$.

We also use the notation $\widetilde{O}(g(t))$ to hide logarithmic factors. That is, $f(t) = \widetilde{O}(g(t))$ if there exists some constant C such that $f(t) \leq C \cdot g(t) \cdot \log^k(t)$ for some k and all t.

Occasionally, we will also use $O(f(t))$ (or one of the other symbols) as shorthand to manipulate function classes. For example, we might write $O(f(t)) + O(g(t)) = O(f(t) + g(t))$ to mean that the sum of two functions in $O(f(t))$ and $O(g(t))$ is in $O(f(t) + g(t))$.

A.2 Union bound

Theorem A.1 (Union bound)**.** *Consider a set of events $A_0, \ldots, A_{N-1}$. Then*

$$\mathbb{P}\left(\bigcup_{n=0}^{N-1} A_n\right) \leq \sum_{n=0}^{N-1} \mathbb{P}(A_n). \tag{A.1}$$

In particular, if $\mathbb{P}(A_n) \geq 1 - \delta$ for each $n \in [N]$, we have

$$\mathbb{P}\left(\bigcap_{n=0}^{N-1} A_n\right) \geq 1 - N\delta. \tag{A.2}$$

In other words, if each event A_n has a small probability δ of "failure", then to get the probability that there are *any* failures out of all N events, we multiply the failure probability by N.

B Proofs

B.1 LQR proof

1. We'll compute $V_H^\star$ (at the end of the horizon) as our base case.
2. Then we'll work step-by-step backwards in time, using $V_{h+1}^\star$ to compute $Q_h^\star$, $\pi_h^\star$, and $V_h^\star$.

Base case:

At the final timestep, there are no possible actions to take, and so $V_H^\star(x) = c(x) = x^\top Q x$. Thus $V_H^\star(x) = x^\top P_H x + p_H$ where $P_H = Q$ and $p_H = 0$.

Inductive hypothesis:

We seek to show that the inductive step holds for both theorems: If $V_{h+1}^\star(x)$ is an upward-curved quadratic, then $V_h^\star(x)$ must also be an upward-curved quadratic, and $\pi_h^\star(x)$ must be linear. We'll break this down into the following steps:

1. Show that $Q_h^\star(x, u)$ is an upward-curved quadratic (in both x and u).
2. Derive the optimal policy $\pi_h^\star(x) = \arg\min_u Q_h^\star(x, u)$ and show that it's linear.
3. Show that $V_h^\star(x)$ is an upward-curved quadratic.

We first assume the inductive hypothesis that our theorems are true at time $h + 1$. That is,

$$V_{h+1}^\star(x) = x^\top P_{h+1} x + p_{h+1} \quad \forall x \in \mathcal{S}.$$

Lemma B.1 ($Q_h^\star(x, u)$ is an upward-curved quadratic)**.** *Let us decompose $Q_h^\star : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$ into the immediate reward plus the expected cost-to-go:*

$$Q_h^\star(x, u) = c(x, u) + \mathop{\mathbb{E}}_{x' \sim f(x,u,w_{h+1})}[V_{h+1}^\star(x')].$$

Recall $c(x, u) := x^\top Q x + u^\top R u$. Let's consider the expectation over the next timestep. The only randomness in the dynamics comes from the noise $w_{h+1} \sim \mathcal{N}(0, \sigma^2 I)$, so we can expand the expectation as:

$$
\begin{aligned}
&\mathop{\mathbb{E}}_{x'}[V^\star_{h+1}(x')] \\
&= \mathop{\mathbb{E}}_{w_{h+1}}[V^\star_{h+1}(Ax + Bu + w_{h+1})] && \textit{definition of } f \\
&= \mathop{\mathbb{E}}_{w_{h+1}}[(Ax + Bu + w_{h+1})^\top P_{h+1}(Ax + Bu + w_{h+1}) + p_{h+1}]. && \textit{inductive hypothesis}
\end{aligned}
$$

Summing and combining like terms, we get

$$
\begin{aligned}
Q^\star_h(x,u) &= x^\top Q x + u^\top R u + \mathop{\mathbb{E}}_{w_{h+1}}[(Ax + Bu + w_{h+1})^\top P_{h+1}(Ax + Bu + w_{h+1}) + p_{h+1}] \\
&= x^\top (Q + A^\top P_{h+1} A)x + u^\top (R + B^\top P_{h+1} B)u + 2x^\top A^\top P_{h+1} B u \\
&\quad + \mathop{\mathbb{E}}_{w_{h+1}}[w_{h+1}^\top P_{h+1} w_{h+1}] + p_{h+1}.
\end{aligned}
$$

Note that the terms that are linear in w_h have mean zero and vanish. Now consider the remaining expectation over the noise. By expanding out the product and using linearity of expectation, we can write this out as

$$
\begin{aligned}
\mathop{\mathbb{E}}_{w_{h+1}}[w_{h+1}^\top P_{h+1} w_{h+1}] &= \sum_{i=1}^{d}\sum_{j=1}^{d}(P_{h+1})_{ij} \mathop{\mathbb{E}}_{w_{h+1}}[(w_{h+1})_i (w_{h+1})_j] \\
&= \sigma^2 \mathrm{Tr}(P_{h+1})
\end{aligned}
$$

B.1.0.1 Quadratic forms

When solving quadratic forms*, i.e. expressions of the form $x^\top A x$, it's often helpful to consider the terms on the diagonal ($i = j$) separately from those off the diagonal.*

In this case, the expectation of each diagonal term becomes

$$
(P_{h+1})_{ii}\, \mathbb{E}(w_{h+1})_i^2 = \sigma^2 (P_{h+1})_{ii}. \tag{B.1}
$$

Off the diagonal, since the elements of w_{h+1} are independent, the expectation factors, and since each element has mean zero, the term vanishes:

$$
(P_{h+1})_{ij}\, \mathbb{E}[(w_{h+1})_i]\, \mathbb{E}[(w_{h+1})_j] = 0. \tag{B.2}
$$

Thus, the only terms left are the ones on the diagonal, so the sum of these can be expressed as the trace of $\sigma^2 P_{h+1}$:

$$\mathop{\mathbb{E}}_{w_{h+1}}[w_{h+1}^\top P_{h+1} w_{h+1}] = \sigma^2 \mathrm{Tr}(P_{h+1}). \tag{B.3}$$

Substituting this back into the expression for $Q_h^\star$, we have:

$$\begin{aligned} Q_h^\star(x,u) = x^\top (Q + A^\top P_{h+1} A)x + u^\top (R + B^\top P_{h+1} B)u + 2x^\top A^\top P_{h+1} B u \\ + \sigma^2 \mathrm{Tr}(P_{h+1}) + p_{h+1}. \end{aligned} \tag{B.4}$$

As we hoped, this expression is quadratic in x and u. Furthermore, we'd like to show that it also curves upwards *with respect to u so that its minimum with respect to u is well-defined. We can do this by noting that the* ***Hessian matrix*** *of second derivatives is positive definite:*

$$\nabla_{uu} Q_h^\star(x,u) = R + B^\top P_{h+1} B$$

Since R is sym. p.d. (def. 3.5), and P_{h+1} is sym. p.d. (by the inductive hypothesis), this sum must also be sym. p.d., and so $Q_h^\star$ is indeed an upward-curved quadratic with respect to u. (If this isn't clear, try proving it as an exercise.) The proof of its upward curvature with respect to x is equivalent.

Lemma B.2 ($\pi_h^\star$ is linear)**.** *Since $Q_h^\star$ is an upward-curved quadratic, finding its minimum over u is easy: we simply set the gradient with respect to u equal to zero and solve for u. First, we calculate the gradient:*

$$\begin{aligned} \nabla_u Q_h^\star(x,u) &= \nabla_u [u^\top (R + B^\top P_{h+1} B)u + 2x^\top A^\top P_{h+1} B u] \\ &= 2(R + B^\top P_{h+1} B)u + 2(x^\top A^\top P_{h+1} B)^\top \end{aligned}$$

Setting this to zero, we get

$$\begin{aligned} 0 &= (R + B^\top P_{h+1} B)\pi_h^\star(x) + B^\top P_{h+1} A x \\ \pi_h^\star(x) &= (R + B^\top P_{h+1} B)^{-1}(-B^\top P_{h+1} A x) \\ &= -K_h x, \end{aligned}$$

where

$$K_h = (R + B^\top P_{h+1} B)^{-1} B^\top P_{h+1} A. \tag{B.5}$$

Note that this optimal policy doesn't depend on the starting distribution μ_0. It's also fully ***deterministic*** *and isn't affected by the noise terms $w_0, \ldots, w_{H-1}$.*

Lemma B.3 (The value function is an upward-curved quadratic)**.** *Using the identity* $V_h^\star(x) = Q_h^\star(x, \pi^\star(x))$*, we have:*

$$\begin{aligned} V_h^\star(x) &= Q_h^\star(x, \pi^\star(x)) \\ &= x^\top (Q + A^\top P_{h+1} A)x + (-K_h x)^\top (R + B^\top P_{h+1} B)(-K_h x) + 2x^\top A^\top P_{h+1} B(-K_h x) \\ &\quad + \mathrm{Tr}(\sigma^2 P_{h+1}) + p_{h+1} \end{aligned}$$

Note that with respect to x*, this is the sum of a quadratic term and a constant, which is exactly what we were aiming for! The scalar term is clearly*

$$p_h = \mathrm{Tr}(\sigma^2 P_{h+1}) + p_{h+1}. \tag{B.6}$$

We can simplify the quadratic term by substituting in K_h *from eq. B.5. Notice that when we do this, the* $(R + B^\top P_{h+1} B)$ *term in the expression is cancelled out by its inverse, and the remaining terms combine to give the* ***Riccati equation****:*

$$P_h = Q + A^\top P_{h+1} A - A^\top P_{h+1} B(R + B^\top P_{h+1} B)^{-1} B^\top P_{h+1} A. \tag{B.7}$$

It remains to prove that $V_h^\star$ curves upwards*, that is, that* P_h *is sym. p.d. We will use the following fact about* ***Schur complements:***

Lemma B.4 (Positive definiteness of Schur complements)**.** *Let*

$$D = \begin{pmatrix} A & B \\ B^\top & C \end{pmatrix} \tag{B.8}$$

be a symmetric $(m+n) \times (m+n)$ *block matrix, where* $A \in \mathbb{R}^{m \times m}, B \in \mathbb{R}^{m \times n}, C \in \mathbb{R}^{n \times n}$*. The* ***Schur complement*** *of* A *is denoted*

$$D/A = C - B^\top A^{-1} B. \tag{B.9}$$

Schur complements have various uses in linear algebra and numerical computation.

A useful fact for us is that if A *is positive* definite*, then* D *is positive* semidefinite *if and only if* D/A *is positive* semidefinite*.*

Let P denote P_{h+1} for brevity. We already know Q is sym. p.d., so it suffices to show that

$$S = P - PB(R + B^\top PB)^{-1}B^\top P \tag{B.10}$$

is p.s.d. (positive semidefinite), since left- and right- multiplying by $A^\top$ and A respectively preserves p.s.d. We note that S is the Schur complement $D/(R + B^\top PB)$, where

$$D = \begin{pmatrix} R + B^\top PB & B^\top P \\ PB & P \end{pmatrix}.$$

Thus we must show that D is p.s.d.. This can be seen by computing

$$\begin{aligned} \begin{pmatrix} y^\top & z^\top \end{pmatrix} D \begin{pmatrix} y \\ z \end{pmatrix} &= y^\top Ry + y^\top B^\top PBy + 2y^\top B^\top Pz + z^\top Pz \\ &= y^\top Ry + (By + z)^\top P(By + z) \\ &> 0. \end{aligned}$$

Since $R + B^\top PB$ is sym. p.d. and D is p.s.d., then $S = D/(R + B^\top PB)$ must be p.s.d., and $P_h = Q + ASA^\top$ must be sym. p.d.

Now we've shown that $V_h^\star(x) = x^\top P_h x + p_h$, where P_h is sym. p.d., proving the inductive hypothesis and completing the proof of Theorem 3.3 and Theorem 3.2.

B.2 UCBVI reward bonus proof

We aim to show that, with high probability,

$$V_h^\star(s) \le \widehat{V}_h^t(s) \quad \forall t \in [T], h \in [H], s \in \mathcal{S}.$$

We'll do this by bounding the error incurred at each step of DP. Recall that DP solves for $\widehat{V}_h^t(s)$ recursively as follows:

$$\widehat{V}_h^t(s) = \max_{a \in \mathcal{A}} \left[\tilde{r}_h^t(s,a) + \mathop{\mathbb{E}}_{s' \sim \widehat{P}_h^t(\cdot|s,a)} \left[\widehat{V}_{h+1}^t(s') \right] \right]$$

where $\tilde{r}_h^t(s,a) = r_h(s,a) + b_h^t(s,a)$ is the reward function of our modelled MDP $\widetilde{\mathcal{M}}^t$. On the other hand, we know that $V^\star$ must satisfy

$$V_h^\star(s) = \max_{a \in \mathcal{A}} \left[\tilde{r}_h^t(s,a) + \mathop{\mathbb{E}}_{s' \sim P_h^?(\cdot|s,a)} [V_{h+1}^\star(s')] \right]$$

so it suffices to bound the difference between the two inner expectations. There are two sources of error:

1. The value functions $\widehat{V}_{h+1}^t$ v.s. $V_{h+1}^\star$
2. The transition probabilities $\widehat{P}_h^t$ v.s. $P_h^?$.

We can bound these individually, and then combine them by the triangle inequality. For the former, we can simply bound the difference by H, assuming that the rewards are within $[0,1]$. Now, all that is left is to bound the error from the transition probabilities:

$$\text{error} = \left| \mathop{\mathbb{E}}_{s' \sim \widehat{P}_h^t(\cdot|s,a)} [V_{h+1}^\star(s')] - \mathop{\mathbb{E}}_{s' \sim P_h^?(\cdot|s,a)} [V_{h+1}^\star(s')] \right|. \tag{B.11}$$

Let us bound this term for a fixed s, a, h, t. (Later we can make this uniform across s, a, h, t using the union bound.) Note that expanding out the definition of $\widehat{P}_h^t$ gives

$$\begin{aligned}
\mathop{\mathbb{E}}_{s' \sim \widehat{P}_h^t(\cdot|s,a)} [V_{h+1}^\star(s')] &= \sum_{s' \in \mathcal{S}} \frac{N_h^t(s,a,s')}{N_h^t(s,a)} V_{h+1}^\star(s') \\
&= \frac{1}{N_h^t(s,a)} \sum_{i=0}^{t-1} \sum_{s' \in \mathcal{S}} \mathbf{1}\left\{(s_h^i, a_h^i, s_{h+1}^i) = (s,a,s')\right\} V_{h+1}^\star(s') \\
&= \frac{1}{N_h^t(s,a)} \sum_{i=0}^{t-1} \underbrace{\mathbf{1}\left\{(s_h^i, a_h^i) = (s,a)\right\} V_{h+1}^\star(s_{h+1}^i)}_{X^i}
\end{aligned}$$

since the terms where $s' \neq s_{h+1}^i$ vanish.

Now, in order to apply Hoeffding's inequality, we would like to express the second term in eq. B.11 as a sum over t random variables as well. We will do this by redundantly averaging over all desired trajectories (i.e. where we visit state s and action a at time h):

$$\begin{aligned}
\mathop{\mathbb{E}}_{s' \sim P_h^?(\cdot|s,a)} [V_{h+1}^\star(s')] &= \sum_{s' \in \mathcal{S}} P_h^?(s' \mid s,a) V_{h+1}^\star(s') \\
&= \sum_{s' \in \mathcal{S}} \frac{1}{N_h^t(s,a)} \sum_{i=0}^{t-1} \mathbf{1}\left\{(s_h^i, a_h^i) = (s,a)\right\} P_h^?(s' \mid s,a) V_{h+1}^\star(s') \\
&= \frac{1}{N_h^t(s,a)} \sum_{i=0}^{t-1} \mathop{\mathbb{E}}_{s_{h+1}^i \sim P_h^?(\cdot|s_h^i, a_h^i)} X^i.
\end{aligned}$$

Now we can apply Hoeffding's inequality to $X^i - \mathbb{E}_{s^i_{h+1} \sim P^?_h(\cdot|s^i_h, a^i_h)} X^i$, which is bounded by H, to obtain that, with probability at least $1 - \delta$,

$$\text{error} = \left| \frac{1}{N_h^t(s,a)} \sum_{i=0}^{t-1} \left(X^i - \mathop{\mathbb{E}}_{s^i_{h+1} \sim P^?_h(\cdot|s^i_h, a^i_h)} X^i \right) \right| \leq 2H \sqrt{\frac{\ln(1/\delta)}{N_h^t(s,a)}}.$$

Applying a union bound over all $s \in \mathcal{S}, a \in \mathcal{A}, t \in [T], h \in [H]$ gives the $b_h^t(s,a)$ term above.