# CS 1840: Introduction to Reinforcement Learning

Alexander Cai

2025-01-03

# Table of contents

# Introduction

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 a friendly 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 or in the Chapter 10:

- **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 comfortable with programming in Python. See **?@sec-programming** for more about this textbook's philosophy regarding programming.

## Reinforcement learning in a nutshell

Broadly speaking, RL studies **sequential decision-making** in **dynamic 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 behavior. Many of the most stunning results in machine learning, ranging from AlphaGo to ChatGPT, 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.

  Conversely, 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 and other purposes.

## Core tasks of reinforcement learning

What tasks, exactly, does RL comprise? An RL algorithm must typically solve two main subtasks:

- **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?
- **Policy optimization (control):** Suppose we fully understand how the environment behaves. What is the best action to take in every scenario?

## Course overview

The course will progress through the following units:

Chapter 1 introduces **Markov Decision Processes,** the core mathematical framework for describing a large class of interactive environments.

Chapter 2 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 3 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 4 is a standalone crash course on some tools from supervised learning that we will use in later chapters.

Chapter 5 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 6 explores an important class of algorithms based on iteratively improving a policy. We will also encounter the use of *deep neural networks* to express more complicated policies and approximate complicated functions.

Chapter 7 attempts to learn a good policy from expert demonstrations. At its most basic, this is an application of supervised learning to RL tasks.

Chapter 8 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 9 continues to investigate the exploration-exploitation tradeoff. We will extend ideas from multi-armed bandits to the MDP setting.

Chapter 10 contains an overview of selected background mathematical content and programming content.

## Notation

We will use the following notation throughout the book. This notation is inspired by Sutton and Barto (2018) and (Agarwal et al. 2022). We use $[N]$ as shorthand for the set $\{0, 1, \dots, N-1\}$.

| Element | Space | Definition (of element) |
|---|---|---|
| $s$ | $\mathcal{S}$ | A state. |
| $a$ | $\mathcal{A}$ | An action. |
| $r$ | | A reward. |
| $\gamma$ | | A discount factor. |
| $\tau$ | $\mathcal{T}$ | A trajectory. |
| $\pi$ | $\Pi$ | A policy. |
| $V^\pi$ | $\mathcal{S} \to \mathbb{R}$ | The value function of policy $\pi$. |
| $Q^\pi$ | $\mathcal{S} \times \mathcal{A} \to \mathbb{R}$ | The action-value function (a.k.a. Q-function) of policy $\pi$. |
| $A^\pi$ | $\mathcal{S} \times \mathcal{A} \to \mathbb{R}$ | The advantage function of policy $\pi$. |
| | $\triangle(\mathcal{X})$ | A distribution supported on $\mathcal{X}$. |
| $h$ | $[H]$ | Time horizon index of an MDP (subscript). |
| $k$ | $[K]$ | Arm index of a multi-armed bandit (superscript). |
| $t$ | $[T]$ | Iteration index of an algorithm (subscript). |
| $\theta$ | $\Theta$ | A set of parameters. |

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

## 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, Wisdom, and Farr (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. 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 Markov Decision Processes

---

---

## 1.1 Introduction

The field of RL studies how an agent can learn to make sequential decisions in an interactive environment. This is a very general problem! How can we *formalize* this task in a way that is both *sufficiently general* yet also tractable enough for *fruitful analysis*?

Let's consider some examples of sequential decision problems to identify the key common properties we'd like to capture:

- **Board games and video games,** where a player takes actions in a virtual environment.
- **Inventory management,** where a company must efficiently move resources from producers to consumers.
- **Robotic control**, where a robot can move and interact with the real world to complete some task.

In these environments and many others, the **state transitions**, the "rules" of the environment, only depend on the *most recent* state and action (generally speaking). For example, if you want to take a break while playing a game of chess, you could take a picture of the board, and later on reset the board to that state and continue playing; the past history of moves doesn't matter (generally speaking). This is called the **Markov property.**

**Definition 1.1** (Markov property)**.** An interactive environment satisfies the **Markov property** if the probability of transitioning to a new state only depends on the current state and action:

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

where $P : \mathcal{S} \times \mathcal{A} \to \triangle(\mathcal{S})$ describes the state transitions. (We'll elaborate on this notation later in the chapter.)

Environments that satisfy the Markov property are called **Markov decision processes** (MDPs). This chapter will focus on introducing core vocabulary for MDPs that will be useful throughout the book.

What information might be encoded in the *state* for each of the above examples? What might the valid set of *actions* be? Describe the *state transitions* heuristically and verify that they satisfy the Markov property.

MDPs are usually classified as **finite-horizon**, where the interactions end after some finite number of time steps, or **infinite-horizon**, where the interactions can continue indefinitely. We'll begin with the finite-horizon case and discuss the infinite-horizon case in the second half of the chapter.

We'll describe how to *evaluate* different strategies, called **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.

```
from utils import NamedTuple, Float, Array, partial, jax, jnp, latexify, latex
```

## 1.2 Finite-horizon MDPs

### 1.2.1 Definition

**Definition 1.2** (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 $\mathcal{S}$ to denote the set of possible states, called the **state space**.
2. The **actions** that the agent can take. We use $\mathcal{A}$ to denote the set of possible actions, called the **action space**.
3. Some **initial state distribution** $\mu \in \triangle(\mathcal{S})$.
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.
5. The **reward** signal. 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 in general many results will extend to a *stochastic* reward signal.
6. A time horizon $H \in \mathbb{N}$ that specifies the number of interactions in an **episode**.

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

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

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):

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

Verify that the types and shapes provided above make sense!

```
class MDP(NamedTuple):
    """A description of a Markov decision process with finitely many states and actions."""
    S: int  # number of states
    A: int  # number of actions
    mu: Float[Array, " S"]
    P: Float[Array, "S A S"]  # "current" state, "current" action, "next" state
    r: Float[Array, "S A"]
    H: int
     : float = 1.0  # discount factor (used later)
```

**Example 1.1** (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.

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 (see table below).

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.

These are summarized in the following table:

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

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

```
tidy_mdp = MDP(
    S=2,  # 0 = orderly, 1 = messy
    A=2,  # 0 = ignore, 1 = tidy
    mu=jnp.array([1.0, 0.0]),  # start in orderly state
    P=jnp.array([
        [
            [0.7, 0.3],  # orderly, ignore
            [1.0, 0.0],  # orderly, tidy
        ],
        [
            [0.0, 1.0],  # messy, ignore
            [1.0, 0.0],  # messy, tidy
        ],
    ]),
    r=jnp.array([
        [
            1.0,   # orderly, ignore
            -1.0,  # orderly, tidy
        ],
        [
            -1.0,  # messy, ignore
            0.0,   # messy, tidy
        ]
    ]),
    H=7,
)
```

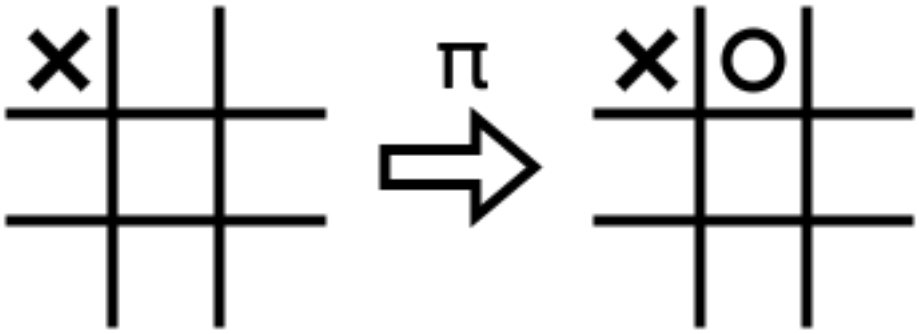
### 1.2.2 Policies

**Definition 1.3** (Policies)**.** A **policy** $\pi$ 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.

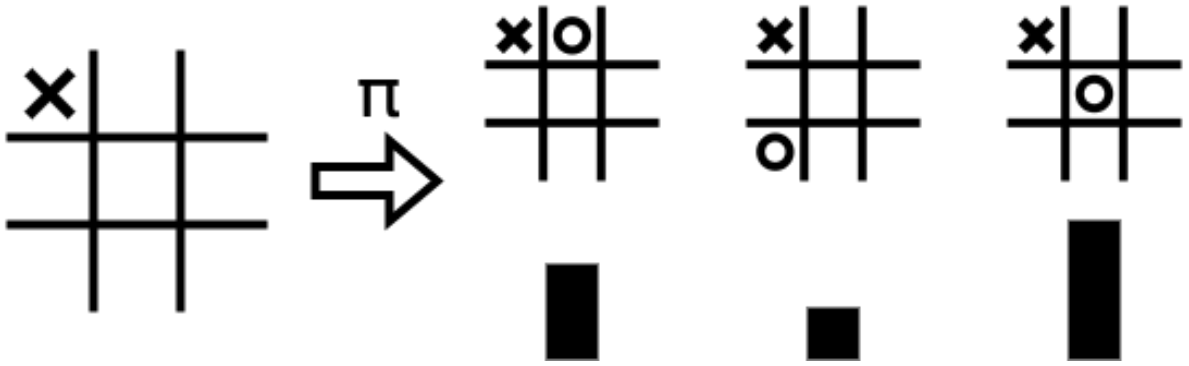
There are three axes along which policies can vary: their outputs, inputs, and time-dependence.

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

2. **State-dependent or history-dependent.** A state-dependent (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 state-dependent policies in this course.




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




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

3. **Stationary or time-dependent.** A stationary (a.k.a. time-homogeneous) policy remains the same function 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. $\pi = \{\pi_0, \ldots, \pi_{H-1}\}$.

Note that for finite state and action spaces, we can represent a randomized mapping $\mathcal{S} \to \Delta(\mathcal{A})$ 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.

A fascinating result is that every finite-horizon MDP has an optimal deterministic time-dependent policy! Intuitively, the Markov property implies that the current state contains all the information we need to make the optimal decision. We'll prove this result constructively later in the chapter.

**Example 1.2** (Policies for the tidying MDP)**.** Here are some possible policies for the tidying MDP Example 1.1:

- 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 $h$.

```
# arrays of shape (H, S, A) represent time-dependent policies
tidy_policy_always_tidy = (
    jnp.zeros((7, 2, 2))
    .at[:, :, 1].set(1.0)
)
tidy_policy_weekends = (
    jnp.zeros((7, 2, 2))
    .at[5:7, :, 1].set(1.0)
    .at[0:5, :, 0].set(1.0)
)
tidy_policy_messy_only = (
    jnp.zeros((7, 2, 2))
    .at[:, 1, 1].set(1.0)
    .at[:, 0, 0].set(1.0)
)
```

*Remark* 1.1. Array objects in Jax are **immutable,** that is, they cannot be *changed.* This might seem inconvenient, but in larger projects, immutability makes code easier to reason about.

### 1.2.3 Trajectories

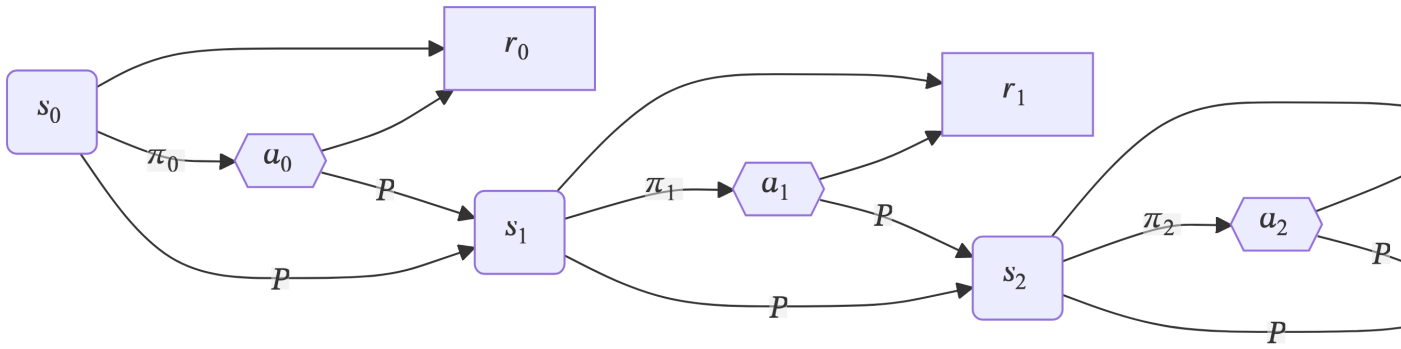
**Definition 1.4** (Trajectories)**.** A sequence of states, actions, and rewards is called a **trajectory**:

$$\tau = (s_0, a_0, r_0, \dots, s_{H-1}, a_{H-1}, r_{H-1})$$

where $r_h = r(s_h, a_h)$. (Note that some sources omit the reward at the final time step. This is a minor detail.)

```
class Transition(NamedTuple):
    """A single state-action-reward interaction with the environment.

    A trajectory comprises a sequence of transitions.
    """
    s: int
    a: int
    r: float
```

Once we've chosen a policy, we can sample trajectories by repeatedly choosing actions according to the policy, transitioning according to the state transitions, and observing the rewards.




That is, a policy induces a distribution $\rho^\pi$ over trajectories. (We assume that $\mu$ and $P$ are clear from context.)

**Example 1.3** (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 |

| $h$ | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|---|
| $r$ | $-1$ | 1 | 1 | $-1$ | 0 | 1 | 1 |

Could any of the policies in Example 1.2 have generated this trajectory?

Note that for a state-dependent policy, using the Markov property Definition 1.1, we can write down the likelihood function of this probability distribution in an **autoregressive** way (i.e. one timestep at a time):

**Definition 1.5** (Autoregressive trajectory distribution)**.**

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

```python
def trajectory_log_likelihood(
    mdp: MDP,
    trj: list[Transition],
    pi: Float[Array, "S A"],
) -> float:
    """Compute the log-likelihood of a trajectory under a given MDP and policy."""

    # initial distribution and action
    total = jnp.log(mdp.mu[trj[0].s])
    total += jnp.log(pi[trj[0].s, trj[0].a])

    # remaining state transitions and actions
    for i in range(1, mdp.H):
        total += jnp.log(mdp.P[trj[i - 1].s, trj[i - 1].a, trj[i].s])
        total += jnp.log(pi[trj[i].s, trj[i].a])

    return total
```

How would you modify this to include stochastic rewards?

For a deterministic policy $\pi$, we have that $\pi_h(a \mid s) = \mathbb{I}[a = \pi_h(s)]$; 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 $\mu$ and the state transitions $P$.

### 1.2.4 Value functions

The main goal of RL is to find a policy that maximizes the expected total reward $\mathbb{E}[r_0 + \cdots + r_{H-1}]$.

Note that $r_0 + \cdots + r_{H-1}$ is a random variable. What sources of randomness does it depend on? Describe the generating process.

Let's introduce some notation for analyzing this quantity.

A policy's **value function** at time $h$ is its expected remaining reward *from a given state*:

**Definition 1.6** (Value function)**.**

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

Similarly, we can define the **action-value function** (aka the **Q-function**) at time $h$ as the expected remaining reward *from a given state and taking a given action*:

**Definition 1.7** (Action-value function)**.**

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

#### 1.2.4.1 Relating the value function and action-value function

Note that the value function is just the expected action-value over actions drawn from the policy:

$$V_h^\pi(s) = \mathbb{E}_{a \sim \pi_h(s)}[Q_h^\pi(s, a)]$$

```
def q_to_v(
    policy: Float[Array, "S A"],
    q: Float[Array, "S A"],
) -> Float[Array, " S"]:
    """
    Compute the value function for a given policy in a known finite MDP
    at a single timestep from its action-value function.
    """
    return jnp.average(q, weights=policy, axis=1)
```

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

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

```
def v_to_q(
    mdp: MDP,
    v_next: Float[Array, " S"],
) -> Float[Array, "S A"]:
    """
    Compute the action-value function in a known finite MDP
    at a single timestep from the corresponding value function.
    """
    # the discount factor is relevant later
    return mdp.r + mdp.  * mdp.P @ v_next


# convert a list of v functions to a list of q functions
v_ary_to_q_ary = jax.vmap(v_to_q, in_axes=(None, 0))
```

#### 1.2.4.2 Greedy policies

For any given $Q \in \mathbb{R}^{|\mathcal{S}| \times |\mathcal{A}|}$, we can define the **greedy policy** $\hat{\pi}_Q$ as the deterministic policy that selects the action with the highest $Q$-value at each state:

$$\hat{\pi}_Q(s) = \arg\max_a Q_{sa}$$

```
def q_to_greedy(q: Float[Array, "S A"]) -> Float[Array, "S A"]:
    """
    Get the (deterministic) greedy policy with respect to an action-value function.
    Return the policy as a matrix of shape (S, A) where each row is a one-hot vector.
    """
    A = q.shape[1]
    a_ary = jnp.argmax(q, axis=1)
    return jnp.eye(A)[a_ary]


def v_to_greedy(mdp: MDP, v: Float[Array, " S"]) -> Float[Array, "S A"]:
    """Get the (deterministic) greedy policy with respect to a value function."""
    return q_to_greedy(v_to_q(mdp, v))
```

### 1.2.5 The one-step (Bellman) consistency equation

Note that by simply considering the cumulative reward as the sum of the *current* reward and the *future* cumulative reward, we can describe the value function recursively (in terms of itself). This is named the **Bellman consistency equation** after **Richard Bellman** (1920–1984), who is credited with introducing dynamic programming in 1953.

**Theorem 1.1** (Bellman consistency equation for the value function)**.**

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

```
def check_bellman_consistency_v(
    mdp: MDP,
    policy: Float[Array, "H S A"],
    v_ary: Float[Array, "H S"],
) -> bool:
    """
    Check that the given (time-dependent) "value function"
    satisfies the Bellman consistency equation.
    """
    return all(
        jnp.allclose(
            # lhs
            v_ary[h],
            # rhs
            jnp.sum(policy[h] * (mdp.r + mdp.  * mdp.P @ v_ary[h + 1]), axis=1),
        )
        for h in range(mdp.H - 1)
    )
```

Verify that this equation holds by expanding $V_h^\pi(s)$ and $V_{h+1}^\pi(s')$.

One can analogously derive the Bellman consistency equation for the action-value function:

**Theorem 1.2** (Bellman consistency equation for action-values)**.**

$$Q_h^\pi(s,a) = r(s,a) + \mathbb{E}_{\substack{s'\sim P(s,a) \\ a'\sim\pi_{h+1}(s')}} [Q_{h+1}^\pi(s',a')]$$

Write a `check_bellman_consistency_q` function for the action-value function.

*Remark* 1.2 (The Bellman consistency equation for deterministic policies). Note that for deterministic policies, the Bellman consistency equation simplifies to

$$V_h^\pi(s) = r(s, \pi_h(s)) + \mathbb{E}_{s' \sim P(s, \pi_h(s))}[V_{h+1}^\pi(s')]$$
$$Q_h^\pi(s, a) = r(s, a) + \mathbb{E}_{s' \sim P(s,a)}[Q_{h+1}^\pi(s', \pi_{h+1}(s'))]$$

### 1.2.6 The one-step Bellman operator

Fix a policy $\pi$. Consider the higher-order operator that takes in a "value function" $v : \mathcal{S} \to \mathbb{R}$ and returns the r.h.s. of the Bellman equation for that "value function":

**Definition 1.8** (Bellman operator)**.**

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

This 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, according to the given policy?

```
def bellman_operator_looping(
    mdp: MDP,
    policy: Float[Array, "S A"],
    v: Float[Array, " S"],
) -> Float[Array, " S"]:
    """
    Looping definition of the Bellman operator.
    Concise version is below
    """
    v_new = jnp.zeros(mdp.S)
    for s in range(mdp.S):
        for a in range(mdp.A):
            for s_next in range(mdp.S):
                v_new[s] += (
                    policy[s, a]
                    * mdp.P[s, a, s_next]
                    * (mdp.r[s, a] + mdp.  * v[s_next])
                )
    return v_new
```

Note that we can concisely implement this using the `q_to_v` and `v_to_q` utilities from above:

```
def bellman_operator(
    mdp: MDP,
    policy: Float[Array, "S A"],
    v: Float[Array, " S"],
) -> Float[Array, " S"]:
    """For a known finite MDP, the Bellman operator can be exactly evaluated."""
    return q_to_v(policy, v_to_q(mdp, v))  # equivalent
    return jnp.sum(policy * (mdp.r + mdp.  * mdp.P @ v), axis=1)
```

We'll call $\mathcal{J}^\pi : \mathbb{R}^{\mathcal{S}} \to \mathbb{R}^{\mathcal{S}}$ the **Bellman operator** of $\pi$. Note that it's defined on any "value function" mapping states to real numbers; $v$ doesn't have to be a well-defined value function for some policy (hence the lowercase notation). The Bellman operator also gives us a concise way to express Theorem 1.1 for the value function:

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

Intuitively, the output of the Bellman operator, a new "value function", evaluates states as follows: from a given state, take one action according to $\pi$, observe the reward, and then evaluate the next state using the input "value function".

When we discuss infinite-horizon MDPs, the Bellman operator will turn out to be more than just a notational convenience: We'll use it to construct algorithms for computing the optimal policy.

## 1.3 Solving finite-horizon MDPs

### 1.3.1 Policy evaluation in finite-horizon MDPs

How can we actually compute the value function of a given policy? This is the task of **policy evaluation**.

#### 1.3.1.1 DP algorithm to evaluate a policy in a finite-horizon MDP

The Bellman consistency equation Theorem 1.1 gives 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 consistency equation to compute the value function at each time step.

```python
def dp_eval_finite(mdp: MDP, policy: Float[Array, "S A"]) -> Float[Array, "H S"]:
    """Evaluate a policy using dynamic programming."""
    V_ary = [None] * mdp.H + [jnp.zeros(mdp.S)]  # initialize to 0 at end of time horizon
    for h in range(mdp.H - 1, -1, -1):
        V_ary[h] = bellman_operator(mdp, policy[h], V_ary[h + 1])
    return jnp.stack(V_ary[:-1])

latex(dp_eval_finite)
```

LatexifyNotSupportedError: Unsupported AST: Slice

This runs in time $O(H \cdot |\mathcal{S}|^2 \cdot |\mathcal{A}|)$ by counting the loops.

Do you see where we compute $Q_h^\pi$ along the way? Make this step explicit.

**Example 1.4** (Tidying policy evaluation)**.** Let's evaluate the policy from Example 1.2 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.

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

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

```
V_messy = dp_eval_finite(tidy_mdp, tidy_policy_messy_only)
V_messy
```

```
Array([[5.5621696, 4.7927704],
       [4.7927704, 4.0241003],
       [4.0241003, 3.253    ],
       [3.253    , 2.49     ],
       [2.49     , 1.7      ],
       [1.7      , 1.       ],
       [1.       , 0.       ]], dtype=float32)
```

### 1.3.2 Optimal policies in finite-horizon MDPs

We've just seen how to *evaluate* a given policy. But how can we find the **optimal policy** for a given environment?

**Definition 1.9** (Optimal policies)**.** We call a policy optimal, and denote it by $\pi^\star$, if it does at least as well as *any* other policy $\pi$ (including stochastic and history-dependent ones) in all situations:

$$\begin{aligned} V_h^{\pi^\star}(s) &= \mathbb{E}_{\tau\sim\rho^{\pi^\star}}[r_h + \cdots + r_{H-1} \mid s_h = s] \\ &\geq \mathbb{E}_{\tau\sim\rho^{\pi}}[r_h + \cdots + r_{H-1} \mid \tau_h] \quad \forall \pi, \tau_h, h \in [H] \end{aligned}$$

where we condition on the trajectory up to time $h$, denoted $\tau_h = (s_0, a_0, r_0, \ldots, s_h)$, where $s_h = s$.

Convince yourself that all optimal policies must have the same value function. We call this the **optimal value function** and denote it by $V_h^\star(s)$. The same goes for the action-value function $Q_h^\star(s, a)$.

It is a stunning fact that **every finite-horizon MDP has an optimal policy that is time-dependent and deterministic.** In particular, we can construct such a policy by acting *greedily* with respect to the optimal action-value function:

**Theorem 1.3** (It is optimal to be greedy with respect to the optimal value function)**.**

$$\pi_h^\star(s) = \arg\max_a Q_h^\star(s, a).$$

*Proof.* Let $V^\star$ and $Q^\star$ denote the optimal value and action-value functions. Consider the greedy policy

$$\hat{\pi}_h(s) := \arg\max_a Q_h^\star(s,a).$$

We aim to show that $\hat{\pi}$ is optimal; that is, $V^{\hat{\pi}} = V^\star$.

Fix an arbitrary state $s \in \mathcal{S}$ and time $h \in [H]$.

Firstly, by the definition of $V^\star$, we already know $V_h^\star(s) \geq V_h^{\hat{\pi}}(s)$. So for equality to hold we just need to show that $V_h^\star(s) \leq V_h^{\hat{\pi}}(s)$. We'll first show that the Bellman operator $\mathcal{J}^{\hat{\pi}}$ never decreases $V_h^\star$. Then we'll apply this result recursively to show that $V^\star = V^{\hat{\pi}}$.

#### 1.3.2.1 The Bellman operator never decreases the optimal value function

$\mathcal{J}^{\hat{\pi}}$ never decreases $V_h^\star$ (elementwise):

$$[\mathcal{J}^{\hat{\pi}}(V_{h+1}^\star)](s) \geq V_h^\star(s).$$

**Proof:**

$$\begin{aligned}
V_h^\star(s) &= \max_{\pi \in \Pi} V_h^\pi(s) \\
&= \max_{\pi \in \Pi} \mathop{\mathbb{E}}_{a \sim \pi(\dots)} \left[ r(s,a) + \mathop{\mathbb{E}}_{s' \sim P(s,a)} V_{h+1}^\pi(s') \right] && \text{Bellman consistency} \\
&\leq \max_{\pi \in \Pi} \mathop{\mathbb{E}}_{a \sim \pi(\dots)} \left[ r(s,a) + \mathop{\mathbb{E}}_{s' \sim P(s,a)} V_{h+1}^\star(s') \right] && \text{definition of } V^\star \\
&= \max_a \left[ r(s,a) + \mathop{\mathbb{E}}_{s' \sim P(s,a)} V_{h+1}^\star(s') \right] && \text{only depends on } \pi \text{ via } a \\
&= [\mathcal{J}^{\hat{\pi}}(V_{h+1}^\star)](s).
\end{aligned}$$

Note that the chosen action $a \sim \pi(\dots)$ above might depend on the past history; this isn't shown in the notation and doesn't affect our result (make sure you see why).

We can now apply this result recursively to get

$$V_t^\star(s) \leq V_t^{\hat{\pi}}(s)$$

as follows. (Note that even though $\hat{\pi}$ is deterministic, we'll use the $a \sim \hat{\pi}(s)$ notation to make it explicit that we're sampling a trajectory from it.)

$$
\begin{aligned}
V_t^\star(s) &\le [\mathcal{J}^{\hat{\pi}}(V_{h+1}^\star)](s) \\
&= \mathop{\mathbb{E}}_{a\sim\hat{\pi}(s)}\left[r(s,a) + \mathop{\mathbb{E}}_{s'\sim P(s,a)}\left[V_{h+1}^\star(s')\right]\right] && \text{definition of } \mathcal{J}^{\hat{\pi}} \\
&\le \mathop{\mathbb{E}}_{a\sim\hat{\pi}(s)}\left[r(s,a) + \mathop{\mathbb{E}}_{s'\sim P(s,a)}\left[[\mathcal{J}^{\hat{\pi}}(V_{t+2}^\star)](s')\right]\right] && \text{above lemma} \\
&= \mathop{\mathbb{E}}_{a\sim\hat{\pi}(s)}\left[r(s,a) + \mathop{\mathbb{E}}_{s'\sim P(s,a)}\left[\mathop{\mathbb{E}}_{a'\sim\hat{\pi}} r(s',a') + \mathop{\mathbb{E}}_{s''} V_{t+2}^\star(s'')\right]\right] && \text{definition of } \mathcal{J}^{\hat{\pi}} \\
&\le \cdots && \text{apply at all timesteps} \\
&= \mathop{\mathbb{E}}_{\tau\sim\rho^{\hat{\pi}}}[G_t \mid s_h = s] && \text{rewrite expectation} \\
&= V_t^{\hat{\pi}}(s) && \text{definition}
\end{aligned}
$$

And so we have $V^\star = V^{\hat{\pi}}$, making $\hat{\pi}$ optimal. □

Note that this also gives simplified forms of the Theorem 1.1 for the optimal policy:

**Theorem 1.4** (Bellman consistency equations for the optimal policy)**.**

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

Now that we've shown this particular greedy policy is optimal, all we need to do is compute the optimal value function and optimal policy. We can do this by working backwards in time using **dynamic programming** (DP).

**Definition 1.10** (DP algorithm to compute an optimal policy in a finite-horizon MDP)**. Base case.** At the end of the episode (time step $H-1$), we can't take any more actions, so the $Q$-function is simply the reward that we obtain:

$$
Q_{H-1}^\star(s,a) = r(s,a)
$$

so the best thing to do is just act greedily and get as much reward as we can!

$$
\pi_{H-1}^\star(s) = \arg\max_a Q_{H-1}^\star(s,a)
$$

Then $V_{H-1}^\star(s)$, the optimal value of state $s$ at the end of the trajectory, is simply whatever action gives the most reward.

$$V^\star_{H-1} = \max_a Q^\star_{H-1}(s,a)$$

**Recursion.** Then, we can work backwards in time, starting from the end, using our consistency equations! i.e. for each $t = H-2, \dots, 0$, we set

$$\begin{aligned} Q^\star_t(s,a) &= r(s,a) + \mathbb{E}_{s' \sim P(s,a)}[V^\star_{h+1}(s')] \\ \pi^\star_t(s) &= \arg\max_a Q^\star_t(s,a) \\ V^\star_t(s) &= \max_a Q^\star_t(s,a) \end{aligned}$$

```
def find_optimal_policy(mdp: MDP):
    Q = [None] * mdp.H
    pi = [None] * mdp.H
    V = [None] * mdp.H + [jnp.zeros(mdp.S)]  # initialize to 0 at end of time horizon

    for h in range(mdp.H - 1, -1, -1):
        Q[h] = mdp.r + mdp.P @ V[h + 1]
        pi[h] = jnp.eye(mdp.S)[jnp.argmax(Q[h], axis=1)]  # one-hot
        V[h] = jnp.max(Q[h], axis=1)

    Q = jnp.stack(Q)
    pi = jnp.stack(pi)
    V = jnp.stack(V[:-1])

    return pi, V, Q
```

At each of the $H$ timesteps, we must compute $Q^\star$ for each of the $|\mathcal{S}||\mathcal{A}|$ state-action pairs. Each computation takes $|\mathcal{S}|$ operations to evaluate the average value over $s'$. This gives 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 `dp_eval_finite` in Section 1.3.1, but instead of *averaging* over the actions chosen by a policy, we instead simply 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.

```
pi_opt, V_opt, Q_opt = find_optimal_policy(tidy_mdp)
assert jnp.allclose(pi_opt, tidy_policy_messy_only)
assert jnp.allclose(V_opt, V_messy)
assert jnp.allclose(Q_opt[:-1], v_ary_to_q_ary(tidy_mdp, V_messy)[1:])
"Assertions passed (the 'tidy when messy' policy is optimal)"
```

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

## 1.4 Infinite-horizon MDPs

What happens if a trajectory is allowed to continue forever (i.e. $H = \infty$)? This is the setting of **infinite horizon** MDPs.

In this chapter, we'll describe the necessary adjustments from the finite-horizon case to make the problem tractable. We'll show that the Bellman operator in the discounted reward setting is a **contraction mapping** for any policy. We'll discuss how to evaluate policies (i.e. compute their corresponding value functions). Finally, we'll present and analyze two iterative algorithms, based on the Bellman operator, for computing the optimal policy: **value iteration** and **policy iteration**.

### 1.4.1 Discounted rewards

First of all, note that maximizing the cumulative reward $r_h + r_{h+1} + r_{h+2} + \cdots$ is no longer a good idea since it might blow up to infinity. Instead of a time horizon $H$, we now need a **discount factor** $\gamma \in [0, 1)$ such that rewards become less valuable the further into the future they are:

$$r_h + \gamma r_{h+1} + \gamma^2 r_{h+2} + \cdots = \sum_{k=0}^{\infty} \gamma^k r_{h+k}.$$

We can think of $\gamma$ as measuring how much we care about the future: if it's close to 0, we only care about the near-term rewards; it's close to 1, we put more weight into future rewards.

You can also analyze $\gamma$ 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 $\gamma$.) This accords with the above interpretation: if $\gamma$ is close to 0, the trajectory will likely be very short, while if $\gamma$ is close to 1, the trajectory will likely continue for a long time.

Assuming that $r_h \in [0, 1]$ for all $h \in \mathbb{N}$, what is the maximum **discounted** cumulative reward? You may find it useful to review geometric series.

The other components of the MDP remain the same:

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

Code-wise, we can reuse the `MDP` class from before Definition 1.2 and set `mdp.H = float('inf')`.

```
tidy_mdp_inf = tidy_mdp._replace(H=float("inf"),  =0.95)
```

### 1.4.2 Stationary policies

The time-dependent policies from the finite-horizon case become difficult to handle in the infinite-horizon case. In particular, 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 $\Delta(\mathcal{A})$ (stochastic).

Which of the policies in Example 1.2 are stationary?

### 1.4.3 Value functions and Bellman consistency

We also consider stationary value functions $V^\pi : \mathcal{S} \to \mathbb{R}$ and $Q^\pi : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$. We need to insert a factor of $\gamma$ into the Bellman consistency equation Theorem 1.1 to account for the discounting:

$$\begin{aligned} V^\pi(s) &= \mathbb{E}_{\tau\sim\rho^\pi}[r_h + \gamma r_{h+1} + \gamma^2 r_{h+2} \cdots \mid s_h = s] && \text{for any } h \in \mathbb{N} \\ &= \mathbb{E}_{\substack{a\sim\pi(s) \\ s'\sim P(s,a)}}[r(s,a) + \gamma V^\pi(s')] \\ Q^\pi(s,a) &= \mathbb{E}_{\tau\sim\rho^\pi}[r_h + \gamma r_{h+1} + \gamma^2 r_{h+2} + \cdots \mid s_h = s, a_h = a] && \text{for any } h \in \mathbb{N} \\ &= r(s,a) + \gamma \mathbb{E}_{\substack{s'\sim P(s,a) \\ a'\sim\pi(s')}}[Q^\pi(s',a')] \end{aligned} \tag{1.1}$$

Heuristically speaking, why does it no longer matter which time step we condition on when defining the value function?

## 1.5 Solving infinite-horizon MDPs

### 1.5.1 The Bellman operator is a contraction mapping

Recall from Definition 1.8 that the Bellman operator $\mathcal{J}^\pi$ for a policy $\pi$ takes in a "value function" $v : \mathcal{S} \to \mathbb{R}$ and returns the r.h.s. of the Bellman equation for that "value function". In the infinite-horizon setting, this is

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

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

**Definition 1.11** (Contraction mapping)**.** Let $X$ be some space 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 $\gamma \in (0, 1)$. Intuitively, this means that if two points are $\delta$ far apart, after applying the mapping,

Show that for a contraction mapping $f$ with coefficient $\gamma$, for all $t \in \mathbb{N}$,

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

i.e. that any two points will be pushed closer by at least a factor of $\gamma$ 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 $x^\star$:

$$\|f^{(t)}(x) - x^\star\| \leq \gamma^t \|x - x^\star\|. \tag{1.2}$$

Let's return to the RL setting and apply this result to the Bellman operator. How can we measure the distance between two "value 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 "value functions" on the state that causes the biggest gap between them. Then Equation 1.2 implies that if we repeatedly apply $\mathcal{J}^\pi$ to any starting "value function", we will eventually converge to $V^\pi$:

$$\|(\mathcal{J}^\pi)^{(t)}(v) - V^\pi\|_\infty \leq \gamma^t \|v - V^\pi\|_\infty. \tag{1.3}$$

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

**Theorem 1.5** (The Bellman operator is a contraction mapping)**.**

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

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

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

□

### 1.5.2 Policy evaluation in infinite-horizon MDPs

The backwards DP technique we used in the finite-horizon case (Section 1.3.1) no longer works since there is no "final timestep" to start from. We'll need another approach to policy evaluation.

The Bellman consistency conditions yield a system of 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.

#### 1.5.2.1 Matrix inversion for deterministic policies

Note that when the policy $\pi$ 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:

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

For $P^\pi$, 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 $\pi$.

**Example 1.5** (Tidying MDP)**.** The tabular MDP from before has $|\mathcal{S}| = 2$ and $|\mathcal{A}| = 2$. Let's write down the quantities for the policy $\pi$ that tidies if and only if the room is messy:

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

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 $Q^\pi$.) This system of equations can be solved with a matrix inversion:

$$V^\pi = (I - \gamma P^\pi)^{-1} r^\pi. \tag{1.4}$$

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. In this case, we can see that $I - \gamma P^\pi$ is invertible because it maps any nonzero vector to a vector with at least one nonzero element.)

```
def eval_deterministic_infinite(
    mdp: MDP, policy: Float[Array, "S A"]
) -> Float[Array, " S"]:
    pi = jnp.argmax(policy, axis=1)  # un-one-hot
    P_pi = mdp.P[jnp.arange(mdp.S), pi]
    r_pi = mdp.r[jnp.arange(mdp.S), pi]
    return jnp.linalg.solve(jnp.eye(mdp.S) - mdp.  * P_pi, r_pi)
```

**Example 1.6** (Tidying policy evaluation)**.** Let's use the same policy $\pi$ 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.

```
eval_deterministic_infinite(tidy_mdp_inf, tidy_policy_messy_only[0])
```

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

#### 1.5.2.2 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.

```
def supremum_norm(v):
    return jnp.max(jnp.abs(v))  # same as jnp.linalg.norm(v, jnp.inf)


def loop_until_convergence(op, v,  =1e-6):
    """Repeatedly apply op to v until convergence (in supremum norm)."""
    while True:
        v_new = op(v)
        if supremum_norm(v_new - v) <  :
            return v_new
        v = v_new
```

```
def iterative_evaluation(mdp: MDP, pi: Float[Array, "S A"],  =1e-6) -> Float[Array, " S"]:
    op = partial(bellman_operator, mdp, pi)
    return loop_until_convergence(op, jnp.zeros(mdp.S),  )
```

Then, as we showed in Equation 1.3, by the Banach fixed-point theorem:

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

```
iterative_evaluation(tidy_mdp_inf, tidy_policy_messy_only[0])
```

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

*Remark* 1.3 (Convergence of iterative policy evaluation). How many iterations do we need for an $\epsilon$-accurate estimate? We can work backwards to solve for $t$:

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

and so the number of iterations required for an $\epsilon$-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 $\log(1/x) \geq 1 - x$.

### 1.5.3 Optimal policies in infinite-horizon MDPs

Now let's move on to solving for an optimal policy in the infinite-horizon case. As in Definition 1.9, 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 $\pi$, states $s \in \mathcal{S}$, times $h \in \mathbb{N}$, and initial trajectories $\tau_h = (s_0, a_0, r_0, \ldots, s_h)$ where $s_h = s$,

$$\begin{aligned} V^{\pi^\star}(s) &= \mathbb{E}_{\tau \sim \rho^{\pi^\star}}[r_h + \gamma r_{h+1} + \gamma^2 r_{h+2} + \cdots \mid s_h = s] \\ &\geq \mathbb{E}_{\tau \sim \rho^{\pi}}[r_h + \gamma r_{h+1} + \gamma^2 r_{h+2} + \cdots \mid \tau_h] \end{aligned} \tag{1.5}$$

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.

Verify this by modifying the proof Theorem 1.3 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 Definition 1.10 since there's no "final timestep" to start from. Instead, we'll exploit the fact that the Bellman consistency equation Equation 1.1 for the optimal value function doesn't depend on any policy:

$$V^\star(s) = \max_a \left[ r(s,a) + \gamma \mathbb{E}_{s' \sim P(s,a)} V^\star(s'). \right] \tag{1.6}$$

Verify this by substituting the greedy policy into the Bellman consistency equation.

As before, thinking of the r.h.s. of Equation 1.6 as an operator on value functions gives the **Bellman optimality operator**

$$[\mathcal{J}^\star(v)](s) = \max_a \left[ r(s,a) + \gamma \mathbb{E}_{s' \sim P(s,a)} v(s') \right] \tag{1.7}$$

```
def bellman_optimality_operator(mdp: MDP, v: Float[Array, " S"]) -> Float[Array, " S"]:
    return jnp.max(mdp.r + mdp.  * mdp.P @ v, axis=1)


def check_optimal(v: Float[Array, " S"], mdp: MDP):
    return jnp.allclose(v, bellman_optimality_operator(v, mdp))
```

#### 1.5.3.1 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**.

```
def value_iteration(mdp: MDP,  : float = 1e-6) -> Float[Array, " S"]:
    """Iterate the Bellman optimality operator until convergence."""
    op = partial(bellman_optimality_operator, mdp)
    return loop_until_convergence(op, jnp.zeros(mdp.S),  )
```

```
value_iteration(tidy_mdp_inf)
```

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

Note that the runtime analysis for an $\epsilon$-optimal value function is exactly the same as Section 1.5.2.2! 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:

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

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 1.6** (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*

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

*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's 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

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

The second quantity is bounded by

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

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 1.3, 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 $\epsilon$-accurate estimate of the optimal value function.

#### 1.5.3.2 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.

```
def policy_iteration(mdp: MDP,  =1e-6) -> Float[Array, "S A"]:
    """Iteratively improve the policy and value function."""
    def op(pi):
        return v_to_greedy(mdp, eval_deterministic_infinite(mdp, pi))
    pi_init = jnp.ones((mdp.S, mdp.A)) / mdp.A  # uniform random policy
    return loop_until_convergence(op, pi_init,  )
```

```
policy_iteration(tidy_mdp_inf)
```

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

Although PI appears more complex than VI, we'll use Theorem 1.5 to show convergence. This will give us the same runtime bound as value iteration and iterative policy evaluation for an $\epsilon$-optimal value function Remark 1.3, although in practice, PI often converges much faster.

**Theorem 1.7** (Policy Iteration runtime and convergence)**.** *We aim to show that the number of iterations required for an $\epsilon$-accurate estimate of the optimal value function is*

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

*This bound follows from the contraction property Equation 1.3:*

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

*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*

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

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

$$\begin{aligned}V^{\pi^{t+1}}(s) - V^{\pi^t}(s) &\geq V^{\pi^{t+1}}(s) - \mathcal{J}^\star(V^{\pi^t})(s) \\ &= \gamma\mathbb{E}_{s'\sim P(s,\pi^{t+1}(s))}\left[V^{\pi^{t+1}}(s') - V^{\pi^t}(s')\right].\end{aligned} \tag{1.8}$$

*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*

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

*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 Equation 1.8 to get the desired result:*

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

*This means we can now apply the Bellman convergence result Equation 1.3 to get*

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

## 1.6 Summary

- 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 $\mu \in \Delta(\mathcal{S})$, 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 $\pi$ that maximizes expected total reward. Policies can be **deterministic** or **stochastic**, **state-dependent** or **history-dependent**, **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 $\pi$. 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 $\pi$. 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 Linear Quadratic Regulators

---

math: ‘s’: ’x’ ‘a’: ‘u’

---

## 2.1 Introduction

Up to this point, we have 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, 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**. We will then extend this basic method to more complex settings.

**Example 2.1** (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*:

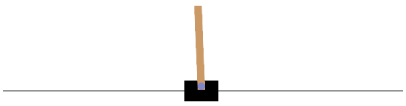


Figure 2.3: Cart pole

The state $s \in \mathbb{R}^4$ can be described by:

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 $a \in \mathbb{R}$.

**Goal:** Stabilize the cart around an ideal state and action $(s^\star, a^\star)$.

## 2.2 Optimal control

Recall that an MDP is defined by its state space $\mathcal{S}$, action space $\mathcal{A}$, state transitions $P$, reward function $r$, and discount factor $\gamma$ or time horizon $H$. These have equivalents in the control setting:

- The state and action spaces are *continuous* rather than finite. That is, $\mathcal{S} \subseteq \mathbb{R}^{n_s}$ and $\mathcal{A} \subseteq \mathbb{R}^{n_a}$, where $n_s$ and $n_a$ are the corresponding dimensions of these spaces, i.e. the number of coordinates to specify a single state or action respectively.

- We call the state transitions the **dynamics** of the system. In the most general case, these might change across timesteps and also include some stochastic **noise** $w_h$ at each timestep. We denote these dynamics as the function $f_h$ such that $s_{h+1} = f_h(s_h, a_h, w_h)$. Of course, we can simplify to cases where the dynamics are *deterministic/noise-free* (no $w_h$ term) and/or *time-homogeneous* (the same function $f$ across timesteps).
- 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** $(s^\star, a^\star)$. An important special case is when the cost is *time-homogeneous*; that is, it remains the same function $c$ at each timestep $h$.
- We seek to minimize the *undiscounted* cost within a *finite time horizon* $H$. Note that we end an episode at the final state $s_H$ – there is no $a_H$, and so we denote the cost for the final state as $c_H(s_H)$.

With all of these components, we can now formulate the **optimal control problem:** *compute a policy to minimize the expected undiscounted cost over $H$ timesteps.* In this chapter, we will only consider *deterministic, time-dependent* policies $\pi = (\pi_0, \dots, \pi_{H-1})$ where $\pi_h : \mathcal{S} \to \mathcal{A}$ for each $h \in [H]$.

**Definition 2.1** (General optimal control problem)**.**

$$\begin{aligned} \min_{\pi_0,\dots,\pi_{H-1}:\mathcal{S}\to\mathcal{A}} \quad & \mathbb{E}\left[\left(\sum_{h=0}^{H-1} c_h(s_h, a_h)\right) + c_H(s_H)\right] \\ \text{where} \quad & s_{h+1} = f_h(s_h, a_h, w_h), \\ & a_h = \pi_h(s_h) \\ & s_0 \sim \mu_0 \\ & w_h \sim \text{noise} \end{aligned}$$

### 2.2.1 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'd be able to work backwards using the DP algorithm for computing the optimal policy in an MDP (Definition 1.10). This inspires us to try *discretizing* the problem.

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

However, the discretized $\widetilde{\mathcal{S}}$ and $\widetilde{\mathcal{A}}$ may be finite, but they may be infeasibly large: we must divide *each dimension* into intervals of length $\varepsilon$, resulting in $|\widetilde{\mathcal{S}}| = (B_s/\varepsilon)^{n_s}$ and $|\widetilde{\mathcal{A}}| = (B_a/\varepsilon)^{n_a}$. To get a sense of how quickly this grows, consider $\varepsilon = 0.01, n_s = n_a = 10$. Then the number

of elements in the transition matrix would be $|\widetilde{\mathcal{S}}|^2|\widetilde{\mathcal{A}}| = (100^{10})^2(100^{10}) = 10^{60}$! (That's a trillion trillion trillion trillion trillion trillion.)

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 $\varepsilon$ wouldn't change the behavior of the system by much; namely, that the cost and dynamics were relatively *continuous*. Can we use this continuous structure in other ways? This leads us to the **linear quadratic regulator**.

## 2.3 The Linear Quadratic Regulator

The optimal control problem Definition 2.1 seems highly complex in general. Is there a relevant simplification that we can analyze? The **linear quadratic regulator** (LQR) is a solvable case and a fundamental tool in control theory.

**Definition 2.2** (The linear quadratic regulator)**.** The LQR problem is a special case of the Definition 2.1 with *linear dynamics* and an *upward-curved quadratic cost function*. Solving the LQR problem will additionally enable us to *locally approximate* more complex setups using *Taylor approximations*.

**Linear, time-homogeneous dynamics**: for each timestep $h \in [H]$,

$$\begin{aligned} s_{h+1} &= f(s_h, a_h, w_h) = As_h + Ba_h + w_h \\ \text{where } w_h &\sim \mathcal{N}(0, \sigma^2 I). \end{aligned}$$

Here, $w_h$ is a spherical Gaussian **noise term** that makes the dynamics random. Setting $\sigma = 0$ gives us **deterministic** state transitions. We will find that the optimal policy actually *does not depend on the noise*, although the optimal value function and Q-function do.

**Upward-curved quadratic, time-homogeneous cost function**:

$$c(s_h, a_h) = \begin{cases} s_h^\top Q s_h + a_h^\top R a_h & h < H \\ s_h^\top Q s_h & h = H \end{cases}.$$

This cost function attempts to stabilize the state and action about $(s^\star, a^\star) = (0, 0)$. We require $Q \in \mathbb{R}^{n_s \times n_s}$ and $R \in \mathbb{R}^{n_a \times n_a}$ to both be *positive definite* matrices so that $c$ has a well-defined unique minimum. We can furthermore assume without loss of generality that they are both *symmetric* (see exercise below).

This results in the LQR optimization problem:

$$\begin{aligned}\min_{\pi_0,\dots,\pi_{H-1}:\mathcal{S}\to\mathcal{A}} \quad & \mathbb{E}\left[\left(\sum_{h=0}^{H-1} s_h^\top Q s_h + a_h^\top R a_h\right) + s_H^\top Q s_H\right] \\ \text{where} \quad & s_{h+1} = As_h + Ba_h + w_h \\ & a_h = \pi_h(s_h) \\ & w_h \sim \mathcal{N}(0, \sigma^2 I) \\ & s_0 \sim \mu_0.\end{aligned}$$

**Exercise 2.1** (Symmetric $Q$ and $R$)**.** Here we'll show that we don't lose generality by assuming that $Q$ and $R$ are symmetric. Show that replacing $Q$ and $R$ with $(Q+Q^\top)/2$ and $(R+R^\top)/2$ (which are symmetric) yields the same cost function.

We will henceforth abbreviate "symmetric positive definite" as s.p.d. and "positive definite" as p.d.

It will be helpful to reintroduce the *value function* notation for a policy to denote the average cost it incurs. These will be instrumental in constructing the optimal policy via **dynamic programming,** as we did in Section 1.3.2 for MDPs.

**Definition 2.3** (Value functions for 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 average **cost-to-go** incurred by that policy:

$$\begin{aligned}V_h^\pi(s) &= \mathbb{E}\left[\left(\sum_{i=h}^{H-1} c(s_i, a_i)\right) + c(s_H) \mid s_h = s, a_i = \pi_i(s_i) \quad \forall h \le i < H\right] \\ &= \mathbb{E}\left[\left(\sum_{i=h}^{H-1} s_i^\top Q s_i + a_i^\top R a_i\right) + s_H^\top Q s_H \mid s_h = s, a_i = \pi_i(s_i) \quad \forall h \le i < H\right]\end{aligned}$$

The Q-function additionally conditions on the first action we take:

$$\begin{aligned}Q_h^\pi(s,a) = \mathbb{E}\Bigg[ & \left(\sum_{i=h}^{H-1} c(s_i, a_i)\right) + c(s_H) \\ & \mid (s_h, a_h) = (s, a), a_i = \pi_i(s_i) \quad \forall h \le i < H\Bigg] \\ = \mathbb{E}\Bigg[ & \left(\sum_{i=h}^{H-1} s_i^\top Q s_i + a_i^\top R a_i\right) + s_H^\top Q s_H \\ & \mid (s_h, a_h) = (s, a), a_i = \pi_i(s_i) \quad \forall h \le i < H\Bigg]\end{aligned}$$

Note that since we use *cost* instead of *reward,* the best policies are the ones with *smaller* values of the value function.

## 2.4 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 $\pi_h^\star$ in Definition 2.2 using **dynamic programming** in a very similar way to the DP algorithms in Section 1.3.1. Recall the definition of the optimal value function:

**Definition 2.4** (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*:

$$\begin{aligned} V_h^\star(s) &= \min_{\pi_h,\dots,\pi_{H-1}} V_h^\pi(s) \\ &= \min_{\pi_h,\dots,\pi_{H-1}} \mathbb{E}\Bigg[ \left( \sum_{i=h}^{H-1} s_h^\top Q s_h + a_h^\top R a_h \right) + s_H^\top Q s_H \\ &\qquad\qquad \mid s_h = s, a_i = \pi_i(s_i) \quad \forall h \le i < H \Bigg] \end{aligned}$$

The optimal Q-function is defined similarly, conditioned on the starting action as well:

$$\begin{aligned} Q_h^\star(s,a) &= \min_{\pi_h,\dots,\pi_{H-1}} Q_h^\pi(s,a) \\ &= \min_{\pi_h,\dots,\pi_{H-1}} \mathbb{E}\Bigg[ \left( \sum_{i=h}^{H-1} s_h^\top Q s_h + a_h^\top R a_h \right) + s_H^\top Q s_H \\ &\qquad\qquad \mid s_h = s, a_h = a, a_i = \pi_i(s_i) \quad \forall h < i < H \Bigg] \end{aligned}$$

Both of the definitions above assume *deterministic* policies. Otherwise we would have to take an *expectation* over actions drawn from the policy, i.e. $a_h \sim \pi_h(s_h)$.

We will prove the striking fact that 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 noise!

**Theorem 2.1** (Optimal value function in LQR is an upward-curved quadratic)**.** *At each timestep $h \in [H]$,*

$$V_h^\star(s) = s^\top P_h s + p_h$$

*for some s.p.d. matrix* $P_h \in \mathbb{R}^{n_s \times n_s}$ *and scalar* $p_h \in \mathbb{R}$.

**Theorem 2.2** (Optimal policy in LQR is linear)**.** *At each timestep* $h \in [H]$*,*

$$\pi_h^\star(s) = -K_h s$$

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

The construction (and inductive proof) proceeds similarly to the one in the MDP setting (Section 1.3.1).

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 $V_H^\star(s) = c(s) = s^\top Q s$. Thus $V_H^\star(s) = s^\top P_H s + p_H$ where $P_H = Q$ and $p_H = 0$.

**Inductive hypothesis:** We seek to show that the inductive step holds for both theorems: If $V_{h+1}^\star(s)$ is an upward-curved quadratic, then $V_h^\star(s)$ must also be an upward-curved quadratic, and $\pi_h^\star(s)$ must be linear. We'll break this down into the following steps:

1. Show that $Q_h^\star(s, a)$ is an upward-curved quadratic (in both $s$ and $a$).
2. Derive the optimal policy $\pi_h^\star(s) = \arg\min_a Q_h^\star(s, a)$ and show that it's linear.
3. Show that $V_h^\star(s)$ is an upward-curved quadratic.

We first assume the inductive hypothesis that our theorems are true at time $h + 1$. That is,

$$V_{h+1}^\star(s) = s^\top P_{h+1} s + p_{h+1} \quad \forall s \in \mathcal{S}.$$

#### 2.4.0.1 $Q_h^\star(s, a)$ 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:

$$Q_h^\star(s, a) = c(s, a) + \mathbb{E}_{s' \sim f(s,a,w_{h+1})}[V_{h+1}^\star(s')].$$

Recall $c(s, a) := s^\top Q s + a^\top R a$. 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:

$$
\begin{aligned}
&\mathbb{E}_{s'}[V^\star_{h+1}(s')] \\
&= \mathbb{E}_{w_{h+1}}[V^\star_{h+1}(As + Ba + w_{h+1})] && \text{definition of } f \\
&= \mathbb{E}_{w_{h+1}}[(As + Ba + w_{h+1})^\top P_{h+1}(As + Ba + w_{h+1}) + p_{h+1}]. && \text{inductive hypothesis}
\end{aligned}
$$

Summing and combining like terms, we get

$$
\begin{aligned}
Q^\star_h(s,a) &= s^\top Q s + a^\top R a + \mathbb{E}_{w_{h+1}}[(As + Ba + w_{h+1})^\top P_{h+1}(As + Ba + w_{h+1}) + p_{h+1}] \\
&= s^\top (Q + A^\top P_{h+1} A)s + a^\top (R + B^\top P_{h+1} B)a + 2s^\top A^\top P_{h+1} B a \\
&\quad + \mathbb{E}_{w_{h+1}}[w_{h+1}^\top P_{h+1} w_{h+1}] + p_{h+1}.
\end{aligned}
$$

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

$$
\begin{aligned}
\mathbb{E}_{w_{h+1}}[w_{h+1}^\top P_{h+1} w_{h+1}] &= \sum_{i=1}^{d}\sum_{j=1}^{d} (P_{h+1})_{ij} \mathbb{E}_{w_{h+1}}[(w_{h+1})_i (w_{h+1})_j] \\
&= \sigma^2 \mathrm{Tr}(P_{h+1})
\end{aligned}
$$

#### 2.4.0.2 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

$$
(P_{h+1})_{ii} \mathbb{E}(w_{h+1})_i^2 = \sigma^2 (P_{h+1})_{ii}.
$$

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:

$$
(P_{h+1})_{ij} \mathbb{E}[(w_{h+1})_i] \mathbb{E}[(w_{h+1})_j] = 0.
$$

Thus, the only terms left are the ones on the diagonal, so the sum of these can be expressed as the trace of $\sigma^2 P_{h+1}$:

$$
\mathbb{E}_{w_{h+1}}[w_{h+1}^\top P_{h+1} w_{h+1}] = \sigma^2 \mathrm{Tr}(P_{h+1}).
$$

Substituting this back into the expression for $Q_h^\star$, we have:

$$\begin{aligned} Q_h^\star(s,a) = s^\top(Q + A^\top P_{h+1} A)s + a^\top(R + B^\top P_{h+1} B)a + 2s^\top A^\top P_{h+1} Ba \\ + \sigma^2 \mathrm{Tr}(P_{h+1}) + p_{h+1}. \end{aligned}$$

As we hoped, this expression is quadratic in $s$ and $a$. Furthermore, we'd like to show that it also *curves upwards* with respect to $a$ so that its minimum with respect to $a$ is well-defined. We can do this by noting that the **Hessian matrix** of second derivatives is positive definite:

$$\nabla_{aa} Q_h^\star(s,a) = R + B^\top P_{h+1} B$$

Since $R$ is s.p.d. (Definition 2.2), and $P_{h+1}$ is s.p.d. (by the inductive hypothesis), this sum must also be s.p.d., and so $Q_h^\star$ is indeed an upward-curved quadratic with respect to $a$. (If this isn't clear, try proving it as an exercise.) The proof of its upward curvature with respect to $s$ is equivalent.

**Lemma 2.1** ($\pi_h^\star$ is linear)**.** *Since $Q_h^\star$ is an upward-curved quadratic, finding its minimum over $a$ is easy: we simply set the gradient with respect to $a$ equal to zero and solve for $a$. First, we calculate the gradient:*

$$\begin{aligned} \nabla_a Q_h^\star(s,a) &= \nabla_a [a^\top(R + B^\top P_{h+1} B)a + 2s^\top A^\top P_{h+1} Ba] \\ &= 2(R + B^\top P_{h+1} B)a + 2(s^\top A^\top P_{h+1} B)^\top \end{aligned}$$

*Setting this to zero, we get*

$$\begin{aligned} 0 &= (R + B^\top P_{h+1} B)\pi_h^\star(s) + B^\top P_{h+1} As \\ \pi_h^\star(s) &= (R + B^\top P_{h+1} B)^{-1}(-B^\top P_{h+1} As) \\ &= -K_h s, \end{aligned}$$

*where*

$$K_h = (R + B^\top P_{h+1} B)^{-1} B^\top P_{h+1} A. \tag{2.1}$$

*Note that this optimal policy doesn't depend on the starting distribution $\mu_0$. It's also fully **deterministic** and isn't affected by the noise terms $w_0, \ldots, w_{H-1}$.*

**Lemma 2.2** (The value function is an upward-curved quadratic)**.** *Using the identity* $V_h^\star(s) = Q_h^\star(s, \pi^\star(s))$*, we have:*

$$\begin{aligned} V_h^\star(s) &= Q_h^\star(s, \pi^\star(s)) \\ &= s^\top (Q + A^\top P_{h+1} A)s + (-K_h s)^\top (R + B^\top P_{h+1} B)(-K_h s) + 2s^\top A^\top P_{h+1} B(-K_h s) \\ &\quad + \mathrm{Tr}(\sigma^2 P_{h+1}) + p_{h+1} \end{aligned}$$

*Note that with respect to* $s$*, this is the sum of a quadratic term and a constant, which is exactly what we were aiming for! The scalar term is clearly*

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

*We can simplify the quadratic term by substituting in* $K_h$ *from Equation 2.1. 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****:*

**Definition 2.5** (Riccati equation)**.**

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

*There are several nice properties to note about the Riccati equation:*

1. *It's defined* ***recursively.*** *Given the dynamics defined by* $A$ *and* $B$*, and the state cost matrix* $Q$*, we can recursively calculate* $P_h$ *across all timesteps starting from* $P_H = Q$*.*
2. $P_h$ *often appears in calculations surrounding optimality, such as* $V_h^\star, Q_h^\star$*, and* $\pi_h^\star$*.*
3. *Together with the dynamics given by* $A$ *and* $B$*, and the action coefficients* $R$ *in the lost function, it fully defines the optimal policy Lemma 2.1.*

*It remains to prove that* $V_h^\star$ curves upwards, *that is, that* $P_h$ *is s.p.d. We will use the following fact about* ***Schur complements:***

**Lemma 2.3** (Positive definiteness of Schur complements)**.** *Let*

$$D = \begin{pmatrix} A & B \\ B^\top & C \end{pmatrix}$$

*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*

$$D/A = C - B^\top A^{-1} B.$$

*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 p.d., so it suffices to show that*

$$S = P - PB(R + B^\top PB)^{-1}B^\top P$$

*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 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 p.d.*

Now we've shown that $V_h^\star(s) = s^\top P_h s + p_h$, where $P_h$ is s.p.d., proving the inductive hypothesis and completing the proof of Theorem 2.2 and Theorem 2.1.

In summary, we just demonstrated that at each timestep $h \in [H]$, the optimal value function $V_h^\star$ and optimal Q-function $Q_h^\star$ are both upward-curved quadratics and the optimal policy $\pi_h^\star$ is linear. We also showed that all of these quantities can be calculated using a sequence of s.p.d. matrices $P_0, \ldots, P_H$ that can be defined recursively using Definition 2.5.

Before we move on to some extensions of LQR, let's consider how the state at time $h$ behaves when we act according to this optimal policy.

### 2.4.1 Expected state at time $h$

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

$$\begin{aligned} s_h &= As_{h-1} + Ba_{h-1} + w_{h-1} \\ &= A(As_{h-2} + Ba_{h-2} + w_{h-2}) + Ba_{h-1} + w_{h-1} \\ &= \cdots \\ &= A^h s_0 + \sum_{i=0}^{h-1} A^i (Ba_{h-i-1} + w_{h-i-1}). \end{aligned}$$

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

$$\mathbb{E}[s_h \mid s_{0:(h-1)}, a_{0:(h-1)}] = A^h s_0 + \sum_{i=0}^{h-1} A^i B a_{h-i-1}. \tag{2.2}$$

**Exercise 2.2** (Expected state)**.** Show that if we choose actions according to the optimal policy Lemma 2.1, Equation 2.2 becomes

$$\mathbb{E}[s_h \mid s_0, a_i = \pi_i^\star(s_i) \quad \forall i \le h] = \left( \prod_{i=0}^{h-1} (A - BK_i) \right) s_0.$$

This introdces the quantity $A - BK_i$, which shows up frequently in control theory. For example, one important question is: will $s_h$ remain bounded, or will it go to infinity as time goes on? To answer this, let's imagine for simplicity that these $K_i$s are equal (call this matrix $K$). Then the expression above becomes $(A - BK)^h s_0$. Now consider the maximum eigenvalue $\lambda_{\max}$ of $A - BK$. If $|\lambda_{\max}| > 1$, then there's some nonzero initial state $\bar{s}_0$, the corresponding eigenvector, for which

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

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

## 2.5 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.

### 2.5.1 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 these sorts of problems, 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 2.6** (Time-dependent LQR)**.**

$$\begin{aligned}
\min_{\pi_0,\dots,\pi_{H-1}} \quad & \mathbb{E}\left[\left(\sum_{h=0}^{H-1}(s_h^\top Q_h s_h) + a_h^\top R_h a_h\right) + s_H^\top Q_H s_H\right] \\
\text{where} \quad & s_{h+1} = f_h(s_h, a_h, w_h) = A_h s_h + B_h a_h + w_h \\
& s_0 \sim \mu_0 \\
& a_h = \pi_h(s_h) \\
& w_h \sim \mathcal{N}(0, \sigma^2 I).
\end{aligned}$$

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 2.7** (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 (Definition 2.5), but with the time index added to each of the relevant matrices.

**Exercise 2.3** (Time dependent LQR proof)**.** Walk through the proof in Section 2.4 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.

### 2.5.2 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 $(s^\star, a^\star) = (0, 0)$. 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 $c_h$:

$$c_h(s_h, a_h) = (s_h^\top Q_h s_h + s_h^\top M_h a_h + a_h^\top R_h a_h) + (s_h^\top q_h + a_h^\top r_h) + c_h. \tag{2.3}$$

Similarly, we can also include a constant term $v_h \in \mathbb{R}^{n_s}$ in the dynamics (note that this is *deterministic* at each timestep, unlike the stochastic noise $w_h$):

$$s_{h+1} = f_h(s_h, a_h, w_h) = A_h s_h + B_h a_h + v_h + w_h.$$

**Exercise 2.4** (General cost function)**.** Derive the optimal solution. You will need to slightly modify the proof in Section 2.4.

### 2.5.3 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 $(s_h^\star, a_h^\star)_{h=0}^{H-1}$. To express this as a control problem, we'll need a corresponding time-dependent cost function:

$$c_h(s_h, a_h) = (s_h - s_h^\star)^\top Q (s_h - s_h^\star) + (a_h - a_h^\star)^\top R (a_h - a_h^\star).$$

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 Equation 2.3:

$$M_h = 0, \qquad q_h = -2Qs_h^\star, \qquad r_h = -2Ra_h^\star, \qquad c_h = (s_h^\star)^\top Q(s_h^\star) + (a_h^\star)^\top R(a_h^\star).$$

## 2.6 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 (Example 2.1). 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 2.8** (Nonlinear control problem)**.**

$$\begin{aligned} \min_{\pi_0,\dots,\pi_{H-1}:\mathcal{S}\to\mathcal{A}} \quad & \mathbb{E}_{s_0}\left[\sum_{h=0}^{H-1} c(s_h, a_h)\right] \\ \text{where} \quad & s_{h+1} = f(s_h, a_h) \\ & a_h = \pi_h(s_h) \\ & s_0 \sim \mu_0 \\ & c(s,a) = d(s,s^\star) + d(a,a^\star). \end{aligned}$$

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 Definition 2.1). 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?

### 2.6.1 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 $(s^\star, a^\star)$ and approximate it nearby with low-order polynomials (i.e. its Taylor approximation). In particular, as long as the dynamics $f$ are differentiable around $(s^\star, a^\star)$ and the cost function $c$ is twice differentiable at $(s^\star, a^\star)$, 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 $(s^\star, a^\star)$ gives:

$$f(s,a) \approx f(s^\star,a^\star) + \nabla_s f(s^\star,a^\star)(s-s^\star) + \nabla_a f(s^\star,a^\star)(a-a^\star)$$
$$(\nabla_s f(s,a))_{ij} = \frac{df_i(s,a)}{ds_j}, \quad i,j \le n_s \qquad (\nabla_a f(s,a))_{ij} = \frac{df_i(s,a)}{da_j}, \quad i \le n_s, j \le n_a$$

and quadratizing the cost function around $(s^\star, a^\star)$ gives:

$$\begin{aligned} c(s,a) \approx\ & c(s^\star,a^\star) \quad \text{constant term} \\ & + \nabla_s c(s^\star,a^\star)(s-s^\star) + \nabla_a c(s^\star,a^\star)(a-a^\star) \quad \text{linear terms} \\ & \left.\begin{aligned} & + \frac{1}{2}(s-s^\star)^\top \nabla_{ss} c(s^\star,a^\star)(s-s^\star) \\ & + \frac{1}{2}(a-a^\star)^\top \nabla_{aa} c(s^\star,a^\star)(a-a^\star) \\ & + (s-s^\star)^\top \nabla_{sa} c(s^\star,a^\star)(a-a^\star) \end{aligned}\right\} \text{quadratic terms} \end{aligned}$$

where the gradients and Hessians are defined as

$$(\nabla_s c(s,a))_i = \frac{dc(s,a)}{ds_i}, \quad i \le n_s \qquad (\nabla_a c(s,a))_i = \frac{dc(s,a)}{da_i}, \quad i \le n_a$$
$$(\nabla_{ss} c(s,a))_{ij} = \frac{d^2c(s,a)}{ds_i ds_j}, \quad i,j \le n_s \qquad (\nabla_{aa} c(s,a))_{ij} = \frac{d^2c(s,a)}{da_i da_j}, \quad i,j \le n_a$$
$$(\nabla_{sa} c(s,a))_{ij} = \frac{d^2c(s,a)}{ds_i da_j}. \quad i \le n_s, j \le n_a$$

**Exercise:** Note that this cost can be expressed in the general quadratic form seen in Equation 2.3. Derive the corresponding quantities $Q, R, M, q, r, c$.

### 2.6.2 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 $\delta$ to the input.

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

Note that 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.

### 2.6.3 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 Section 2.4.

But what happens when we enter states far away from $s^\star$ or want to use actions far from $a^\star$? 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.

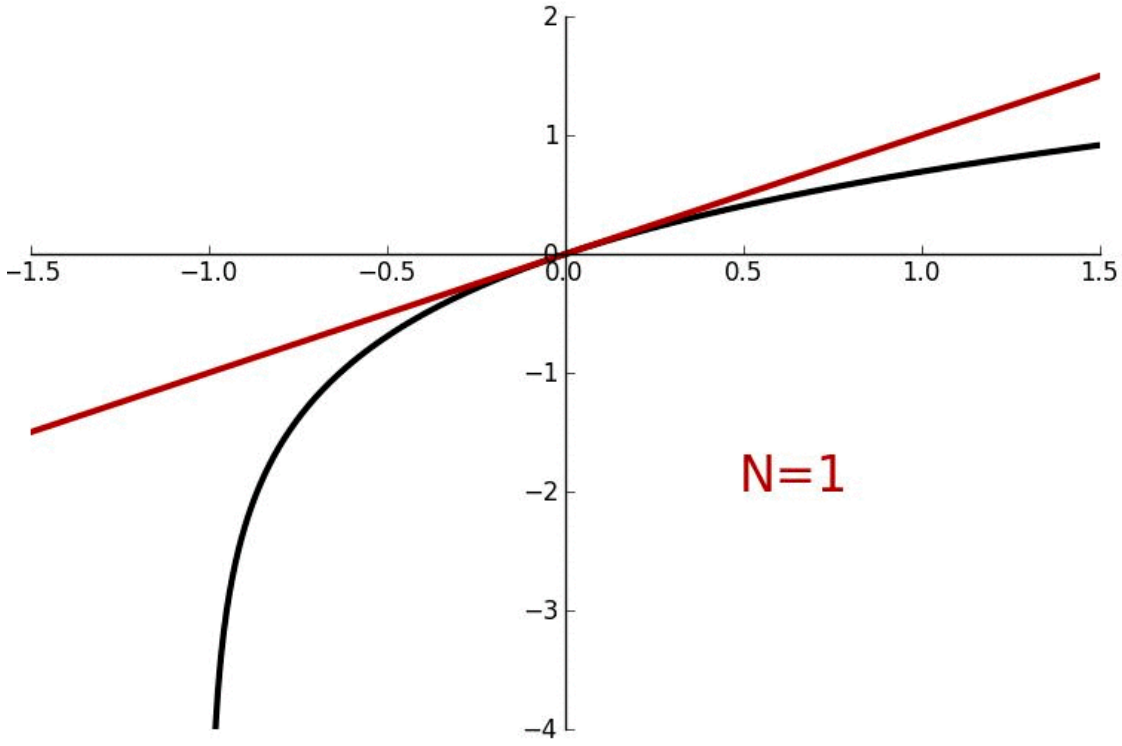



Figure 2.4: Local linearization might only be accurate in a small region around the point of linearization

### 2.6.4 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 2.9** (Iterative LQR)**.** For each iteration of the algorithm:

1. Form a time-dependent LQR problem around the current candidate trajectory using local linearization.
2. Compute the optimal policy using Section 2.5.1.
3. Generate a new series of actions using this policy.

4. Compute a better candidate trajectory by interpolating between the current and proposed actions.

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 $\bar{s}_0 = \mathbb{E}_{s_0 \sim \mu_0}[s_0]$ as the expected initial state.

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

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

$$\begin{aligned} f_h(s,a) &\approx f(\bar{s}_h^i, \bar{a}_h^i) + \nabla_s f(\bar{s}_h^i, \bar{a}_h^i)(s - \bar{s}_h^i) + \nabla_a f(\bar{s}_h^i, \bar{a}_h^i)(a - \bar{a}_h^i) \\ c_h(s,a) &\approx c(\bar{s}_h^i, \bar{a}_h^i) + \begin{bmatrix} s - \bar{s}_h^i & a - \bar{a}_h^i \end{bmatrix} \begin{bmatrix} \nabla_s c(\bar{s}_h^i, \bar{a}_h^i) \\ \nabla_a c(\bar{s}_h^i, \bar{a}_h^i) \end{bmatrix} \\ &\quad + \frac{1}{2} \begin{bmatrix} s - \bar{s}_h^i & a - \bar{a}_h^i \end{bmatrix} \begin{bmatrix} \nabla_{ss} c(\bar{s}_h^i, \bar{a}_h^i) & \nabla_{sa} c(\bar{s}_h^i, \bar{a}_h^i) \\ \nabla_{as} c(\bar{s}_h^i, \bar{a}_h^i) & \nabla_{aa} c(\bar{s}_h^i, \bar{a}_h^i) \end{bmatrix} \begin{bmatrix} s - \bar{s}_h^i \\ a - \bar{a}_h^i \end{bmatrix}. \end{aligned}$$

**Step 2: Compute the optimal policy.** We can now solve the time-dependent LQR problem using the Riccati equation from Section 2.5.1 to compute the optimal policy $\pi_0^i, \dots, \pi_{H-1}^i$.

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

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

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 $\tilde{a}_h$ and aren't directly using them for the next iteration $\bar{a}_h^{i+1}$. Rather, we want to *interpolate* between them and the actions from the previous iteration $\bar{a}_0^i, \dots, \bar{a}_{H-1}^i$. 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. (Can you think of an intuitive example where this might happen?)

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

$$\begin{aligned} \min_{\alpha \in [0,1]} \quad & \sum_{h=0}^{H-1} c(s_h, \bar{a}_h^{i+1}) \\ \text{where} \quad & s_{h+1} = f(s_h, \bar{a}_h^{i+1}) \\ & \bar{a}_h^{i+1} = \alpha \bar{a}_h^i + (1-\alpha)\tilde{a}_h \\ & s_0 = \bar{s}_0. \end{aligned}$$

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 $n_{\text{steps}}$ 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).

## 2.7 Summary

This chapter introduced some approaches to solving different variants of the optimal control problem Definition 2.1. We began with the simple case of linear dynamics and an upward-curved quadratic cost. This model is called the LQR and we solved for the optimal policy using dynamic programming. We then extended these results to the more general nonlinear case via local linearization. We finally saw the iterative LQR algorithm for solving nonlinear control problems.

# 3 Multi-Armed Bandits

## 3.1 Introduction

The **multi-armed bandits** (MAB) setting 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.

In particular, we'll spend a lot of time discussing 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?

**Example 3.1** (Online advertising)**.** Let's suppose you, the agent, are an advertising company. You have $K$ different ads that you can show to users; For concreteness, let's suppose there's just 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.

**Example 3.2** (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 to 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.

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.

```
from jaxtyping import Float, Array
import numpy as np
import latexify
from typing import Callable, Union
import matplotlib.pyplot as plt

import solutions.bandits as solutions

np.random.seed(184)

def random_argmax(ary: Array) -> int:
    """Take an argmax and randomize between ties."""
    max_idx = np.flatnonzero(ary == ary.max())
    return np.random.choice(max_idx).item()


# used as decorator
latex = latexify.algorithmic(
    trim_prefixes={"mab"},
    id_to_latex={"arm": "a_t", "reward": "r", "means": "\mu"},
    use_math_symbols=True,
)
```

*Remark* 3.1 (Namesake). The name "multi-armed bandits" 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.

Let $K$ denote the number of arms. We'll label them $0, \dots, K-1$ and use *superscripts* to indicate the arm index; since we seldom need to raise a number to a power, this won't cause much confusion. In this chapter, we'll consider the **Bernoulli bandit** setting from the examples above, where arm $k$ either returns reward 1 with probability $\mu^k$ or 0 otherwise. The agent gets to pull an arm $T$ times in total. We can formalize the Bernoulli bandit in the following Python code:

```
class MAB:
    """The Bernoulli multi-armed bandit environment.

    :param means: the means (success probabilities) of the reward distributions for each arm
    :param T: the time horizon
    """

    def __init__(self, means: Float[Array, " K"], T: int):
```

```
        assert all(0 <= p <= 1 for p in means)
        self.means = means
        self.T = T
        self.K = self.means.size
        self.best_arm = random_argmax(self.means)

    def pull(self, k: int) -> int:
        """Pull the `k`-th arm and sample from its (Bernoulli) reward distribution."""
        reward = np.random.rand() < self.means[k].item()
        return +reward
```

```
mab = MAB(means=np.array([0.1, 0.8, 0.4]), T=100)
```

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

```
@latex
def mab_loop(mab: MAB, agent: "Agent") -> int:
    for t in range(mab.T):
        arm = agent.choose_arm()  # in 0, ..., K-1
        reward = mab.pull(arm)
        agent.update_history(arm, reward)


mab_loop
```

**function** mab_loop(mab : MAB, agent : "Agent")
  **for** $t \in \text{range}(T)$ **do**
    $a_t \leftarrow$ agent.choose_arm()
    $r \leftarrow \text{pull}(a_t)$
    agent.update_history$(a_t, r)$
  **end for**
**end function**

The `Agent` class stores the pull history and uses it to decide which arm to pull next. Since we are working with Bernoulli bandits, we can summarize the pull history concisely in a $\mathbb{N}^{K\times 2}$ array.

```
class Agent:
    def __init__(self, K: int, T: int):
        """The MAB agent that decides how to choose an arm given the past history."""
        self.K = K
```

```
        self.T = T
        self.rewards = []  # for plotting
        self.choices = []
        self.history = np.zeros((K, 2), dtype=int)

    def choose_arm(self) -> int:
        """Choose an arm of the MAB. Algorithm-specific."""
        ...

    def count(self) -> int:
        """The number of pulls made. Also the current step index."""
        return len(self.rewards)

    def update_history(self, arm: int, reward: int):
        self.rewards.append(reward)
        self.choices.append(arm)
        self.history[arm, reward] += 1
```

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:

$$\mu^\star := \max_{k \in [K]} \mu^k.$$

The goal, then, can be rephrased as to minimize the **regret**, defined below:

**Definition 3.1** (Regret)**.** The agent's **regret** after $T$ timesteps is defined as

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

```
def regret_per_step(mab: MAB, agent: Agent):
    """Get the difference from the average reward of the optimal arm. The sum of these is the
    return [mab.means[mab.best_arm] - mab.means[arm] for arm in agent.choices]
```

Note that this depends on the *true means* of the pulled arms, *not* the actual observed rewards. We typically think of this as a random variable where the randomness comes from the agent's strategy (i.e. the sequence of actions $a_0, \dots, a_{T-1}$).

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 $\mathbb{E}[\text{Regret}_T] \leq M_T$.
2. Find a *high-probability* upper bound on the regret, i.e. show $\mathbb{P}(\text{Regret}_T \leq M_{T,\delta}) \geq 1 - \delta$.

Note that these two different approaches say very different things about the regret. The first approach says that the *average* regret is at most $M_T$. However, the agent might still achieve higher regret on many runs. The second approach says that, *with high probability*, the agent will achieve regret at most $M_{T,\delta}$. However, it doesn't say anything about the regret in the remaining $\delta$ fraction of runs, which might be arbitrarily high.

We'd like to achieve **sublinear regret** in expectation, i.e. $\mathbb{E}[\text{Regret}_T] = o(T)$. That is, as we learn more about the environment, we'd like to be able to exploit that knowledge to take the optimal arm as often as possible.

The rest of the chapter comprises a series of increasingly sophisticated MAB algorithms.

```
def plot_strategy(mab: MAB, agent: Agent):
    plt.figure(figsize=(10, 6))

    # plot reward and cumulative regret
    plt.plot(np.arange(mab.T), np.cumsum(agent.rewards), label="reward")
    cum_regret = np.cumsum(regret_per_step(mab, agent))
    plt.plot(np.arange(mab.T), cum_regret, label="cumulative regret")

    # draw colored circles for arm choices
    colors = ["red", "green", "blue"]
    color_array = [colors[k] for k in agent.choices]
    plt.scatter(np.arange(mab.T), np.zeros(mab.T), c=color_array, label="arm")

    # labels and title
    plt.xlabel("timestep")
    plt.legend()
    plt.title(f"{agent.__class__.__name__} reward and regret")
    plt.show()
```

## 3.2 Pure exploration

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

```
class PureExploration(Agent):
    def choose_arm(self):
        """Choose an arm uniformly at random."""
        return solutions.pure_exploration_choose_arm(self)
```

Note that

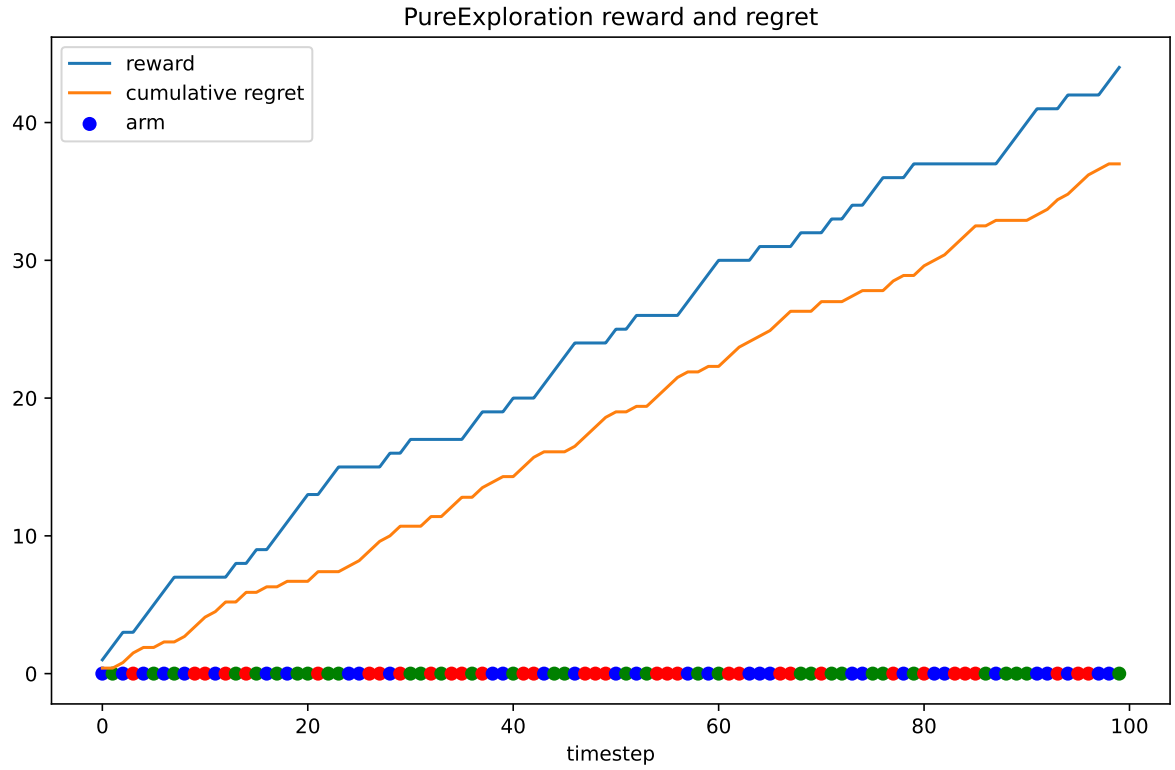
$$\mathbb{E}_{a_t \sim \text{Unif}([K])}[\mu^{a_t}] = \bar{\mu} = \frac{1}{K}\sum_{k=1}^{K} \mu^k$$

so the expected regret is simply

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

This scales as $\Theta(T)$, i.e. *linear* in the number of timesteps $T$. There's no learning here: the agent doesn't use any information about the environment to improve its strategy. You can see that the distribution over its arm choices always appears "(uniformly) random".

```
agent = PureExploration(mab.K, mab.T)
mab_loop(mab, agent)
plot_strategy(mab, agent)
```



## 3.3 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.

```
class PureGreedy(Agent):
    def choose_arm(self):
        """Choose the arm with the highest observed reward on its first pull."""
        return solutions.pure_greedy_choose_arm(self)
```

Note we've used superscripts $r^k$ during the exploration phase to indicate that we observe exactly one reward for each arm. Then we use subscripts $r_t$ during the exploitation phase to indicate that we observe a sequence of rewards from the chosen greedy arm $\hat{k}$.

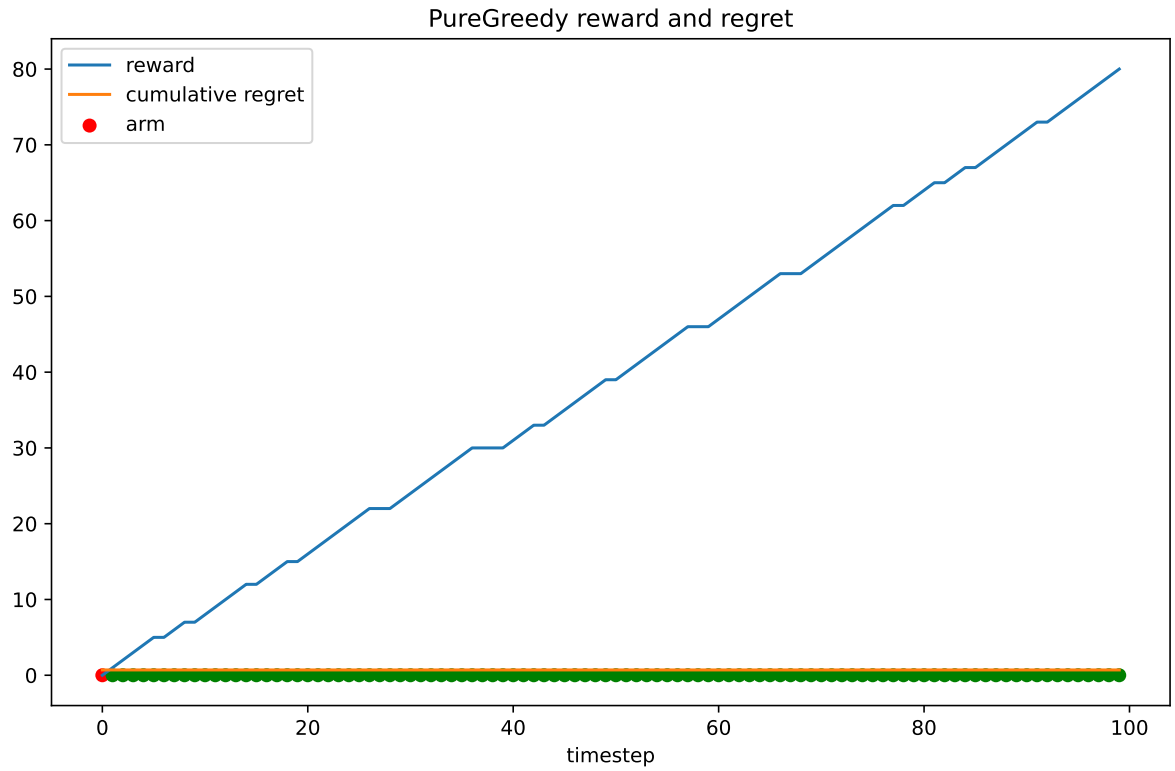
How does the expected regret of this strategy compare to that of pure exploration? We'll do a more general analysis in the following section. Now, for intuition, suppose there's just $K = 2$ arms, with Bernoulli reward distributions with means $\mu^0 > \mu^1$.

Let's let $r^0$ be the random reward from the first arm and $r^1$ be the random reward from the second. If $r^0 > r^1$, then we achieve zero regret. Otherwise, we achieve regret $T(\mu^0 - \mu^1)$. Thus, the expected regret is simply:

$$\begin{aligned}\mathbb{E}[\text{Regret}_T] &= \mathbb{P}(r^0 < r^1) \cdot T(\mu^0 - \mu^1) + c \\ &= (1 - \mu^0)\mu^1 \cdot T(\mu^0 - \mu^1) + c\end{aligned}$$

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

```
agent = PureGreedy(mab.K, mab.T)
mab_loop(mab, agent)
plot_strategy(mab, agent)
```



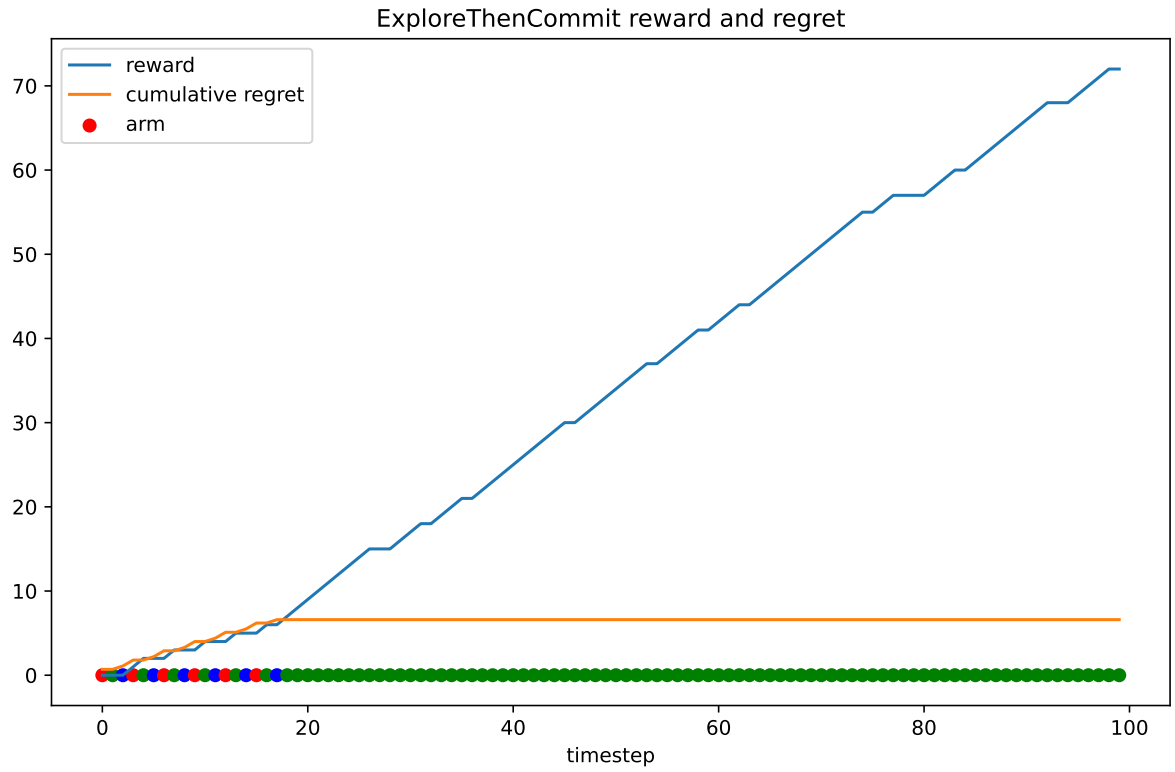

The cumulative regret is a straight line because the regret only depends on the arms chosen and not the actual reward observed. In fact, if the greedy algorithm happens to get lucky on the first set of pulls, it may act entirely optimally for that episode! But its *average* regret is what measures its effectiveness.

## 3.4 Explore-then-commit

We can improve the pure greedy algorithm as follows: let's reduce the variance of the reward estimates by pulling each arm $N_{\text{explore}} > 1$ times before committing. This is called the **explore-then-commit** strategy. Note that the "pure greedy" strategy above is just the special case where $N_{\text{explore}} = 1$.

```
class ExploreThenCommit(Agent):
    def __init__(self, K: int, T: int, N_explore: int):
        super().__init__(K, T)
        self.N_explore = N_explore

    def choose_arm(self):
        return solutions.etc_choose_arm(self)
```

```
agent = ExploreThenCommit(mab.K, mab.T, mab.T // 15)
mab_loop(mab, agent)
plot_strategy(mab, agent)
```




Notice that now, the graphs are much more consistent, and the algorithm finds the true optimal arm and sticks with it much more frequently. We would expect ETC to then have a better (i.e. lower) average regret. Can we prove this?

### 3.4.1 ETC regret analysis

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

#### 3.4.1.1 Exploration phase.

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

#### 3.4.1.2 Exploitation phase.

This will take a bit more effort. We'll prove that for any total time $T$, we can choose $N_{\text{explore}}$ 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

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

So we'd like to bound $\mu^\star - \mu^{\hat{k}} = o(1)$ (as a function of $T$) in order to achieve sublinear regret. How can we do this?

Let's define $\Delta^k = \hat{\mu}^k - \mu^k$ 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 3.1** (Hoeffding's inequality)**.** *Let $X_0, \ldots, X_{n-1}$ be i.i.d. random variables with $X_i \in [0,1]$ almost surely for each $i \in [n]$. Then for any $\delta > 0$,*

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

The proof of this inequality is beyond the scope of this book. See Vershynin (2018) Chapter 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.:

$$\mathbb{P}\left(|\Delta^k| > \sqrt{\frac{\ln(2/\delta)}{2N_{\text{explore}}}}\right) \leq \delta. \tag{3.1}$$

But note that we can't apply this to arm $\hat{k}$ directly since $\hat{k}$ is itself a random variable. Instead, we need to "uniform-ize" this bound across *all* the arms, i.e. 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:

**Theorem 3.2** (Union bound)**.** *Consider a set of events $A_0, \dots, A_{n-1}$. Then*

$$\mathbb{P}(\exists i \in [n].A_i) \leq \sum_{i=0}^{n-1} \mathbb{P}(A_i).$$

*In particular, if $\mathbb{P}(A_i) \geq 1 - \delta$ for each $i \in [n]$, we have*

$$\mathbb{P}(\forall i \in [n].A_i) \geq 1 - n\delta.$$

**Exercise:** Prove the second statement above.

Applying the union bound across the arms for the l.h.s. event of Equation 3.1, we have

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

Then to apply this bound to $\hat{k}$ in particular, we can apply the useful trick of "adding zero":

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

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

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

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

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

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

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

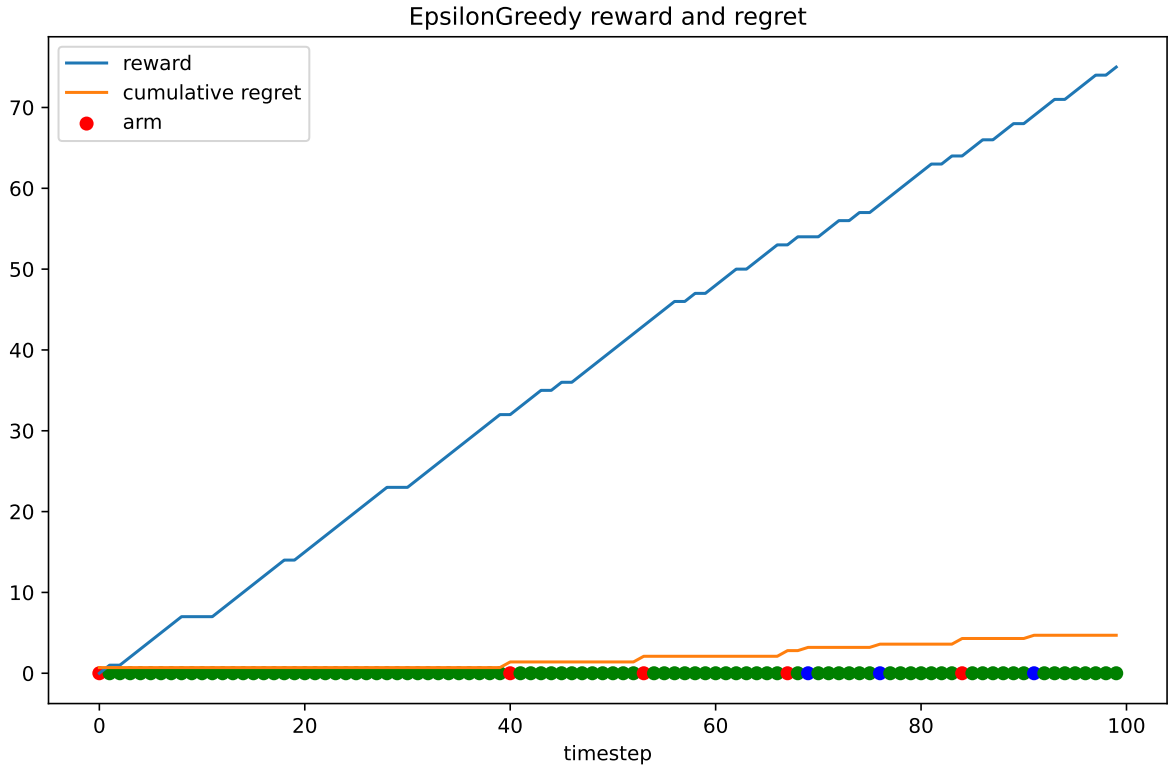
The ETC algorithm is rather "abrupt" in that it switches from exploration to exploitation after a fixed number of timesteps. In practice, it's often better to use a more gradual transition, which brings us to the *epsilon-greedy* algorithm.

## 3.5 Epsilon-greedy

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

```
class EpsilonGreedy(Agent):
    def __init__(
        self,
        K: int,
        T: int,
         _array: Float[Array, " T"],
    ):
        super().__init__(K, T)
        self._array = _array

    def choose_arm(self):
        return solutions.epsilon_greedy_choose_arm(self)
```

```
agent = EpsilonGreedy(mab.K, mab.T, np.full(mab.T, 0.1))
mab_loop(mab, agent)
plot_strategy(mab, agent)
```




Note that we let $\epsilon$ vary over time. In particular, we might want to gradually *decrease* $\epsilon$ as we learn more about the reward distributions and no longer need to spend time exploring.

What is the expected regret of the algorithm if we set $\epsilon$ to be a constant?

It turns out that setting $\epsilon_t = \sqrt[3]{K \ln(t)/t}$ also achieves a regret of $\tilde{O}(t^{2/3}K^{1/3})$ (ignoring the logarithmic factors). (We will not prove this here; a proof can be found in (Agarwal et al. 2022).)

In ETC, we had to set $N_{\text{explore}}$ based on the total number of timesteps $T$. But the epsilon-greedy algorithm actually handles the exploration *automatically*: the regret rate holds for *any* $t$, and doesn't depend on the final horizon $T$.

But the way these algorithms explore is rather naive: we've been exploring *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?

## 3.6 Upper Confidence Bound (UCB)

To quantify how *certain* we are about the mean of each arm, we'll compute *confidence intervals* for our estimators, and then choose the arm with the highest *upper confidence bound*. 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.

In particular, for each arm $k$ at time $t$, we'd like to compute some upper confidence bound $M_t^k$ such that $\hat{\mu}_t^k \leq M_t^k$ with high probability, and then choose $a_t := \arg\max_{k \in [K]} M_t^k$. But how should we compute $M_t^k$?

In Section 3.4.1, we were able to compute this bound using Hoeffding's inequality, which assumes that the number of samples is *fixed*. This was the case in ETC (where we pull each arm $N_{\text{explore}}$ times), but in UCB, the number of times we pull each arm depends on the agent's actions, which in turn depend on the random rewards and are therefore stochastic. So we *can't* use Hoeffding's inequality directly.

Instead, we'll apply the same trick we used in the ETC analysis: we'll use the **union bound** to compute a *looser* bound that holds *uniformly* across all timesteps and arms. Let's introduce some notation to discuss this.

Let $N_t^k$ denote the (random) number of times arm $k$ has been pulled within the first $t$ timesteps, and $\hat{\mu}_t^k$ denote the sample average of those pulls. That is,

$$N_t^k := \sum_{\tau=0}^{t-1} \mathbf{1}\{a_\tau = k\}$$
$$\hat{\mu}_t^k := \frac{1}{N_t^k} \sum_{\tau=0}^{t-1} \mathbf{1}\{a_\tau = k\} r_\tau.$$

To achieve the "fixed sample size" assumption, we'll need to shift our index from *time* to *number of samples from each arm*. In particular, we'll define $\tilde{r}_n^k$ to be the $n$th sample from arm $k$, and $\tilde{\mu}_n^k$ to be the sample average of the first $n$ samples from arm $k$. Then, for a fixed $n$, this satisfies the "fixed sample size" assumption, and we can apply Hoeffding's inequality to get a bound on $\tilde{\mu}_n^k$.

So how can we extend our bound on $\tilde{\mu}_n^k$ to $\hat{\mu}_t^k$? Well, we know $N_t^k \leq t$ (where equality would be the case if and only if we had pulled arm $k$ every time). So we can apply the same trick as last time, where we uniform-ize across all possible values of $N_t^k$:

$$\mathbb{P}\left(\forall n \leq t, |\tilde{\mu}_n^k - \mu^k| \leq \sqrt{\frac{\ln(2/\delta)}{2n}}\right) \geq 1 - t\delta.$$

In particular, since $N_t^k \leq t$, and $\tilde{\mu}_{N_t^k}^k = \hat{\mu}_t^k$ by definition, we have

$$\mathbb{P}\left(|\hat{\mu}_t^k - \mu^k| \leq \sqrt{\frac{\ln(2t/\delta')}{2N_t^k}}\right) \geq 1 - \delta' \text{ where } \delta' := t\delta.$$

This bound would then suffice for applying the UCB algorithm! That is, the upper confidence bound for arm $k$ would be

$$M_t^k := \hat{\mu}_t^k + \sqrt{\frac{\ln(2t/\delta')}{2N_t^k}},$$

where we can choose $\delta'$ depending on how tight we want the interval to be.

- A smaller $\delta'$ would give us a larger and higher-confidence interval, emphasizing the exploration term.
- A larger $\delta'$ would give a tighter and lower-confidence interval, prioritizing the current sample averages.

We can now use this to define the UCB algorithm.

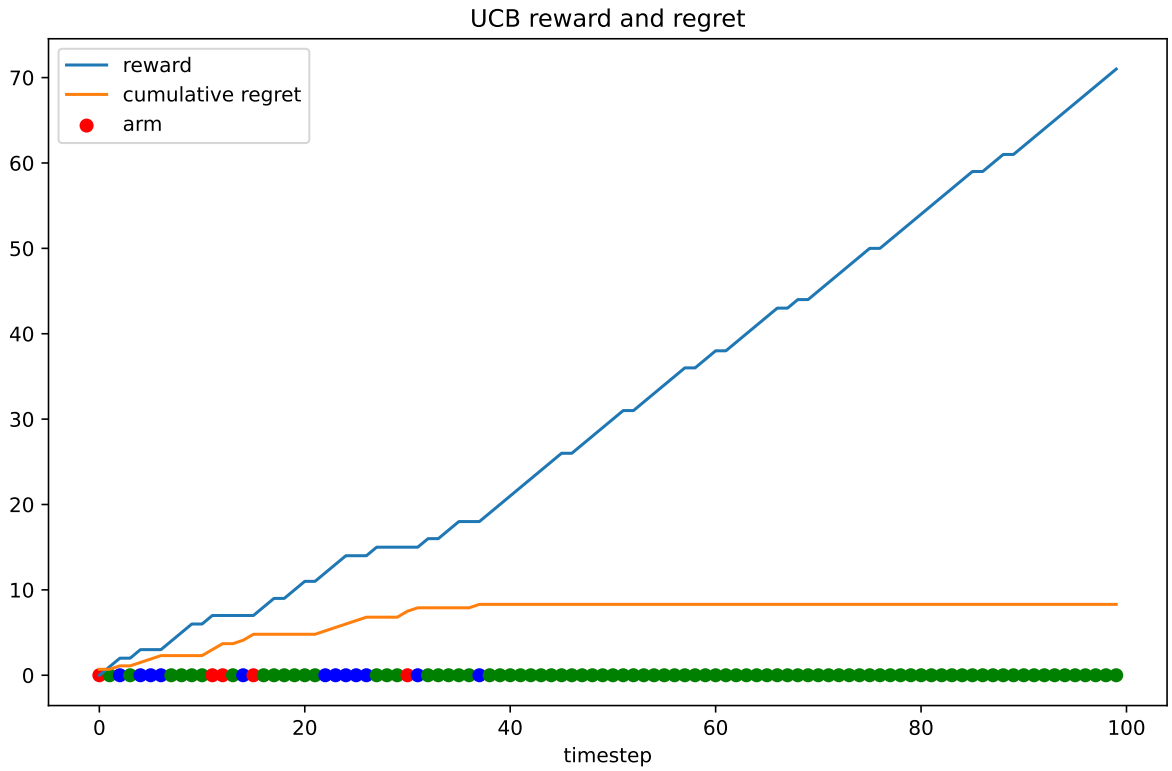
```
class UCB(Agent):
    def __init__(self, K: int, T: int, delta: float):
        super().__init__(K, T)
        self.delta = delta

    def choose_arm(self):
        return solutions.ucb_choose_arm(self)
```

Intuitively, UCB prioritizes arms where:

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

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

```
agent = UCB(mab.K, mab.T, 0.9)
mab_loop(mab, agent)
plot_strategy(mab, agent)
```




### 3.6.1 UCB regret analysis

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).

$$\mathbb{P}\left(\forall k \leq K, t < T. |\hat{\mu}_t^k - \mu^k| \leq B_t^k\right) \geq 1 - \delta''$$

$$\text{where} \quad B_t^k := \sqrt{\frac{\ln(2TK/\delta'')}{2N_t^k}}.$$

Intuitively, $B_t^k$ 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:

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

Summing this across timesteps gives

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

Putting everything together gives

$$\begin{aligned}
\text{Regret}_T &\le 2K\sqrt{2T\ln(2TK/\delta'')} \quad \text{with probability } 1-\delta'' \\
&= \tilde{O}(K\sqrt{T})
\end{aligned}$$

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

### 3.6.2 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

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

## 3.7 Thompson sampling and Bayesian bandits

So far, we've treated the parameters $\mu^0, \ldots, \mu^{K-1}$ 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!

```
class Distribution:
    def sample(self) -> Float[Array, " K"]:
        """Sample a vector of means for the K arms."""
        ...

    def update(self, arm: int, reward: float):
        """Condition on obtaining `reward` from the given arm."""
        ...
```

```
class ThompsonSampling(Agent):
    def __init__(self, K: int, T: int, prior: Distribution):
        super().__init__(K, T)
        self.distribution = prior

    def choose_arm(self):
        means = self.distribution.sample()
```

```
        return random_argmax(means)

    def update_history(self, arm: int, reward: int):
        super().update_history(arm, reward)
        self.distribution.update(arm, reward)
```

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

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

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

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

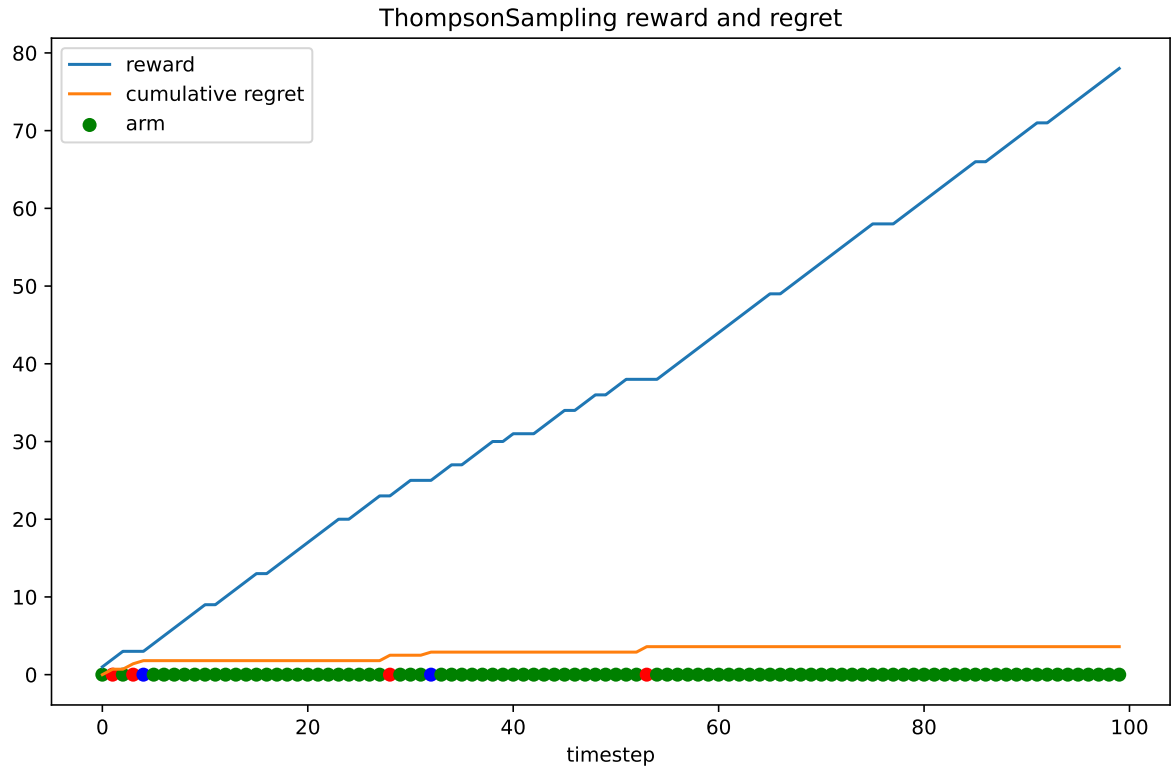
$$\begin{aligned} \mathbb{P}(\mu \mid a_0, r_0) &\propto \mathbb{P}(r_0 \mid a_0, \mu)\mathbb{P}(a_0 \mid \mu)\mathbb{P}(\mu) \\ &\propto (\mu^{a_0})^{r_0}(1-\mu^{a_0})^{1-r_0}. \end{aligned}$$

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 $\mu^k$ (note that Unif$([0,1])$ = Beta$(1,1)$), then the posterior, after conditioning on samples from Bern$(\mu^k)$, 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.

```
class Beta(Distribution):
    def __init__(self, K: int, alpha: int = 1, beta: int = 1):
        self.alphas = np.full(K, alpha)
        self.betas = np.full(K, beta)
```

```
    def sample(self):
        return np.random.beta(self.alphas, self.betas)

    def update(self, arm: int, reward: int):
        self.alphas[arm] += reward
        self.betas[arm] += 1 - reward
```

```
beta_distribution = Beta(mab.K)
agent = ThompsonSampling(mab.K, mab.T, beta_distribution)
mab_loop(mab, agent)
plot_strategy(mab, agent)
```




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

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

where

$$\mathrm{KL}(\mu^k \parallel \mu^\star) = \mu^k \ln \frac{\mu^k}{\mu^\star} + (1-\mu^k) \ln \frac{1-\mu^k}{1-\mu^\star}$$

measures the **Kullback-Leibler divergence** from the Bernoulli distribution with mean $\mu^k$ 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.

## 3.8 Contextual bandits

In the above MAB environment, the reward distributions of the arms remain constant. However, in many real-world settings, we might receive additional information that affects these distributions. For example, in the online advertising case where each arm corresponds to an ad we could show the user, we might receive information about the user's preferences that changes how likely they are to click on a given ad. We can model such environments using **contextual bandits**.

**Definition 3.2** (Contextual bandit)**.** At each timestep $t$, a new *context* $x_t$ is drawn from some distribution $\nu_{\mathrm{x}}$. The learner gets to observe the context, and choose an action $a_t$ according to some context-dependent policy $\pi_t(x_t)$. Then, the learner observes the reward from the chosen arm $r_t \sim \nu^{a_t}(x_t)$. The reward distribution also depends on the context.

Assuming our context is *discrete*, we can just perform the same algorithms, treating each context-arm pair as its own arm. This gives us an enlarged MAB of $K|\mathcal{X}|$ arms.

Write down the UCB algorithm for this enlarged MAB. That is, write an expression for $\pi_t(x_t) = \arg\max_a \ldots$.

Recall that running UCB for $T$ timesteps on an MAB with $K$ arms achieves a regret bound of $\tilde{O}(\sqrt{TK})$. So in this problem, we would achieve regret $\tilde{O}(\sqrt{TK|\mathcal{X}|})$ in the contextual MAB, which has a polynomial dependence on $|\mathcal{X}|$. But in a situation where we have large, or even infinitely many contexts, e.g. in the case where our context is a continuous value, this becomes intractable.

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: for example, we might want to advertise similar products to users with similar preferences. How can we incorporate this structure into our solution?

### 3.8.1 Linear contextual bandits

We want to model the *mean reward* of arm $k$ as a function of the context, i.e. $\mu^k(x)$. One simple model is the *linear* one: $\mu^k(x) = x^\top \theta^k$, where $x \in \mathcal{X} = \mathbb{R}^d$ and $\theta^k \in \mathbb{R}^d$ describes a *feature direction* for arm $k$. Recall that **supervised learning** gives us a way to estimate a conditional expectation from samples: We learn a *least squares* estimator from the timesteps where arm $k$ was selected:

$$\hat{\theta}_t^k = \arg\min_{\theta \in \mathbb{R}^d} \sum_{\{i \in [t]: a_i = k\}} (r_i - x_i^\top \theta)^2.$$

This has the closed-form solution known as the *ordinary least squares* (OLS) estimator:

$$\begin{aligned} \hat{\theta}_t^k &= (A_t^k)^{-1} \sum_{\{i \in [t]: a_i = k\}} x_i r_i \\ \text{where} \quad A_t^k &= \sum_{\{i \in [t]: a_i = k\}} x_i x_i^\top. \end{aligned} \tag{3.2}$$

We can now apply the UCB algorithm in this environment in order to balance *exploration* of new arms and *exploitation* of arms that we believe to have high reward. But how should we construct the upper confidence bound? 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 Equation 3.2. Instead, we'll use **Chebyshev's inequality** to construct an upper confidence bound.

**Theorem 3.3** (Chebyshev's inequality)**.** *For a random variable $Y$ such that $\mathbb{E}Y = 0$ and $\mathbb{E}Y^2 = \sigma^2$,*

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

Since the OLS estimator is known to be unbiased (try proving this yourself), we can apply Chebyshev's inequality to $x_t^\top(\hat{\theta}_t^k - \theta^k)$:

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

We haven't explained why $x_t^\top (A_t^k)^{-1} x_t$ is the correct expression for the variance of $x_t^\top \hat{\theta}_t^k$. This result follows from some algebra on the definition of the OLS estimator Equation 3.2.

The first term is exactly our predicted reward $\hat{\mu}_t^k(x_t)$. To interpret the second term, note that

$$x_t^\top (A_t^k)^{-1} x_t = \frac{1}{N_t^k} x_t^\top (\Sigma_t^k)^{-1} x_t,$$

where

$$\Sigma_t^k = \frac{1}{N_t^k} \sum_{\{i \in [t] : a_i = k\}} x_i x_i^\top$$

is the empirical covariance matrix of the contexts (assuming that the context has mean zero). That is, the learner is encouraged to choose arms 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.

We can now substitute these quantities into UCB to get the **LinUCB** algorithm:

```
class LinUCBPseudocode(Agent):
    def __init__(
        self, K: int, T: int, D: int, lam: float, get_c: Callable[[int], float]
    ):
        super().__init__(K, T)
        self.lam = lam
        self.get_c = get_c
        self.contexts = [None for _ in range(K)]
        self.A = np.repeat(lam * np.eye(D)[...], K)  # regularization
        self.targets = np.zeros(K, D)
        self.w = np.zeros(K, D)

    def choose_arm(self, context: Float[Array, " D"]):
        c = self.get_c(self.count)
        scores = self.w @ context + c * np.sqrt(
            context.T @ np.linalg.solve(self.A, context)
        )
        return random_argmax(scores)

    def update_history(self, context: Float[Array, " D"], arm: int, reward: int):
        self.A[arm] += np.outer(context, context)
        self.targets[arm] += context * reward
        self.w[arm] = np.linalg.solve(self.A[arm], self.targets[arm])
```

Note that the matrix $A_t^k$ above might not be invertible. When does this occur? One way to address this is to include a $\lambda I$ regularization term to ensure that $A_t^k$ is invertible. This is equivalent to solving a *ridge regression* problem instead of the unregularized least squares problem. Implement this solution.

$c_t$ is similar to the $\log(2t/\delta')$ term of UCB: It controls the width of the confidence interval. Here, we treat it as a tunable parameter, though in a theoretical analysis, it would depend on $A_t^k$ and the probability $\delta$ with which the bound holds.

Using similar tools for UCB, we can also prove an $\tilde{O}(\sqrt{T})$ regret bound. The full details of the analysis can be found in Section 3 of (Agarwal et al. 2022).

## 3.9 Summary

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 Supervised learning

## 4.1 Introduction

```
from utils import Float, Array, Callable, plt, np, latex
from torchvision.datasets import MNIST
from config import MNIST_PATH
```

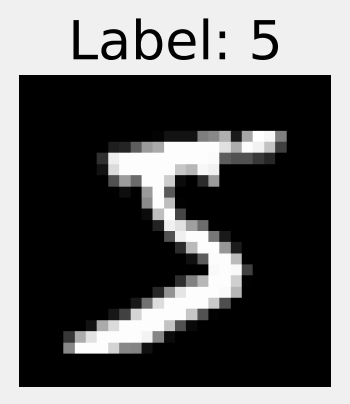
**Supervised learning** (SL) is a core subfield of machine learning alongside RL and unsupervised learning. The typical SL task is to approximate an unknown function given a dataset of input-output pairs from that function.

**Example 4.1** (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.

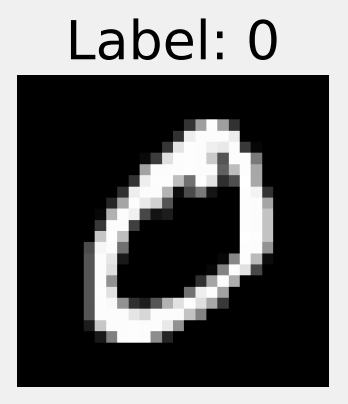
Figure 4.1 illustrates two samples (that is, input-output pairs) from the MNIST database of handwritten digits (Deng 2012). This is a task that most humans can easily accomplish. By providing many samples of digits and their labels to a machine, SL algorithms can learn to solve this task as well.

```
data = MNIST(MNIST_PATH, train=True, download=True)

plt.axis('off')
plt.imshow(data.data[0], cmap='gray')
plt.title(f"Label: {data.targets[0]}")
plt.gcf().set_size_inches(2, 2)
plt.show()

plt.axis('off')
```

```
plt.imshow(data.data[1], cmap='gray')
plt.title(f"Label: {data.targets[1]}")
plt.gcf().set_size_inches(2, 2)
plt.show()
```




(a) A handwritten digit.




(b) Another handwritten digit.

Figure 4.1: The MNIST image classification dataset of handwritten digits.

Where might function approximation be useful in RL? There are many functions involved in the definition of an MDP (Definition 1.2), such as the state transitions $P$ or the reward function $r$, any of which might be unknown. We can plug in an SL algorithm to **model** these functions, and then solve the modeled environment using dynamic programming (Section 1.3.2). This approach is called **fitted DP** and will be covered in Chapter 5. In the rest of this chapter, we'll formalize the SL task and examine some basic algorithms.

## 4.2 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 joint distribution $p \in \triangle(X \times Y)$ known as the **data generating process**. Note that, by the chain rule of probability, this can be factored as $p(x, y) = p(y \mid x)p(x)$.

**Example 4.2.** For example, in Example 4.1, the marginal distribution over $x$ is assumed to be the distribution of handwritten digits by humans, scanned as $28 \times 28$ grayscale images, and 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" **predictor** $\hat{f} : X \to Y$ that, as its name suggests, takes an input and tries to predict the corresponding output.

### 4.2.1 Loss functions

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

**Example 4.3** (Zero-one loss)**.** In the image classification task Example 4.1, 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,

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

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

**Example 4.4** (Square loss)**.** For a continuous output (i.e. $Y \subseteq \mathbb{R}$), we typically use the **squared difference** as the loss function:

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

The squared 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.)

```
x = np.linspace(-1, 1, 20)
y = x ** 2
plt.plot(x, y)
plt.xlabel(r"$\hat y - y$")
plt.ylabel(r"$\ell(\hat y, y)$")
plt.show()
```

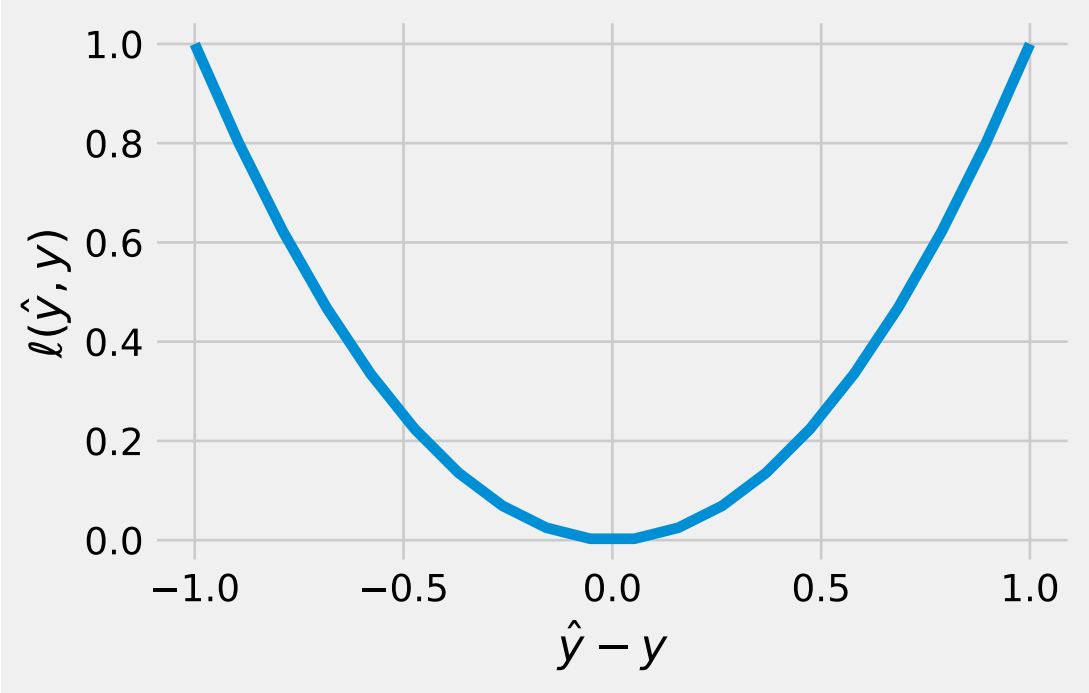



Figure 4.2: Squared loss.

### 4.2.2 Model selection

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

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

Our goal is then to find the function $\hat{f}$ that minimizes the test error. For certain loss functions, this can be analytically computed, such as for squared error.

**Theorem 4.1** (The conditional expectation minimizes mean squared error)**.** *An important result is that, under the squared loss, the optimal predictor is the* ***conditional expectation****:*

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

*Proof.* We can decompose the mean squared error as

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

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 Example 4.2, the joint distribution of $x, y$ is intractable to compute, and so we can't evaluate $\mathbb{E}[y \mid x]$ analytically. Instead, all we have are $N$ samples from the joint distribution of $x$ and $y$. How might we use these to *approximate* the generalization error?

## 4.3 Empirical risk minimization

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

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

By the law of large numbers, as $N$ grows to infinity, the training loss converges to the generalization error.

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

$$\begin{aligned}\hat{f}_{\text{ERM}} &:= \arg\min_{f \in \mathcal{F}} \text{training loss}(f) \\ &= \arg\min_{f \in \mathcal{F}} \frac{1}{N}\sum_{n=1}^{N} \ell(f(x_n), y_n).\end{aligned} \tag{4.1}$$

### 4.3.1 Function classes

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

**Exercise 4.1** (Overfitting)**.** Suppose we are trying to approximate a relationship between real-valued inputs and outputs using squared loss as our loss function. Consider the predictor (visualized in Figure 4.3)

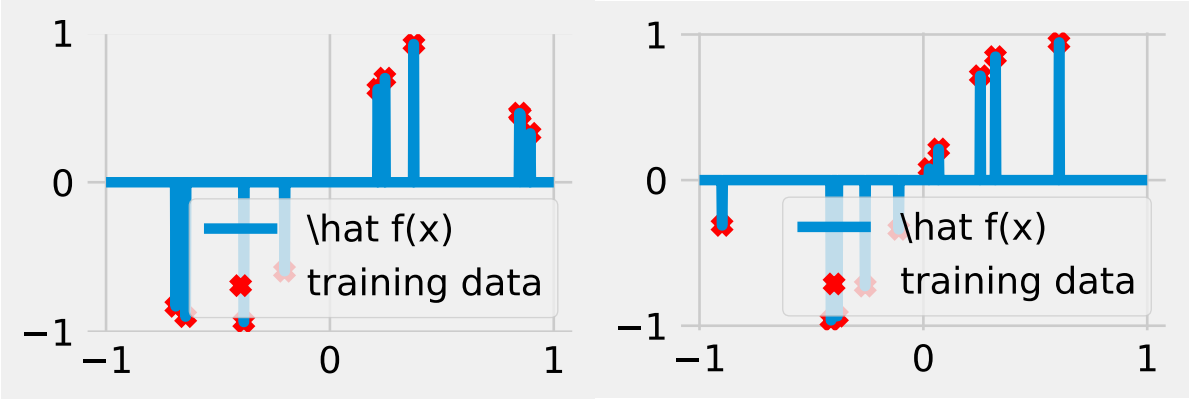
$$\hat{f}(x) = \sum_{n=1}^{N} y_n \mathbf{1}\{x = x_n\}.$$

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

```
n = 1000
x_axis = np.linspace(-1, +1, n)

for _ in range(2):
    x_train = np.random.uniform(-1, +1, 10)
    y_train = np.sin(np.pi * x_train)
    y_hat = np.where(np.isclose(x_axis[:, None], x_train, atol=2/n), y_train, 0).sum(axis=-1)

    plt.plot(x_axis, y_hat, label=r'\hat f(x)')
    plt.scatter(x_train, y_train, color='red', marker='x', label='training data')
    plt.legend()
    plt.gcf().set_size_inches(3, 2)
    plt.show()
```

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**). The mathematical details of the so-called **bias-variance tradeoff** can be found, for example, in Hastie, Tibshirani, and Friedman (2013, chap. 2.9).

```
n_samples = 10
x_axis = np.linspace(-1, +1, 50)
```



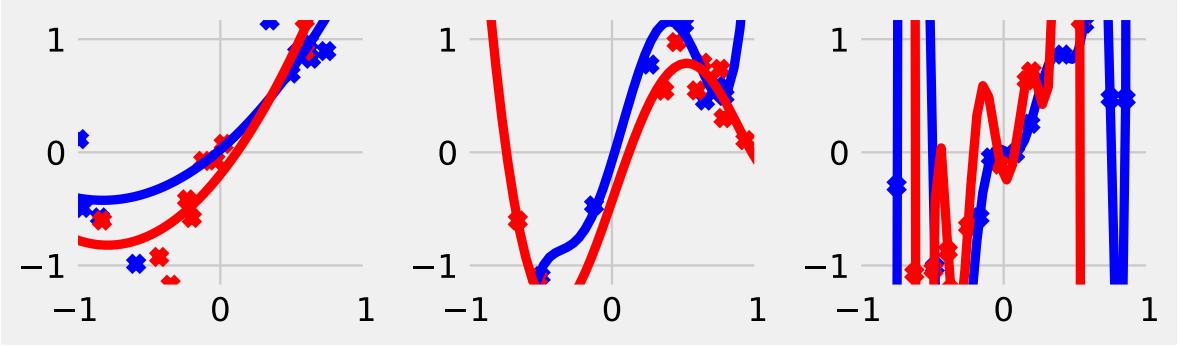

(a) One training dataset.

(b) Another training dataset.

Figure 4.3: A pathological predictor.

```
def generate_data(sigma=0.2):
    x_train = np.random.uniform(-1, +1, n_samples)
    y_train = np.sin(np.pi * x_train) + sigma * np.random.normal(size=n_samples)
    return x_train, y_train


def transform(x: Float[Array, " N"], d: int):
    return np.column_stack([
        x ** d_
        for d_ in range(d + 1)
    ])


for d in [2, 5, 50]:
    for _ in range(2):
        x_train, y_train = generate_data()

        x_features = transform(x_train, d)
        w = np.linalg.lstsq(x_features, y_train)[0]
        y_hat = transform(x_axis, d) @ w

        color = 'blue' if _ == 0 else 'red'
        plt.scatter(x_train, y_train, color=color, marker='x')
        plt.plot(x_axis, y_hat, color=color)
    plt.xlim(-1, +1)
    plt.ylim(-1.2, 1.2)
```

```
    plt.gcf().set_size_inches(2, 2)
    plt.show()
```




(a) Degree 2 polynomials (b) Degree 5 polynomials (c) Degree 50 polynomials

Figure 4.4: 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 make certain parallelizable operations more efficient.

### 4.3.2 Parameterized function classes

Both of the function classes we will consider, linear maps and neural networks, are **finite-dimensional**, a.k.a. **parameterized**. This means each function can be identified using some finite set of **parameters**, which we denote $\theta \in \mathbb{R}^D$.

**Example 4.5** (Quadratic functions)**.** As a third example of a parameterized function class, consider the class of **quadratic functions**, i.e. polynomials of degree 2. This is a three-dimensional function space, since we can describe any quadratic $p$ as

$$p(x) = ax^2 + bx + c,$$

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

$$p(x) = a'(x - b')^2 + c'.$$

Note that the choice of parameterization can impact the performance of the chosen fitting method. What is the derivative of the first expression with respect to $a, b, c$? Compare this to the derivative of the second expression with respect to $a', b', c'$. This shows that gradient-based fitting methods may change their behavior depending on the parameterization.

Using a parameterized function class allows us to reframe the ERM problem Equation 4.1 in terms of optimizing over the *parameters* instead of over the functions they represent:

$$\begin{aligned} \hat{\theta}_{\text{ERM}} &:= \arg\min_{\theta \in \mathbb{R}^D} \text{training loss}(f_\theta) \\ &= \frac{1}{N} \sum_{n=1}^{N} (y_n - f_\theta(x_n))^2 \end{aligned} \tag{4.2}$$

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

### 4.3.3 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 $\theta^t$. 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.

```
Params = Float[Array, " D"]


def gradient_descent(
    loss: Callable[[Params], float],
     _init: Params,
     : float,
    epochs: int,
):
    """
```

```
    Run gradient descent to minimize the given loss function
    (expressed in terms of the parameters).
    """
     = _init
    for _ in range(epochs):
         =   -   * grad(loss)( )
    return
```

In Section 6.2.1, 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 $\eta$? 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 (or rather, gradient ascent) 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 6, we will do exactly this.

## 4.4 Examples of parameterized function classes

### 4.4.1 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

$$\begin{aligned} \hat{\theta} &= \arg\min_{\theta \in \mathbb{R}^D} \frac{1}{2} \sum_{n=1}^{N} (y_n - \theta^\top x_n)^2 \\ &= \arg\min_{\theta \in \mathbb{R}^D} \frac{1}{2} \|Y - X\theta\|^2 \\ &= (X^\top X)^{-1} X^\top Y, \end{aligned} \tag{4.3}$$

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 4.2.** Gradient descent on the ERM problem (Equation 4.3), 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 $\phi$, 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.

### 4.4.2 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 $\sigma$):

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

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 $\sigma$ 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.

## 4.5 Summary

We have now gotten a glimpse into **supervised learning**, which seeks to learn about some input-output relationship using a dataset of example points. In particular, we typically seek to compute a **predictor** that takes in an input value and returns a good guess for the corresponding output. We score predictors using a **loss function** that measures how incorrectly it guesses. We want to find a predictor 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.

## 4.6 References

James et al. (2023) provides an accessible introduction to supervised learning. Hastie, Tibshirani, and Friedman (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.

# 5 Fitted Dynamic Programming Algorithms

---

---

## 5.1 Introduction

We borrow these definitions from the Chapter 1 chapter:

```
from utils import gym, tqdm, rand, Float, Array, NamedTuple, Callable, Optional, np

key = rand.PRNGKey(184)

class Transition(NamedTuple):
    s: int
    a: int
    r: float


Trajectory = list[Transition]


def get_num_actions(trajectories: list[Trajectory]) -> int:
    """Get the number of actions in the dataset. Assumes actions range from 0 to A-1."""
    return max(max(t.a for t in  ) for   in trajectories) + 1


State = Float[Array, "..."]  # arbitrary shape

# assume finite `A` actions and f outputs an array of Q-values
# i.e. Q(s, a, h) is implemented as f(s, h)[a]
QFunction = Callable[[State, int], Float[Array, " A"]]
```

```
def Q_zero(A: int) -> QFunction:
    """A Q-function that always returns zero."""
    return lambda s, a: np.zeros(A)


# a deterministic time-dependent policy
Policy = Callable[[State, int], int]


def q_to_greedy(Q: QFunction) -> Policy:
    """Get the greedy policy for the given state-action value function."""
    return lambda s, h: np.argmax(Q(s, h))
```

The Chapter 1 chapter discussed the case of **finite** MDPs, where the state and action spaces $\mathcal{S}$ and $\mathcal{A}$ were finite. This gave us a closed-form expression for computing the r.h.s. of Theorem 1.1. In this chapter, we consider the case of **large** or **continuous** state spaces, where the state space is too large to be enumerated. In this case, we need to *approximate* the value function and Q-function using methods from **supervised learning**.

## 5.2 Fitted value iteration

Let us apply ERM to the RL problem of computing the optimal policy / value function.

How did we compute the optimal value function in MDPs with *finite* state and action spaces?

- In a Section 1.3, we can use Definition 1.10, working backwards from the end of the time horizon, to compute the optimal value function exactly.
- In an Section 1.4, we can use Section 1.5.3.1, which iterates the Bellman optimality operator Equation 1.7 to approximately compute the optimal value function.

Our existing approaches represent the value function, and the MDP itself, in matrix notation. But what happens if the state space is extremely large, or even infinite (e.g. real-valued)? Then computing a weighted sum over all possible next states, which is required to compute the Bellman operator, becomes intractable.

Instead, we will need to use *function approximation* methods from supervised learning to solve for the value function in an alternative way.

In particular, suppose we have a dataset of $N$ trajectories $\tau_1, \dots, \tau_N \sim \rho_\pi$ from some policy $\pi$ (called the **data collection policy**) acting in the MDP of interest. Let us indicate the trajectory index in the superscript, so that

$$\tau_i = \{s_0^i, a_0^i, r_0^i, s_1^i, a_1^i, r_1^i, \ldots, s_{H-1}^i, a_{H-1}^i, r_{H-1}^i\}.$$

```
def collect_data(
    env: gym.Env, N: int, H: int, key: rand.PRNGKey,  : Optional[Policy] = None
) -> list[Trajectory]:
    """Collect a dataset of trajectories from the given policy (or a random one)."""
    trajectories = []
    seeds = [rand.bits(k).item() for k in rand.split(key, N)]
    for i in tqdm(range(N)):
          = []
        s, _ = env.reset(seed=seeds[i])
        for h in range(H):
            # sample from a random policy
            a =  (s, h) if   else env.action_space.sample()
            s_next, r, terminated, truncated, _ = env.step(a)
             .append(Transition(s, a, r))
            if terminated or truncated:
                break
            s = s_next
        trajectories.append( )
    return trajectories
```

```
env = gym.make("LunarLander-v3")
trajectories = collect_data(env, 100, 300, key)
trajectories[0][:5]  # show first five transitions from first trajectory
```

```
  0%|          | 0/100 [00:00<?, ?it/s] 59%|       | 59/100 [00:00<00:00, 584.45it/s]100%|

[Transition(s=array([-0.00767412,  1.4020356 , -0.77731264, -0.3948966 ,  0.00889908,
         0.17607284,  0.        ,  0.        ], dtype=float32), a=np.int64(3), r=np.float64(0
 Transition(s=array([-0.01526899,  1.392572  , -0.76625407, -0.42065707,  0.01559265,
         0.133885  ,  0.        ,  0.        ], dtype=float32), a=np.int64(2), r=np.float64(0
 Transition(s=array([-0.02275753,  1.3831123 , -0.75616544, -0.4205166 ,  0.02282397,
         0.14463985,  0.        ,  0.        ], dtype=float32), a=np.int64(1), r=np.float64(-
 Transition(s=array([-0.0303196 ,  1.3730422 , -0.7653763 , -0.44773343,  0.03190062,
         0.18154958,  0.        ,  0.        ], dtype=float32), a=np.int64(2), r=np.float64(1
 Transition(s=array([-0.03785544,  1.3635046 , -0.76299477, -0.42412326,  0.04121665,
         0.18633768,  0.        ,  0.        ], dtype=float32), a=np.int64(3), r=np.float64(-
```

Can we view the dataset of trajectories as a "labelled dataset" in order to apply supervised learning to approximate the optimal Q-function? Yes! Recall that we can characterize the optimal Q-function using the Theorem 1.4, which don't depend on an actual policy:

$$Q_h^\star(s,a) = r(s,a) + \mathbb{E}_{s' \sim P(s,a)}[\max_{a'} Q_{h+1}^\star(s',a')]$$

We can think of the arguments to the Q-function – i.e. the current state, action, and timestep $h$ – as the inputs $x$, and the r.h.s. of the above equation as the label $f(x)$. Note that the r.h.s. can also be expressed as a **conditional expectation**:

$$f(x) = \mathbb{E}[y \mid x] \quad \text{where} \quad y = r(s_h, a_h) + \max_{a'} Q_{h+1}^\star(s',a').$$

Approximating the conditional expectation is precisely the task that Section 4.3 is suited for!

Our above dataset would give us $N \cdot H$ samples in the dataset:

$$x_{ih} = (s_h^i, a_h^i, h) \qquad y_{ih} = r(s_h^i, a_h^i) + \max_{a'} Q_{h+1}^\star(s_{h+1}^i, a')$$

```
def get_X(trajectories: list[Trajectory]):
    """
    We pass the state and timestep as input to the Q-function
    and return an array of Q-values.
    """
    rows = [( [h].s,  [h].a, h) for   in trajectories for h in range(len( ))]
    return [np.stack(ary) for ary in zip(*rows)]


def get_y(
    trajectories: list[Trajectory],
    f: Optional[QFunction] = None,
     : Optional[Policy] = None,
):
    """
    Transform the dataset of trajectories into a dataset for supervised learning.
    If ` ` is None, instead estimates the optimal Q function.
    Otherwise, estimates the Q function of  .
    """
    f = f or Q_zero(get_num_actions(trajectories))
    y = []
    for   in trajectories:
        for h in range(len( ) - 1):
```

```
            s, a, r =  [h]
            Q_values = f(s, h + 1)
            y.append(r + (Q_values[ (s, h + 1)] if   else Q_values.max()))
        y.append( [-1].r)
    return np.array(y)
```

```
s, a, h = get_X(trajectories[:1])
print("states:", s[:5])
print("actions:", a[:5])
print("timesteps:", h[:5])
```

```
states: [[-0.00767412  1.4020356  -0.77731264 -0.3948966   0.00889908  0.17607284
   0.          0.        ]
 [-0.01526899  1.392572   -0.76625407 -0.42065707  0.01559265  0.133885
   0.          0.        ]
 [-0.02275753  1.3831123  -0.75616544 -0.4205166   0.02282397  0.14463985
   0.          0.        ]
 [-0.0303196   1.3730422  -0.7653763  -0.44773343  0.03190062  0.18154958
   0.          0.        ]
 [-0.03785544  1.3635046  -0.76299477 -0.42412326  0.04121665  0.18633768
   0.          0.        ]]
actions: [3 2 1 2 3]
timesteps: [0 1 2 3 4]
```

```
get_y(trajectories[:1])[:5]
```

```
array([ 0.01510256,  0.80230962, -2.09420313,  1.07975308, -0.18929023])
```

Then we can use empirical risk minimization to find a function $\hat{f}$ that approximates the optimal Q-function.

```
# We will see some examples of fitting methods in the next section
FittingMethod = Callable[[Float[Array, "N D"], Float[Array, " N"]], QFunction]
```

But notice that the definition of $y_{ih}$ depends on the Q-function itself! How can we resolve this circular dependency? Recall that we faced the same issue when evaluating a policy in an infinite-horizon MDP (Section 1.5.2.2). There, we iterated the Definition 1.8 since we knew that the policy's value function was a fixed point of the policy's Bellman operator. We can apply the same strategy here, using the $\hat{f}$ from the previous iteration to compute the labels $y_{ih}$, and then using this new dataset to fit the next iterate.

**Definition 5.1** (Fitted Q-function iteration)**.**

1. Initialize some function $\hat{f}(s, a, h) \in \mathbb{R}$.
2. Iterate the following:
    1. Generate a supervised learning dataset $X, y$ from the trajectories and the current estimate $f$, where the labels come from the r.h.s. of the Bellman optimality operator Equation 1.7
    2. Set $\hat{f}$ to the function that minimizes the empirical risk:

$$\hat{f} \leftarrow \arg\min_f \frac{1}{N} \sum_{i=1}^{N} (y_i - f(x_i))^2.$$

```python
def fitted_q_iteration(
    trajectories: list[Trajectory],
    fit: FittingMethod,
    epochs: int,
    Q_init: Optional[QFunction] = None,
) -> QFunction:
    """
    Run fitted Q-function iteration using the given dataset.
    Returns an estimate of the optimal Q-function.
    """
    Q_hat = Q_init or Q_zero(get_num_actions(trajectories))
    X = get_X(trajectories)
    for _ in range(epochs):
        y = get_y(trajectories, Q_hat)
        Q_hat = fit(X, y)
    return Q_hat
```

## 5.3 Fitted policy evaluation

We can also use this fixed-point interation to *evaluate* a policy using the dataset (not necessarily the one used to generate the trajectories):

**Definition 5.2** (Fitted policy evaluation)**. Input:** Policy $\pi : \mathcal{S} \times [H] \to \Delta(\mathcal{A})$ to be evaluated.

**Output:** An approximation of the value function $Q^\pi$ of the policy.

1. Initialize some function $\hat{f}(s, a, h) \in \mathbb{R}$.
2. Iterate the following:

1. Generate a supervised learning dataset $X, y$ from the trajectories and the current estimate $f$, where the labels come from the r.h.s. of the Theorem 1.1 for the given policy.
2. Set $\hat{f}$ to the function that minimizes the empirical risk:

$$\hat{f} \leftarrow \arg\min_f \frac{1}{N} \sum_{i=1}^{N} (y_i - f(x_i))^2.$$

```
def fitted_evaluation(
    trajectories: list[Trajectory],
    fit: FittingMethod,
     : Policy,
    epochs: int,
    Q_init: Optional[QFunction] = None,
) -> QFunction:
    """
    Run fitted policy evaluation using the given dataset.
    Returns an estimate of the Q-function of the given policy.
    """
    Q_hat = Q_init or Q_zero(get_num_actions(trajectories))
    X = get_X(trajectories)
    for _ in tqdm(range(epochs)):
        y = get_y(trajectories, Q_hat,  )
        Q_hat = fit(X, y)
    return Q_hat
```

Spot the difference between `fitted_evaluation` and `fitted_q_iteration`. (See the definition of `get_y`.) How would you modify this algorithm to evaluate the data collection policy?

## 5.4 Fitted policy iteration

We can use this policy evaluation algorithm to adapt Section 1.5.3.2 to this new setting. The algorithm remains exactly the same – repeatedly make the policy greedy w.r.t. its own value function – except now we must evaluate the policy (i.e. compute its value function) using the iterative `fitted-evaluation` algorithm.

```
def fitted_policy_iteration(
    trajectories: list[Trajectory],
    fit: FittingMethod,
    epochs: int,
```

```
    evaluation_epochs: int,
     _init: Optional[Policy] = lambda s, h: 0,  # constant zero policy
):
    """Run fitted policy iteration using the given dataset."""
      = _init
    for _ in range(epochs):
        Q_hat = fitted_evaluation(trajectories, fit,  , evaluation_epochs)
         = q_to_greedy(Q_hat)
    return
```

## 5.5 Summary

# 6 Policy Gradient Methods

## 6.1 Introduction

The core task of RL is finding the **optimal policy** in a given environment. This is essentially an *optimization problem:* out of some space of policies, we want to find the one that achieves the maximum total reward (in expectation).

It's typically intractable to compute the optimal policy exactly in some finite number of steps. Instead, **policy optimization algorithms** start from some randomly initialized policy, and then *improve* it step by step. We've already seen some examples of these, namely Section 1.5.3.2 for finite MDPs and Section 2.6.4 in continuous control.

In particular, we often use policies that can be described by some finite set of **parameters.** We will see some examples in Section 6.3.1. For such parameterized policies, we can approximate the **policy gradient:** the gradient of the expected total reward with respect to the parameters. This tells us the direction the parameters should be updated to achieve a higher expected total reward. Policy gradient methods are responsible for groundbreaking applications including AlphaGo, OpenAI Five, and large language models, many of which use policies parameterized as deep neural networks.

1. We begin the chapter with a short review of gradient ascent, a general **optimization method.**
2. We'll then see how to estimate the **policy gradient,** enabling us to apply (stochastic) gradient ascent in the RL setting.
3. Then we'll explore some *proximal optimization* techniques that ensure the steps taken are "not too large". This is helpful to stabilize training and widely used in practice.

```
%load_ext autoreload
%autoreload 2
```

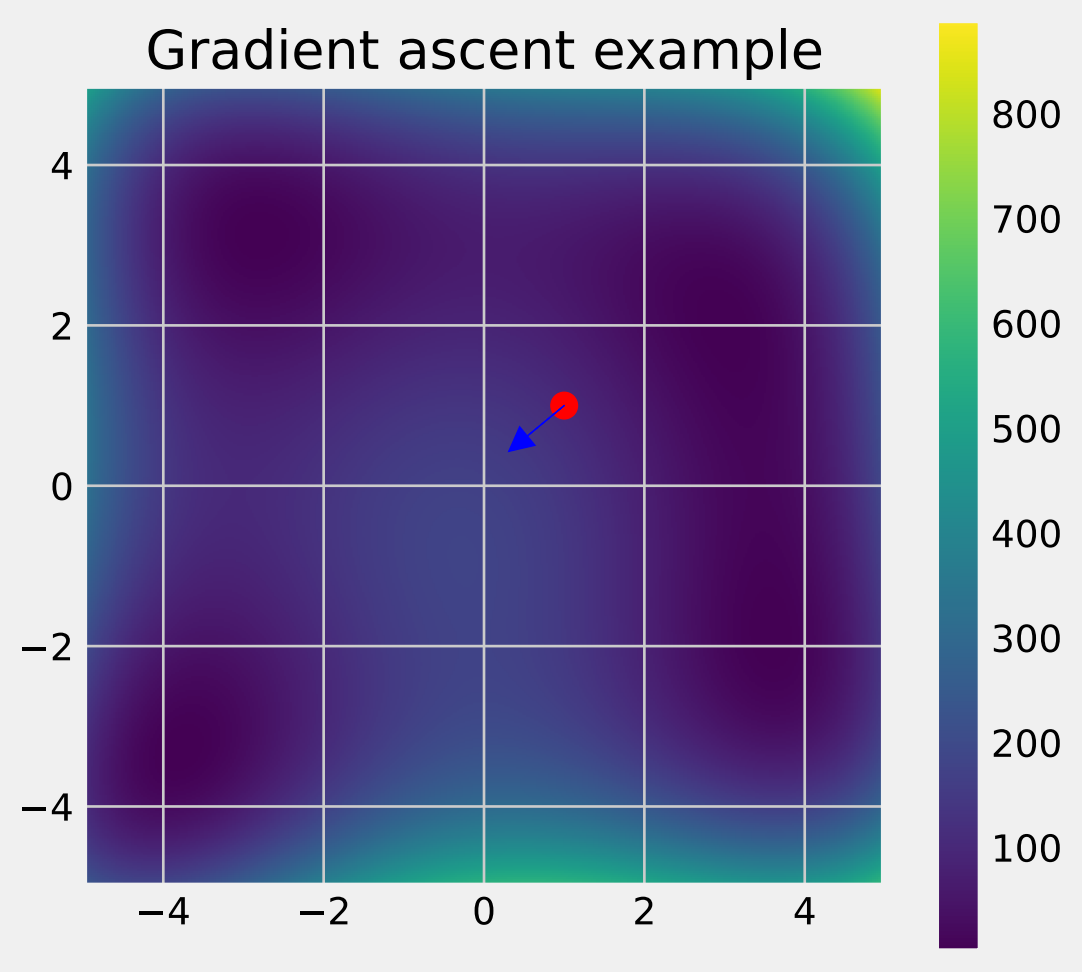
```
from utils import plt, Array, Float, Callable, jax, jnp, latex, gym
```

## 6.2 Gradient Ascent

You may have previously heard of *gradient descent* for minimizing functions. Optimization problems are usually posed as *minimization* problems by convention. However, in RL, we usually talk about *maximizing* the expected total reward, and so we perform gradient *ascent* instead.

**Gradient ascent** is a general optimization algorithm for any differentiable function. A suitable analogy for this algorithm is hiking up a mountain, where you keep taking steps in the steepest direction upwards. Here, your vertical position $y$ is the function being optimized, and your horizontal position $(x, z)$ is the input to the function. The *slope* of the mountain at your current position is given by the *gradient*, written $\nabla y(x, z) \in \mathbb{R}^2$.

```
def f(x, y):
    """Himmelblau's function"""
    return (x**2 + y - 11)**2 + (x + y**2 - 7)**2

# Create a grid of points
x = jnp.linspace(-5, 5, 400)
y = jnp.linspace(-5, 5, 400)
X, Y = jnp.meshgrid(x, y)
Z = f(X, Y)

fig, ax = plt.subplots(figsize=(6, 6))
img = ax.imshow(Z, extent=[-5, 5, -5, 5], origin='lower')
fig.colorbar(img, ax=ax)

tx, ty = 1.0, 1.0
gx, gy = jax.grad(f, argnums=(0, 1))(tx, ty)

ax.scatter(tx, ty, color='red', s=100)
ax.arrow(tx, ty, gx * 0.01, gy * 0.01, head_width=0.3, head_length=0.3, fc='blue', ec='blue')

ax.set_title("Gradient ascent example")
plt.show()
```




For differentiable functions, this can be thought of as the vector of partial derivatives,

$$\nabla y(x, z) = \begin{pmatrix} \frac{\partial y}{\partial x} \\ \frac{\partial y}{\partial z} \end{pmatrix}.$$

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. This means that the direction with the highest slope is exactly the gradient itself, so we can describe the gradient ascent algorithm as follows:

**Definition 6.1** (Gradient ascent)**.**

$$\begin{pmatrix} x^{k+1} \\ z^{k+1} \end{pmatrix} = \begin{pmatrix} x^k \\ z^k \end{pmatrix} + \eta \nabla y(x^k, z^k)$$

where $k$ 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, $\eta^0, \ldots, \eta^K$.)

The case of a two-dimensional input is easy to visualize. 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 $\theta$ to denote the parameters being optimized over. (In the above example, $\theta = \begin{pmatrix} x & z \end{pmatrix}^\top$).

Notice that our parameters will stop changing once $\nabla J(\theta) = 0$. Once we reach this **stationary point,** our current parameters are 'locally optimal' in some sense; it's impossible to increase the function by moving in any direction. If $J$ is *convex*, 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.*

### 6.2.1 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 supports symbolic differentiation. However, functions implemented 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:

$$\nabla_u J(x) = \lim_{\varepsilon \to 0} \frac{J(x + \varepsilon u) - J(x)}{\varepsilon}$$

Then, we can substitute a small value of $\varepsilon$ on the r.h.s. to approximate the directional derivative. How small, though? If we need an accurate estimate, we may need such a small value of $\varepsilon$ 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.

### 6.2.2 Stochastic gradient ascent

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. However, if our dataset is very large, it might not fit into our computer's memory! Typically in these cases, we compute some *estimate* of the gradient at each step, and walk in that direction instead. This is called **stochastic** gradient ascent. In the SL example above, we might randomly choose a *minibatch* of samples and use them to estimate the true prediction error. (This approach is known as ***minibatch* SGD**.)

```
def sgd(
    theta_init: Float[Array, " D"],
    estimate_gradient: Callable[[Float[Array, " D"]], Float[Array, " D"]],
    eta: float,
    n_steps: int,
):
    # Perform `n_steps` steps of SGD.

    # `estimate_gradient` eats the current parameters and returns an estimate of the objecti
    theta = theta_init
    for step in range(n_steps):
        theta += eta * estimate_gradient(theta)
    return theta

latex(sgd)
```

**function** $\mathrm{sgd}(\theta_{\text{init}} : \mathbb{R}^D, \mathrm{estimate\_gradient} : \mathrm{Callable}_{([\mathbb{R}^D],\mathbb{R}^D)}, \eta : \mathrm{float}, n_{\text{steps}} : \mathrm{int})$
  $\theta \leftarrow \theta_{\text{init}}$
  **for** $\mathrm{step} \in \mathrm{range}(n_{\text{steps}})$ **do**
    $\theta \leftarrow \theta + \eta \cdot \mathrm{estimate\_gradient}(\theta)$
  **end for**
  **return** $\theta$
**end function**

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:

$$\mathbb{E}[\tilde{\nabla} J(\theta)] = \nabla J(\theta).$$

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 $\theta$ that is "close" to a stationary point. In another perspective, for such functions, the local "landscape" of $J$ around $\theta$ becomes flatter and flatter the longer we run SGD.

#### 6.2.2.1 SGD convergence

More formally, suppose we run SGD for $K$ steps, using an unbiased gradient estimator. Let the step size $\eta^k$ scale as $O(1/\sqrt{k})$. Then if $J$ is bounded and $\beta$-smooth (see below), and the *norm* of the gradient estimator has a bounded second moment $\sigma^2$,

$$\|\nabla J(\theta^K)\|^2 \leq O\left(M\beta\sigma^2/K\right).$$

We call a function $\beta$-smooth if its gradient is Lipschitz continuous with constant $\beta$:

$$\|\nabla J(\theta) - \nabla J(\theta')\| \leq \beta\|\theta - \theta'\|.$$

We'll now see a concrete application of gradient ascent in the context of policy optimization.

## 6.3 Policy (stochastic) gradient ascent

Remember that in RL, the primary goal is to find the *optimal policy* that achieves the maximimum total reward, which we can express using Definition 1.6:

$$J(\pi) := \mathbb{E}_{s_0\sim\mu_0} V^\pi(s_0) = \mathbb{E}_{\tau\sim\rho^\pi} \sum_{h=0}^{H-1} r(s_h, a_h) \tag{6.1}$$

where $\rho^\pi$ is the distribution over trajectories induced by $\pi$ (see Definition 1.5).

(Note that we'll continue to work in the *undiscounted, finite-horizon case.* Analogous results hold for the *discounted, infinite-horizon setup.*)

As shown by the notation, this is exactly the function $J$ that we want to maximize using gradient ascent. What variables are we optimizing over in this problem? Well, the objective function $J$ is a function of the policy $\pi$, but in general, $\pi$ is a function, and optimizing over the entire space of arbitrary input-output mappings would be intractable. Instead, we need to describe $\pi$ in terms of some finite set of *parameters* $\theta$.

### 6.3.1 Example policy parameterizations

What are some ways we could parameterize our policy?

**Example 6.1** (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. Then to turn this into a valid distribution, we perform a **softmax** operation: we exponentiate each of them, and then normalize to form a valid distribution.

$$\pi_\theta^{\text{softmax}}(a|s) = \frac{\exp(\theta_{s,a})}{\sum_{s,a'} \exp(\theta_{s,a'})}.$$

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.

**Example 6.2** (Linear in features)**.** Another approach is to map each state-action pair into some **feature space** $\phi(s,a) \in \mathbb{R}^p$. Then, to map a feature vector to a probability, we take a linear combination of the features and take a softmax:

$$\pi_\theta^{\text{linear in features}}(a|s) = \frac{\exp(\theta^\top \phi(s,a))}{\sum_{a'} \exp(\theta^\top \phi(s,a'))}.$$

Another interpretation is that $\theta$ represents the feature vector of the "desired" state-action pair, as state-action pairs whose features align closely with $\theta$ are given higher probability.

**Example 6.3** (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:

$$\pi_\theta^{\text{general}}(a|s) = \frac{\exp(f_\theta(s,a))}{\sum_{a'} \exp(f_\theta(s,a'))}.$$

**Example 6.4** (Diagonal Gaussian policies for continuous action spaces)**.** Consider a continuous $n$-dimensional action space $\mathcal{A} = \mathbb{R}^n$. Then for a stochastic policy, we could use a function to predict the *mean* action and then add some random noise about it. For example, we could use a neural network to predict the mean action $\mu_\theta(s)$ and then add some noise $\epsilon \sim \mathcal{N}(0, \sigma^2 I)$ to it:

$$\pi_\theta(a|s) = \mathcal{N}(\mu_\theta(s), \sigma^2 I).$$

Now that we have seen some examples of parameterized policies, we will write the total reward in terms of the parameters, overloading notation and letting $\rho_\theta := \rho^{\pi_\theta}$:

$$J(\theta) = \mathbb{E}_{\tau \sim \rho_\theta} R(\tau)$$

where $R(\tau) = \sum_{h=0}^{H-1} r(s_h, a_h)$ denotes the total reward in the trajectory.

Now how do we maximize this function (the expected total reward) over the parameters? One simple idea would be to directly apply gradient ascent:

$$\theta^{k+1} = \theta^k + \eta \nabla J(\theta^k).$$

In order to apply this technique, we need to be able to evaluate the gradient $\nabla J(\theta)$. But $J(\theta)$ is very difficult, or even intractable, to compute exactly, since it involves taking an expectation over all possible trajectories $\tau$. Can we rewrite it in a form that's more convenient to implement?

### 6.3.2 Importance Sampling

There is a general trick called **importance sampling** for evaluating difficult expectations. Suppose we want to estimate $\mathbb{E}_{x \sim p}[f(x)]$ where $p$ is hard or expensive to sample from, but easy to evaluate the likelihood $p(x)$ of. Suppose that we *can* easily sample from a different distribution $q$. Since an expectation is just a weighted average, we can sample $x$ from $q$, compute $f(x)$, and then reweight the results: if $x$ is very likely under $p$ but unlikely under $q$, we should boost its weighting, and if it is common under $q$ but uncommon under $p$, we should lower its weighting. The reweighting factor is exactly the **likelihood ratio** between the target distribution $p$ and the sampling distribution $q$:

$$\mathbb{E}_{x \sim p}[f(x)] = \sum_{x \in \mathcal{X}} f(x) p(x) = \sum_{x \in \mathcal{X}} f(x) \frac{p(x)}{q(x)} q(x) = \mathbb{E}_{x \sim q}\left[\frac{p(x)}{q(x)} f(x)\right].$$

Doesn't this seem too good to be true? If there were no drawbacks, we could use this to estimate *any* expectation of any function on any arbitrary distribution! The drawback is that the variance may be very large due to the likelihood ratio term. If there are values of $x$ that are very rare in the sampling distribution $q$, but common under $p$, then the likelihood ratio $p(x)/q(x)$ will cause the variance to blow up.

## 6.4 The REINFORCE policy gradient

Returning to RL, suppose there is some trajectory distribution $\rho(\tau)$ that is **easy to sample from,** such as a database of existing trajectories. We can then rewrite $\nabla J(\theta)$, a.k.a. the *policy gradient*, as follows. All gradients are being taken with respect to $\theta$.

$$\begin{aligned}
\nabla J(\theta) &= \nabla \mathbb{E}_{\tau\sim\rho_\theta}[R(\tau)] \\
&= \nabla \mathbb{E}_{\tau\sim\rho}\left[\frac{\rho_\theta(\tau)}{\rho(\tau)}R(\tau)\right] && \text{likelihood ratio trick} \\
&= \mathbb{E}_{\tau\sim\rho}\left[\frac{\nabla\rho_\theta(\tau)}{\rho(\tau)}R(\tau)\right] && \text{switching gradient and expectation}
\end{aligned}$$

Note that for $\rho = \rho_\theta$, the inside term becomes

$$\nabla J(\theta) = \mathbb{E}_{\tau\sim\rho_\theta}[\nabla \log \rho_\theta(\tau) \cdot R(\tau)].$$

(The order of operations is $\nabla(\log \rho_\theta)(\tau)$.)

Recall that when the state transitions are Markov (i.e. $s_t$ only depends on $s_{t-1}, a_{t-1}$) and the policy is time-homogeneous (i.e. $a_h \sim \pi_\theta(s_h)$), we can write out the *likelihood of a trajectory* under the policy $\pi_\theta$ autoregressively, as in Definition 1.5. Taking the log of the trajectory likelihood turns it into a sum of terms:

$$\log \rho_\theta(\tau) = \log \mu(s_0) + \sum_{h=0}^{H-1} \log \pi_\theta(a_h \mid s_h) + \log P(s_{h+1} \mid s_h, a_h)$$

When we take the gradient with respect to the parameters $\theta$, only the $\pi_\theta(a_h|s_h)$ terms depend on $\theta$. This gives the following expression for the policy gradient, known as the "REINFORCE" policy gradient Williams (1992):

$$\nabla J(\theta) = \mathbb{E}_{\tau\sim\rho_\theta}\left[\sum_{h=0}^{H-1} \nabla_\theta \log \pi_\theta(a_h|s_h) R(\tau)\right] \tag{6.2}$$

This expression allows us to estimate the gradient by sampling a few sample trajectories from $\pi_\theta$, calculating the likelihoods of the chosen actions, and substituting these into the expression inside the brackets of Equation 6.2. Then we can update the parameters $\theta$ in this direction to perform stochastic gradient ascent.

The rest of this chapter investigates ways to *reduce the variance* of this estimator by subtracting off certain correlated quantities.

**Lemma 6.1** (Intuition behind REINFORCE)**.** *Intuitively speaking, we want to update the policy parameters to maximize the probability of taking* optimal actions*. That is, suppose we are in state $s$, and $a^\star$ is an optimal action to take. Then we want to solve $\theta = \arg\max_{\theta'} \pi_{\theta'}(a^\star \mid s)$, which would lead to the gradient ascent expression*

$$\theta \leftarrow \theta + \nabla \pi_\theta(a^\star \mid s).$$

*However, we don't know the optimal action $a^\star$ in practice. So instead, we must try many actions, and* increase *the probability of the "good" ones and* decrease *the probability of the "bad" ones. Suppose $A(s, a)$ is a measure of how good action $a$ is in state $s$. Then we could write*

$$\theta \leftarrow \theta + \sum_a \pi_\theta(a \mid s) A(s, a) \nabla \pi_\theta(a \mid s).$$

*But this has an issue: the size of each step doesn't just depend on how good it is, but also how* often *the policy takes it already. This could lead to a positive feedback loop where likely actions become more and more likely, without respect to the quality of the action. So we divide by the likelihood to cancel out this factor:*

$$\theta \leftarrow \theta + \sum_a \pi_\theta(a \mid s) A(s, a) \frac{\nabla \pi_\theta(a \mid s)}{\pi_\theta(a \mid s)}.$$

*But once we simplify, and sum across timesteps, this becomes* almost *exactly the gradient written above!*

$$\theta \leftarrow \theta + \mathbb{E}_{a \sim \pi_\theta(\cdot \mid s)}[\sum_{h=0}^{H-1} A(s_h, a_h) \nabla \log \pi_\theta(a_h \mid s_h)].$$

*We will see later on what $A$ concretely corresponds to.*

```python
def estimate_gradient_reinforce_pseudocode(env: gym.Env, pi, theta: Float[Array, " D"]):
    """Estimate the policy gradient using REINFORCE."""
    tau = sample_trajectory(env, pi(theta))
    nabla_hat = jnp.zeros_like(theta)
    total_reward = sum(r for _s, _a, r in tau)
    for s, a, r in tau:
        def policy_log_likelihood(theta: Float[Array, " D"]) -> float:
            return log(pi(theta)(s, a))
        nabla_hat += jax.grad(policy_log_likelihood)(theta) * total_reward
    return nabla_hat
```

```
latex(estimate_gradient_reinforce_pseudocode, id_to_latex={"jax.grad": r"\nabla"})
```

**function** estimate_gradient_reinforce_pseudocode(env : gym.Env, $\pi, \theta : \mathbb{R}^D$)
  "Estimate the policy gradient using REINFORCE."
  $\tau \leftarrow$ sample_trajectory(env, $\pi(\theta)$)
  $\widehat{\nabla} \leftarrow$ jnp.zeros_like($\theta$)
  total_reward $\leftarrow \sum_{(\_s,\_a,r)\in\tau}(r)$
  **for** $(s, a, r) \in \tau$ **do**
    **function** policy_log_likelihood($\theta : \mathbb{R}^D$)
      **return** $\log \pi(\theta)\,(s, a)$
    **end function**
    $\widehat{\nabla} \leftarrow \widehat{\nabla} + \nabla(\text{policy\_log\_likelihood})\,(\theta) \cdot \text{total\_reward}$
  **end for**
  **return** $\widehat{\nabla}$
**end function**

For some intuition into how this method works, recall that we update our parameters according to

$$\begin{aligned}\theta_{t+1} &= \theta_t + \eta \nabla J(\theta_t) \\ &= \theta_t + \eta \mathbb{E}_{\tau\sim\rho_{\theta_t}}[\nabla \log \rho_{\theta_t}(\tau) \cdot R(\tau)].\end{aligned}$$

Consider the "good" trajectories where $R(\tau)$ is large. Then $\theta$ gets updated so that these trajectories become more likely. To see why, recall that $\rho_\theta(\tau)$ is the likelihood of the trajectory $\tau$ under the policy $\pi_\theta$, so the gradient points in the direction that makes $\tau$ more likely.

## 6.5 Baselines and advantages

A central idea from supervised 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. The REINFORCE gradient estimator Equation 6.2 is already *unbiased,* meaning that its expectation over trajectories is the true policy gradient. Can we find ways to reduce its *variance* as well?

As a first step, consider that the action taken at step $t$ does not affect the reward from previous timesteps, since they're already in the past. You can also show rigorously that this is the case, and that we only need to consider the present and future rewards to calculate the policy gradient:

$$\nabla J(\theta) = \mathbb{E}_{\tau \sim \rho_\theta} \left[ \sum_{h=0}^{H-1} \nabla_\theta \log \pi_\theta(a_h|s_h) \sum_{h'=h}^{H-1} r(s_{h'}, a_{h'}) \right]$$

Furthermore, by a conditioning argument, we can replace the inner sum over remaining rewards with the policy's Q-function, evaluated at the current state:

$$\nabla J(\theta) = \mathbb{E}_{\tau \sim \rho_\theta} \left[ \sum_{h=0}^{H-1} \nabla_\theta \log \pi_\theta(a_h|s_h) Q^{\pi_\theta}(s_h, a_h) \right] \tag{6.3}$$

**Exercise:** Prove that this is equivalent to the previous definitions. What modification to the expression must be made for the discounted, infinite-horizon setting?

We can further reduce variance by subtracting a **baseline function** $b_h : \mathcal{S} \to \mathbb{R}$ at each timestep $h$. This modifies the policy gradient as follows:

$$\nabla J(\theta) = \mathbb{E}_{\tau \sim \rho_\theta} \left[ \sum_{h=0}^{H-1} \nabla \log \pi_\theta(a_h|s_h) \left( Q^{\pi_\theta}(s_h, a_h) - b_h(s_h) \right) \right]. \tag{6.4}$$

(Again, you should try to prove that this equality still holds.) For example, we might want $b_h$ to estimate the average reward-to-go at a given timestep:

$$b_h^\theta = \mathbb{E}_{\tau \sim \rho_\theta} R_h(\tau).$$

As a better baseline, we could instead choose the *value function.* Note that the random variable $Q_h^\pi(s, a) - V_h^\pi(s)$, where the randomness is taken over the actions, 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 Lemma 6.1: it is *positive* for actions that are better than average (in state $s$), and *negative* for actions that are worse than average. In fact, this quantity has a particular name: the **advantage function.**

**Definition 6.2** (Advantage function)**.**

$$A_h^\pi(s, a) = Q_h^\pi(s, a) - V_h^\pi(s)$$

This measures how much better this action does than the average for that policy. (Note that for an optimal policy $\pi^\star$, the advantage of a given state-action pair is always zero or negative.)

We can now express the policy gradient as follows. Note that the advantage function effectively replaces the $Q$-function from Equation 6.3:

$$\nabla J(\theta) = \mathbb{E}_{\tau \sim \rho_\theta} \left[ \sum_{h=0}^{H-1} \nabla \log \pi_\theta(a_h|s_h) A_h^{\pi_\theta}(s_h, a_h) \right]. \tag{6.5}$$

**Example 6.5** (Policy gradient for the linear-in-features parameterization)**.** The gradient-log-likelihood for the linear-in-features parameterization Example 6.2 is also quite elegant:

$$\begin{aligned}\nabla \log \pi_\theta(a|s) &= \nabla \left( \theta^\top \phi(s,a) - \log \left( \sum_{a'} \exp(\theta^\top \phi(s,a')) \right) \right) \\ &= \phi(s,a) - \mathbb{E}_{a' \sim \pi_\theta(s)} \phi(s,a')\end{aligned}$$

Plugging this into our policy gradient expression, we get

$$\begin{aligned}\nabla J(\theta) &= \mathbb{E}_{\tau \sim \rho_\theta} \left[ \sum_{t=0}^{T-1} \nabla \log \pi_\theta(a_h|s_h) A_h^{\pi_\theta} \right] \\ &= \mathbb{E}_{\tau \sim \rho_\theta} \left[ \sum_{t=0}^{T-1} \left( \phi(s_h,a_h) - \mathbb{E}_{a' \sim \pi(s_h)} \phi(s_h,a') \right) A_h^{\pi_\theta}(s_h,a_h) \right] \\ &= \mathbb{E}_{\tau \sim \rho_\theta} \left[ \sum_{t=0}^{T-1} \phi(s_h,a_h) A_h^{\pi_\theta}(s_h,a_h) \right]\end{aligned}$$

Why can we drop the $\mathbb{E}\phi(s_h,a')$ term? By linearity of expectation, consider the dropped term at a single timestep: $\mathbb{E}_{\tau \sim \rho_\theta} \left[ \left( \mathbb{E}_{a' \sim \pi(s_h)} \phi(s,a') \right) A_h^{\pi_\theta}(s_h,a_h) \right]$. 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.

Note that to avoid correlations between the gradient estimator and the value estimator (i.e. baseline), we must estimate them with independently sampled trajectories:

```
def pg_with_learned_baseline(env: gym.Env, pi, eta: float, theta_init, K: int, N: int) -> Flo
    theta = theta_init
    for k in range(K):
        trajectories = sample_trajectories(env, pi(theta), N)
        V_hat = fit_value(trajectories)
        tau = sample_trajectories(env, pi(theta), 1)
        nabla_hat = jnp.zeros_like(theta)  # gradient estimator

        for h, (s, a) in enumerate(tau):
            def log_likelihood(theta_opt):
                return jnp.log(pi(theta_opt)(s, a))
            nabla_hat = nabla_hat + jax.grad(log_likelihood)(theta) * (return_to_go(tau, h)

        theta = theta + eta * nabla_hat
    return theta
```

```
latex(pg_with_learned_baseline)
```

**function** pg_with_learned_baseline(env : gym.Env, $\pi, \eta$ : float, $\theta_{\text{init}}, K$ : int, $N$ : int)
  $\theta \leftarrow \theta_{\text{init}}$
  **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($\theta$)
    **for** $(h, (s, a)) \in \text{enumerate}(\tau)$ **do**
      **function** log_likelihood($\theta_{\text{opt}}$)
        **return** $\log \pi\left(\theta_{\text{opt}}\right)(s, a)$
      **end function**
      $\widehat{\nabla} \leftarrow \widehat{\nabla} + \text{jax.grad(log\_likelihood)}(\theta) \cdot \left(\text{return\_to\_go}(\tau, h) - \widehat{V}(s)\right)$
    **end for**
    $\theta \leftarrow \theta + \eta \widehat{\nabla}$
  **end for**
  **return** $\theta$
**end function**

Note that you could also generalize this by allowing the learning rate $\eta$ to vary across steps, or take multiple trajectories $\tau$ and compute the sample average 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.

## 6.6 Comparing policy gradient algorithms to policy iteration

What advantages does the policy gradient algorithm have over the policy iteration algorithms covered in Section 1.5.3.2?

*Remark* 6.1 (Policy iteration review). Recall that policy iteration is an algorithm for MDPs with unknown state transitions where we alternate between these 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 6.1** (Performance difference lemma)**.** *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:*

$$V_0^{Alice}(s) - V_0^{Bob}(s) = \mathbb{E}_{\tau \sim \rho_{Alice,s}} \left[ \sum_{h=0}^{H-1} A_h^{Bob}(s_h, a_h) \right] \tag{6.6}$$

*where $\rho_{Alice,s}$ denotes the distribution over trajectories starting in state $s$ when Alice is playing.*

*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!*

*Formally, this corresponds to a nice telescoping simplification when we expand out the definition of the advantage function. Note that*

$$\begin{aligned} A_h^\pi(s_h, a_h) &= Q_h^\pi(s_h, a_h) - V_h^\pi(s_h) \\ &= r_h(s_h, a_h) + \mathbb{E}_{s_{h+1} \sim P(s_h, a_h)}[V_{h+1}^\pi(s_{h+1})] - V_h^\pi(s_h) \end{aligned}$$

*so expanding out the r.h.s. expression of Equation 6.6 and grouping terms together gives*

$$\begin{aligned} \mathbb{E}_{\tau \sim \rho_{Alice,s}} \left[ \sum_{h=0}^{H-1} A_h^{Bob}(s_h, a_h) \right] &= \mathbb{E}_{\tau \sim \rho_{Alice,s}} \left[ \left( \sum_{h=0}^{H-1} r_h(s_h, a_h) \right) + (V_1^{Bob}(s_1) + \cdots + V_H^{Bob}(s_H)) - (V^{Bob_0}(s_0) + \cdots + \right. \\ &= V_0^{Alice}(s) - V_0^{Bob}(s) \end{aligned}$$

*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.)*

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 $\pi$ gets updated to $\tilde{\pi}$. We'll assume these policies are deterministic. Suppose the new policy $\tilde{\pi}$ chooses some action with a negative advantage with respect to $\pi$. That is, when acting according to $\pi$, taking the action from $\tilde{\pi}$ would perform worse than expected. Define $\Delta_\infty$ to be the most negative advantage, that is, $\Delta_\infty = \min_{s \in \mathcal{S}} A_h^\pi(s, \tilde{\pi}(s))$. Plugging this into the Theorem 6.1 gives

$$
\begin{aligned}
V_0^{\tilde{\pi}}(s) - V_0^{\pi}(s) &= \mathbb{E}_{\tau \sim \rho_{\tilde{\pi},s}} \left[ \sum_{h=0}^{H-1} A_h^{\pi}(s_h, a_h) \right] \\
&\geq H\Delta_{\infty} \\
V_0^{\tilde{\pi}}(s) &\geq V_0^{\pi}(s) - H|\Delta_{\infty}|.
\end{aligned}
$$

That is, for some state $s$, the lower bound on the performance of $\tilde{\pi}$ is *lower* than the performance of $\pi$. This doesn't state that $\tilde{\pi}$ *will* necessarily perform worse than $\pi$, 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 $\rho_{\pi}$ and $\rho_{\tilde{\pi}}$ be close to each other. In other words, the "training distribution" that our prediction rule is fitted on, $\rho_{\pi}$, may differ significantly from the "evaluation distribution" $\rho_{\tilde{\pi}}$.

On the other hand, policy gradient methods *do*, albeit implicitly, encourage $\rho_{\pi}$ 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 next three methods: - **trust region policy optimization** (TRPO), which explicitly constrains the difference between the distributions before and after each step; - the **natural policy gradient** (NPG), a first-order approximation of TRPO; - **proximal policy optimization** (PPO), a "soft relaxation" of TRPO.

## 6.7 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. Can we design an algorithm that *explicitly* constrains the "step size"? That is, we want to *improve* the policy as much as possible, measured in terms of the r.h.s. of the Theorem 6.1, while ensuring that its trajectory distribution does not change too much:

$$
\begin{aligned}
\theta^{k+1} \leftarrow \arg\max_{\theta^{\text{opt}}} & \mathbb{E}_{s_0,\ldots,s_{H-1} \sim \pi^k} \left[ \sum_{h=0}^{H-1} \mathbb{E}_{a_h \sim \pi^{\theta^{\text{opt}}}(s_h)} A^{\pi^k}(s_h, a_h) \right] \\
& \text{where distance}(\rho_{\theta^{\text{opt}}}, \rho_{\theta^k}) < \delta
\end{aligned}
$$

Note that we have made a small change to the r.h.s. expression: we use the *states* sampled from the old policy, and only use the *actions* from the new policy. It would be computationally infeasible to sample entire trajectories from $\pi_{\theta}$ as we are optimizing over $\theta$. On the other hand, if $\pi_{\theta}$ returns a vector representing a probability distribution over actions, then evaluating the

expected advantage with respect to this distribution only requires taking a dot product. This approximation also matches the r.h.s. of the PDL to first order in $\theta$. (We will elaborate more on this later.)

How do we describe the distance between $\rho_{\theta^{\text{opt}}}$ and $\rho_{\theta^k}$? We'll use the **Kullback-Leibler divergence (KLD)**:

**Definition 6.3** (Kullback-Leibler divergence)**.** For two PDFs $p, q$,

$$\mathrm{KL}\left(p \parallel q\right) := \mathbb{E}_{x\sim p}\left[\log \frac{p(x)}{q(x)}\right]$$

This 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 my average "surprise" if I *think* data is being generated by $q$ but it's actually generated by $p$. (The **surprise** of an event with probability $p$ is $-\log_2 p$.) Note that $\mathrm{KL}\left(p \parallel q\right) = 0$ if and only if $p = q$. Also note that it is generally *not* symmetric.

Both the objective function and the KLD constraint involve a weighted average over the space of all trajectories. This is intractable in general, so we need to estimate the expectation. As before, we can do this by taking an empirical average over samples from the trajectory distribution. This gives us the following pseudocode:

```
def kl_div_trajectories(pi, theta_1, theta_2, trajectories):
    # Assume trajectories are sampled from pi(theta_1)
    kl_div = 0
    for tau in trajectories:
        for s, a, _r in tau:
            kl_div += jnp.log(pi(theta_1)(s, a)) - jnp.log(pi(theta_2)(s, a))
    return kl_div / len(trajectories)

latex(kl_div_trajectories)
```

**function** kl_div_trajectories($\pi, \theta_1, \theta_2$, trajectories)
  kl_div $\leftarrow 0$
  **for** $\tau \in$ trajectories **do**
    **for** $(s, a, \_\mathrm{r}) \in \tau$ **do**
      kl_div $\leftarrow$ kl_div $+ \log \pi(\theta_1)\left(s, a\right) - \log \pi(\theta_2)\left(s, a\right)$
    **end for**
  **end for**
  **return** $\frac{\text{kl\_div}}{\text{len(trajectories)}}$
**end function**

```
def trpo(env,  , theta_init, n_interactions):
    theta = theta_init
    for k in range(K):
        trajectories = sample_trajectories(env, pi(theta), n_interactions)
        A_hat = fit_advantage(trajectories)

        def approximate_gain(theta_opt):
            A_total = 0
            for tau in trajectories:
                for s, _a, _r in tau:
                    for a in env.action_space:
                        A_total += pi(theta)(s, a) * A_hat(s, a)
            return A_total

        def constraint(theta_opt):
            return kl_div_trajectories(pi, theta, theta_opt, trajectories) <=

        theta = optimize(approximate_gain, constraint)

    return theta

latex(trpo)
```

```
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
```

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 might be an intractable optimization. Do we need to solve it exactly, 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.

## 6.8 Natural policy gradient

We take a *linear* (first-order) approximation to the objective function and a *quadratic* (second-order) approximation to the KL divergence constraint about the current estimate $\theta^k$. This results in the optimization problem

$$\begin{aligned} &\max_{\theta} \nabla_\theta J(\pi_{\theta^k})^\top (\theta - \theta^k) \\ &\text{where } \frac{1}{2}(\theta - \theta^k)^\top F_{\theta^k}(\theta - \theta^k) \leq \delta \end{aligned} \tag{6.7}$$

where $F_{\theta^k}$ is the **Fisher information matrix** defined below.

**Definition 6.4** (Fisher information matrix)**.** Let $p_\theta$ denote a parameterized distribution. Its Fisher information matrix $F_\theta$ can be defined equivalently as:

$$\begin{aligned} F_\theta &= \mathbb{E}_{x\sim p_\theta}\left[(\nabla_\theta \log p_\theta(x))(\nabla_\theta \log p_\theta(x))^\top\right] && \text{covariance matrix of the Fisher score} \\ &= \mathbb{E}_{x\sim p_\theta}[-\nabla_\theta^2 \log p_\theta(x)] && \text{average Hessian of the negative log-likelihood} \end{aligned}$$

Recall that the Hessian of a function describes its curvature: for a vector $\delta \in \Theta$, the quantity $\delta^\top F_\theta \delta$ describes how rapidly the negative log-likelihood changes if we move by $\delta$. The Fisher information matrix is precisely the Hessian of the KL divergence (with respect to either one of the parameters).

In particular, when $p_\theta = \rho_\theta$ denotes a trajectory distribution, we can further simplify the expression:

$$F_\theta = \mathbb{E}_{\tau\sim\rho_\theta}\left[\sum_{h=0}^{H-1}(\nabla \log \pi_\theta(a_h \mid s_h))(\nabla \log \pi_\theta(a_h \mid s_h))^\top\right] \tag{6.8}$$

Note that we've used the Markov property to cancel out the cross terms corresponding to two different time steps.

This is a convex optimization problem with a closed-form solution. To see why, it helps to visualize the case where $\theta$ 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. We recommend Boyd and Vandenberghe (2004) for a comprehensive treatment of convex optimization.

More generally, for a higher-dimensional $\theta$, we can compute the global optima by setting the gradient of the Lagrangian to zero:

$$\begin{aligned} \mathcal{L}(\theta, \alpha) &= \nabla J(\pi_{\theta^k})^\top(\theta - \theta^k) - \alpha\left[\frac{1}{2}(\theta - \theta^k)^\top F_{\theta^k}(\theta - \theta^k) - \delta\right] \\ \nabla\mathcal{L}(\theta^{k+1}, \alpha) &:= 0 \\ \implies \nabla J(\pi_{\theta^k}) &= \alpha F_{\theta^k}(\theta^{k+1} - \theta^k) \\ \theta^{k+1} &= \theta^k + \eta F_{\theta^k}^{-1}\nabla J(\pi_{\theta^k}) \\ \text{where } \eta &= \sqrt{\frac{2\delta}{\nabla J(\pi_{\theta^k})^\top F_{\theta^k}^{-1}\nabla J(\pi_{\theta^k})}} \end{aligned}$$

This gives us the closed-form update. Now the only challenge is to estimate the Fisher information matrix, since, as with the KL divergence constraint, it is an expectation over trajectories, and computing it exactly is therefore typically intractable.

**Definition 6.5** (Natural policy gradient)**.** How many trajectory samples do we need to accurately estimate the Fisher information matrix? As a rule of thumb, the sample complexity should scale with the dimension of the parameter space. This makes this approach intractable in the deep learning setting where we might have a very large number of parameters.

As you can see, the NPG is the "basic" policy gradient algorithm we saw above, but with the gradient transformed by the inverse Fisher information matrix. This matrix can be understood as accounting for the **geometry of the parameter space.** The typical gradient descent algorithm implicitly measures distances between parameters using the typical *Euclidean distance*. Here, where the parameters map to a *distribution*, using the natural gradient update is equivalent to optimizing over **distribution space** rather than parameter space, where distance between distributions is measured by the Definition 6.3.

**Example 6.6** (Natural gradient on a simple problem)**.** Let's step away from RL and consider the following optimization problem over Bernoulli distributions $\pi \in \Delta(\{0, 1\})$:

$$J(\pi) = 100 \cdot \pi(1) + 1 \cdot \pi(0)$$

We can think of the space of such distributions as the line between $(0, 1)$ to $(1, 0)$ on the Cartesian plane:

```
x = jnp.linspace(0, 1, 50)
y = 1 - x
plt.plot(x, y)
plt.xlabel(r"$\pi(0)$")
plt.ylabel(r"$\pi(1)$")
plt.title("Space of Bernoulli distributions")
plt.show()
```

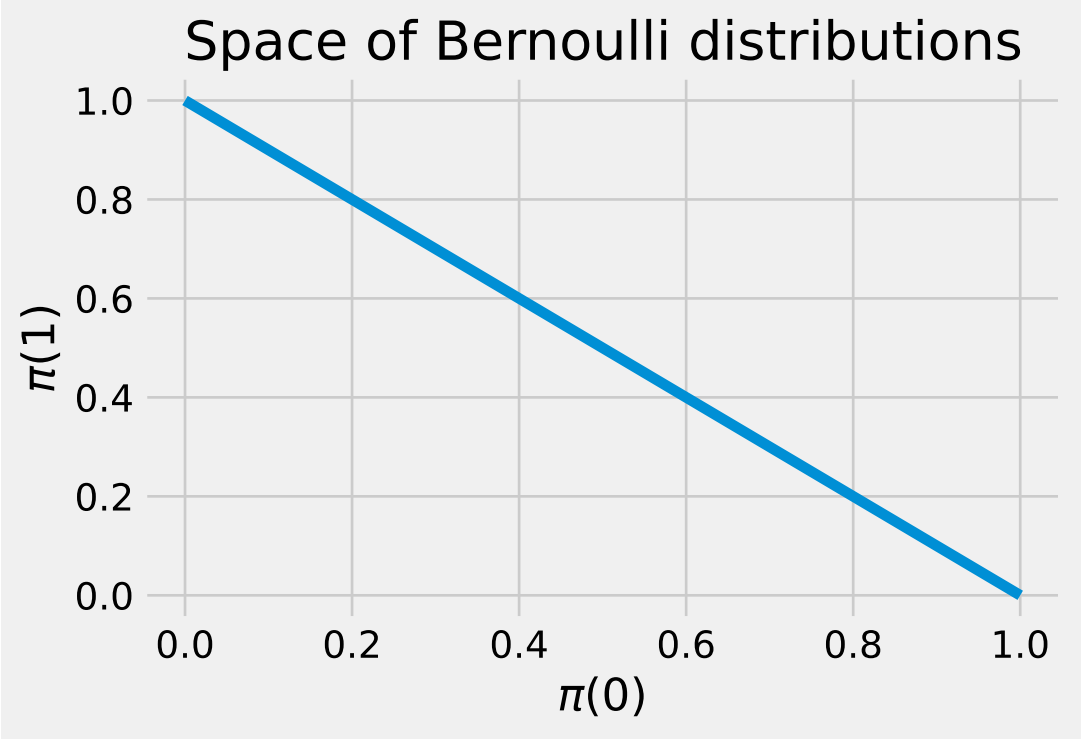



Figure 6.1: A line from (0, 1) to (1, 0)

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 $\theta$ to be unboundedly large. Then the "vanilla" 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.

$$\begin{aligned} F_\theta &= \mathbb{E}_{x\sim\pi_\theta}[(\nabla_\theta \log \pi_\theta(x))^2] \\ &= \frac{\exp(\theta)}{(1+\exp(\theta))^2}. \end{aligned}$$

This gives the natural gradient update

$$\begin{aligned}\theta^{k+1} &= \theta^k + \eta F_{\theta^k}^{-1} \nabla_\theta J(\theta^k) \\ &= \theta^k + 99\eta\end{aligned}$$

which increases at a constant rate, i.e. improves the objective more quickly than "vanilla" gradient ascent.

Though the NPG now gives a closed-form optimization step, it requires computing the inverse Fisher information matrix, which typically scales as $O((\dim \Theta)^3)$. This can be expensive if the parameter space is large. Can we find an algorithm that works in *linear time* with respect to the dimension of the parameter space?

## 6.9 Proximal policy optimization

We can relax the TRPO optimization problem in a different way: Rather than imposing a hard constraint on the KL distance, we can instead impose a *soft* constraint by incorporating it into the objective and penalizing parameter values that drastically change the trajectory distribution.

$$\theta^{k+1} \leftarrow \arg\max_\theta \mathbb{E}_{s_0,\ldots,s_{H-1}\sim\rho_{\pi^k}} \left[\sum_{h=0}^{H-1} \mathbb{E}_{a_h\sim\pi_\theta(s_h)} A^{\pi^k}(s_h, a_h)\right] - \lambda \mathrm{KL}\left(\rho_\theta \parallel \rho_{\theta^k}\right)$$

Here $\lambda$ is a **regularization hyperparameter** that controls the tradeoff between the two terms. This is the objective of the **proximal policy optimization** algorithm (Schulman et al. (2017)).

How do we solve this optimization? Let us begin by simplifying the $\mathrm{KL}\left(\rho_{\pi^k} \parallel \rho_{\pi_\theta}\right)$ term. Expanding gives

$$\begin{aligned}\mathrm{KL}\left(\rho_{\pi^k} \parallel \rho_{\pi_\theta}\right) &= \mathbb{E}_{\tau\sim\rho_{\pi^k}}\left[\log \frac{\rho_{\pi^k}(\tau)}{\rho_{\pi_\theta}(\tau)}\right] \\ &= \mathbb{E}_{\tau\sim\rho_{\pi^k}}\left[\sum_{h=0}^{H-1} \log \frac{\pi^k(a_h \mid s_h)}{\pi_\theta(a_h \mid s_h)}\right] \qquad \text{state transitions cancel} \\ &= \mathbb{E}_{\tau\sim\rho_{\pi^k}}\left[\sum_{h=0}^{H-1} \log \frac{1}{\pi_\theta(a_h \mid s_h)}\right] + c\end{aligned}$$

where $c$ is some constant with respect to $\theta$, and can be ignored. This gives the objective

$$\ell^k(\theta) = \mathbb{E}_{s_0,\dots,s_{H-1}\sim\rho_{\pi^k}}\left[\sum_{h=0}^{H-1}\mathbb{E}_{a_h\sim\pi_\theta(s_h)}A^{\pi^k}(s_h,a_h)\right] - \lambda\mathbb{E}_{\tau\sim\rho_{\pi^k}}\left[\sum_{h=0}^{H-1}\log\frac{1}{\pi_\theta(a_h \mid s_h)}\right]$$

Once again, this takes an expectation over trajectories. But here we cannot directly sample trajectories from $\pi^k$, since in the first term, the actions actually come from $\pi_\theta$. To make this term line up with the other expectation, we would need the actions to also come from $\pi^k$.

This should sound familiar: we want to estimate an expectation over one distribution by sampling from another. We can once again use Section 6.3.2 to rewrite the inner expectation:

$$\mathbb{E}_{a_h\sim\pi_\theta(s_h)}A^{\pi^k}(s_h,a_h) = \mathbb{E}_{a_h\sim\pi^k(s_h)}\frac{\pi_\theta(a_h \mid s_h)}{\pi^k(a_h \mid s_h)}A^{\pi^k}(s_h,a_h)$$

Now we can combine the expectations together to get the objective

$$\ell^k(\theta) = \mathbb{E}_{\tau\sim\rho_{\pi^k}}\left[\sum_{h=0}^{H-1}\left(\frac{\pi_\theta(a_h \mid s_h)}{\pi^k(a_h \mid s_h)}A^{\pi^k}(s_h,a_h) - \lambda\log\frac{1}{\pi_\theta(a_h \mid s_h)}\right)\right]$$

Now we can estimate this function by a sample average over trajectories from $\pi^k$. Remember that to complete a single iteration of PPO, we execute

$$\theta^{k+1} \leftarrow \arg\max_\theta \ell^k(\theta).$$

If $\ell^k$ is differentiable, we can optimize it by gradient ascent, completing a single iteration of PPO.

```
from typing import TypeVar

State = TypeVar("State")
Action = TypeVar("Action")

def ppo(
    env,
    pi: Callable[[Float[Array, " D"]], Callable[[State, Action], float]],
     : float,
    theta_init: Float[Array, " D"],
    n_iters: int,
    n_fit_trajectories: int,
    n_sample_trajectories: int,
):
```

```
    theta = theta_init
    for k in range(n_iters):
        fit_trajectories = sample_trajectories(env, pi(theta), n_fit_trajectories)
        A_hat = fit(fit_trajectories)

        sample_trajectories = sample_trajectories(env, pi(theta), n_sample_trajectories)

        def objective(theta_opt):
            total_objective = 0
            for tau in sample_trajectories:
                for s, a, _r in tau:
                    total_objective += pi(theta_opt)(s, a) / pi(theta)(s, a) * A_hat(s, a) +
            return total_objective / n_sample_trajectories

        theta = optimize(objective, theta)

    return theta

latex(ppo)
```

**function** ppo(env, $\pi$ : Callable$_{([\mathbb{R}^D],\text{Callable}_{([\text{State},\text{Action}],\text{float})})}$, $\lambda$ : float, $\theta_{\text{init}} : \mathbb{R}^D$, $n_{\text{iters}}$ : int, n_fit_trajectories : int,
  $\theta \leftarrow \theta_{\text{init}}$
  **for** $k \in \text{range}(n_{\text{iters}})$ **do**
    fit_trajectories $\leftarrow$ sample_trajectories(env, $\pi(\theta)$, n_fit_trajectories)
    $\widehat{A} \leftarrow$ fit(fit_trajectories)
    sample_trajectories $\leftarrow$ sample_trajectories(env, $\pi(\theta)$, n_sample_trajectories)
    **function** objective($\theta_{\text{opt}}$)
      total_objective $\leftarrow 0$
      **for** $\tau \in$ sample_trajectories **do**
        **for** $(s, a, \_\text{r}) \in \tau$ **do**
          total_objective $\leftarrow$ total_objective $+ \frac{\pi(\theta_{\text{opt}})(s,a)}{\pi(\theta)(s,a)} \widehat{A}(s,a) + \lambda \cdot \log \pi\left(\theta_{\text{opt}}\right)(s,a)$
        **end for**
      **end for**
      **return** $\frac{\text{total\_objective}}{\text{n\_sample\_trajectories}}$
    **end function**
    $\theta \leftarrow \text{optimize}(\text{objective}, \theta)$
  **end for**
  **return** $\theta$
**end function**

## 6.10 Summary

Policy gradient methods are a powerful family of algorithms that directly 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 Equation 6.5.

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 Section 6.7, and then saw ways to relax the constraint in Definition 6.5 and Section 6.9.

These are still popular methods to this day, especially because they efficiently integrate with *deep neural networks* for representing complex functions.

# 7 Imitation Learning

## 7.1 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 7.1: a robot imitating the pose of a young child (Photo by Pavel Danilyuk: https://www.pexels.com/photo/a-robot-imitating-a-girl-s-movement-8294811/)

Now in machine learning, we are often trying 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 reinforcement learning 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 impractical to learn from scratch.

We'll see that the most naive form of imitation learning, called **behavioral cloning** (or "behavior 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.

## 7.2 Behavioral cloning

This notion of "learning from human-provided data" may remind you of the basic premise of Chapter 4. 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 **behavioral cloning**.

**Definition 7.1** (Behavioral cloning)**.**

1. Collect a training dataset of trajectories $\mathcal{D} = (s^n, a^n)_{n=1}^N$ generated by an **expert policy** $\pi_{\text{expert}}$. (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 Section 4.3.2):

$$\tilde{\pi} = \arg\min_{\pi \in \Pi} \sum_{n=0}^{N-1} \text{loss}(\pi(s^n), a^n)$$

where $\Pi$ 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 $\pi$ that outputs a *distribution* over actions. This allows us to assign a *likelihood* to observing the entire dataset $\mathcal{D}$ under the policy $\pi$, as if the state-action pairs are independent:

$$\mathbb{P}_\pi(\mathcal{D}) = \prod_{n=1}^{N} \pi(a_n \mid s_n)$$

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:

$$\tilde{\pi} = \arg\max_{\pi \in \Pi} \mathbb{P}_\pi(\mathcal{D})$$

This is also equivalent to doing empirical risk minimization with the **negative log likelihood** as the loss function:

$$\begin{aligned}\tilde{\pi} &= \arg\min_{\pi \in \Pi} -\log \mathbb{P}_\pi(\mathcal{D}) \\ &= \arg\min_{\pi \in \Pi} \sum_{n=1}^{N} -\log \pi(a_n \mid s_n)\end{aligned}$$

### 7.2.1 Performance of behavioral cloning

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. Suppose the learned policy obtains $\varepsilon$ *classification error*. That is, for trajectories drawn from the expert policy, the learned policy chooses a different action at most $\varepsilon$ of the time:

$$\mathbb{E}_{\tau \sim \rho_{\pi_{\text{expert}}}} \left[ \frac{1}{H} \sum_{h=0}^{H-1} \mathbf{1} \left\{ \tilde{\pi}(s_h) \neq \pi_{\text{expert}}(s_h) \right\} \right] \leq \varepsilon$$

Then, their value functions differ by

$$|V^{\pi_{\text{expert}}} - V^{\tilde{\pi}}| \leq H^2 \varepsilon$$

where $H$ is the horizon.

#### 7.2.1.1 Performance of behavioral cloning

Recall the Performance Difference Lemma (Theorem 6.1) allows us to express the difference between $\pi -$ expert and $\tilde{\pi}$ as

$$V_0^{\pi_{\text{expert}}}(s) - V_0^{\tilde{\pi}}(s) = \mathbb{E}_{\tau \sim \rho^{\pi_{\text{expert}}} | s_0 = s} \left[ \sum_{h=0}^{H-1} A_h^{\tilde{\pi}}(s_h, a_h) \right]. \tag{7.1}$$

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 $\pi_{\text{expert}}$ is deterministic, the advantage of the chosen action is exactly zero:

$$A^{\pi_{\text{expert}}}(s, \pi_{\text{expert}}(s)) = Q^{\pi_{\text{expert}}}(s, \pi_{\text{expert}}(s)) - V^{\pi_{\text{expert}}}(s) = 0.$$

But the right-hand-side of Equation 7.1 uses $A^{\widetilde{\pi}}$, not $A^{\pi-\text{expert}}$. To bridge this gap, we now use the assumption that $\widetilde{\pi}$ obtains $\varepsilon$ classification error. Note that $A_h^{\widetilde{\pi}}(s_h, \pi_{\text{expert}}(s_h)) = 0$ when $\pi_{\text{expert}}(s_h) = \widetilde{\pi}(s_h)$. In the case where the two policies differ on $s_h$, which occurs with probability $\varepsilon$, the advantage is naively upper bounded by $H$ (assuming rewards are bounded between 0 and 1). Taking the final sum gives the desired bound.

## 7.3 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 not perform well if this distribution 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 behavior; 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*.

## 7.4 Dataset aggregation (DAgger)

The DAgger algorithm (Ross, Gordon, and Bagnell (2010)) 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 behavioral 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 7.2** (DAgger algorithm)**.** Inputs: $\pi_{\text{expert}}$, an initial policy $\pi_{\text{init}}$, 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 $t = 1, \ldots, T$:
   - Collect $N$ trajectories $\tau_1, \ldots, \tau_N$ using the current policy $\pi$.
   - For each trajectory $\tau_n$:
     - Replace each action $a_h$ in $\tau_n$ with the **expert action** $\pi_{\text{expert}}(s_h)$.
     - Call the resulting trajectory $\tau_n^{\text{expert}}$.
   - $\mathcal{D} \leftarrow \mathcal{D} \cup \{\tau_1^{\text{expert}}, \ldots, \tau_n^{\text{expert}}\}$.
   - Let $\pi \leftarrow \texttt{fit}(\mathcal{D})$, where `fit` is a behavioral cloning algorithm.
3. Return $\pi$.

We leave the implementation as an exercise. How well does DAgger perform? A full proof can be found in Ross, Gordon, and Bagnell (2010) that under certain assumptions, the DAgger algorithm can better approximate the expert policy:

$$|V^{\pi_{\text{expert}}} - V^{\pi_{\text{DAgger}}}| \leq H\varepsilon$$

where $\varepsilon$ is the "classification error" guaranteed by the supervised learning algorithm.

## 7.5 Summary

For tasks where it is too difficult or expensive to learn from scratch, we can instead start off with a collection of **expert demonstrations**. Then we can use supervised learning techniques 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 **behavioral 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 behavioral cloning.

# 8 Tree Search Methods

---

---

## 8.1 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.

```
%load_ext autoreload
%autoreload 2
```

```
from utils import Int, Array, latex, jnp, NamedTuple
from enum import IntEnum
```

## 8.2 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.* 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.

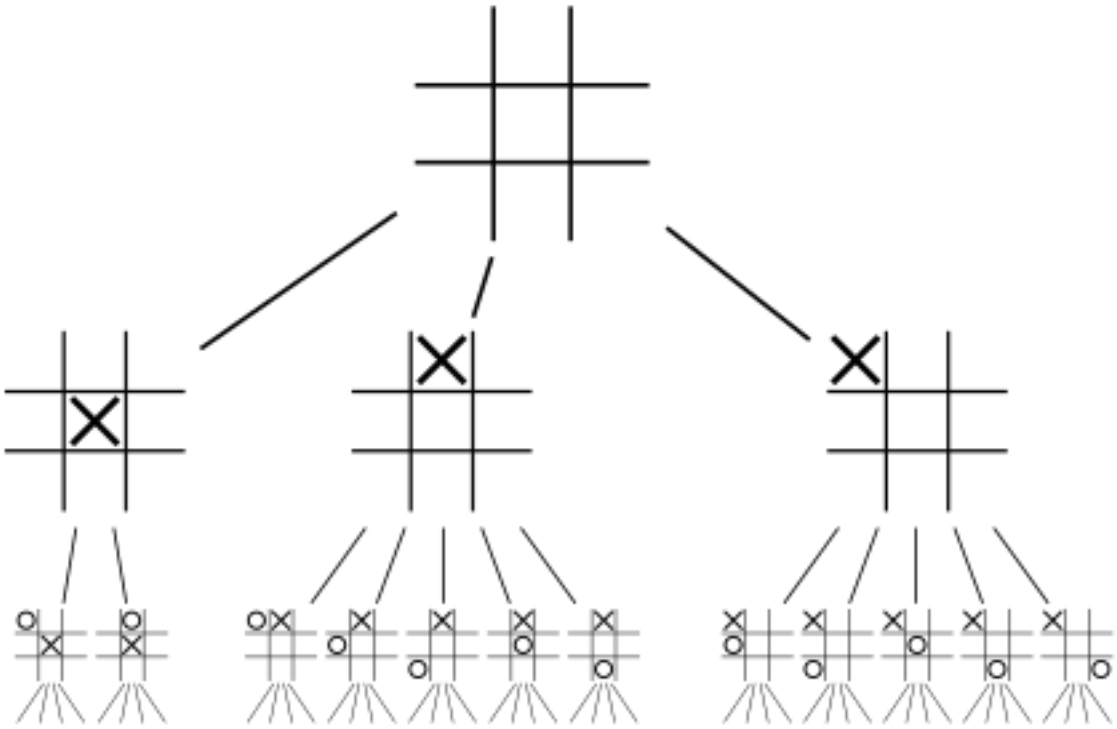


Figure 8.1: The first two layers of the complete game tree of tic-tac-toe.

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 Section 8.3. 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 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.

### 8.2.1 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**.

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 8.1** (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 1. 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 behavior of *both* players at the same time, describing such a game as an MDP is more trouble than it's worth.

```
class Player(IntEnum):
    EMPTY = 0
    X = 1
    O = 2


if False:
    class TicTacToeEnv(gym.Env):
        metadata = {"render.modes": ["human"]}

        def __init__(self):
            super().__init__()
            self.action_space = spaces.Discrete(9)
            self.observation_space = spaces.Box(
                low=0, high=2, shape=(3, 3), dtype=jnp.int32
            )
            self.board = None
            self.current_player = None
            self.done = None

        def reset(self, seed=None, options=None):
            super().reset(seed=seed)
            self.board = jnp.zeros((3, 3), dtype=jnp.int32)
            self.current_player = Player.X
            self.done = False
            return self.board, {}

        def step(self, action: jnp.int32) -> Int[Array, "3 3"]:
            """Take the action a in state s."""
            if self.done:
                raise ValueError("The game is already over. Call `env.reset()` to reset the e

            row, col = divmod(action, 3)
            if self.board[row, col] != Player.EMPTY:
                return self.board, -10
            return s.at[row, col].set(player)

        @staticmethod
        def is_terminal(s: Int[Array, "3 3"]):
            """Check if the game is over."""
```

```
        return is_winner(s, Player.X) or is_winner(s, Player.O) or jnp.all(s == Player.EM

    @staticmethod
    def is_winner(board: Int[Array, "3 3"], player: Player):
        """Check if the given player has won."""
        return any(
            jnp.all(board[i, :] == player) or
            jnp.all(board[:, i] == player)
            for i in range(3)
        ) or jnp.all(jnp.diag(board) == player) or jnp.all(jnp.diag(jnp.fliplr(board)) ==

    @staticmethod
    def show(s: Int[Array, "3 3"]):
        """Print the board."""
        for row in range(3):
            print(" | ".join(" XO"[s[row, col]] for col in range(3)))
            if row < 2:
                print("-" * 5)
```

## 8.3 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 "obvious" 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 8.1** (Min-max search algorithm)**.**

$$V_h^\star(s) = \begin{cases} r(s) & h = H \\ \max_{a \in \mathcal{A}_h(s)} V_{h+1}^\star(P(s,a)) & h \text{ is even and } h < H \\ \min_{a \in \mathcal{A}_h(s)} V_{h+1}^\star(P(s,a)) & h \text{ is odd and } h < H \end{cases}$$

We can compute this by starting at the terminal states, when the game's outcome is known, and working backwards, assuming that Max chooses the action that leads to the highest score and Min chooses the action that leads to the lowest score.

This translates directly into a recursive depth-first search algorithm for searching the complete game tree.

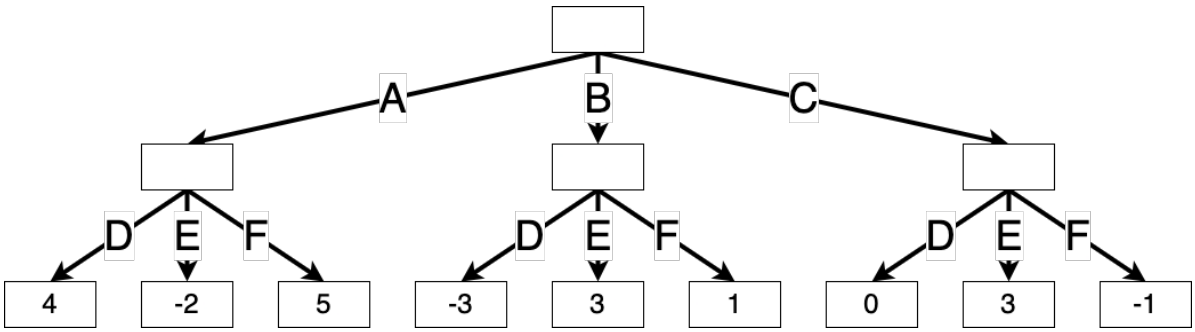
```
def minimax_search(s, player) -> tuple["Action", "Value"]:
    # Return the value of the state (for Max) and the best action for Max to take.
    if env.is_terminal(s):
        return None, env.winner(s)

    if player is max:
        a_max, v_max = None, None
        for a in env.action_space(s):
            _, v = minimax_search(env.step(s, a), min)
            if v > v_max:
                a_max, v_max = a, v
        return a_max, v_max
    else:
        a_min, v_min = None, None
        for a in env.action_space(s):
            _, v = minimax_search(env.step(s, a), max)
            if v < v_min:
                a_min, v_min = a, v
        return a_min, v_min

latex(minimax_search, id_to_latex={"env.step": "P", "env.action_space": r"\mathcal{A}"})
```

```
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
```

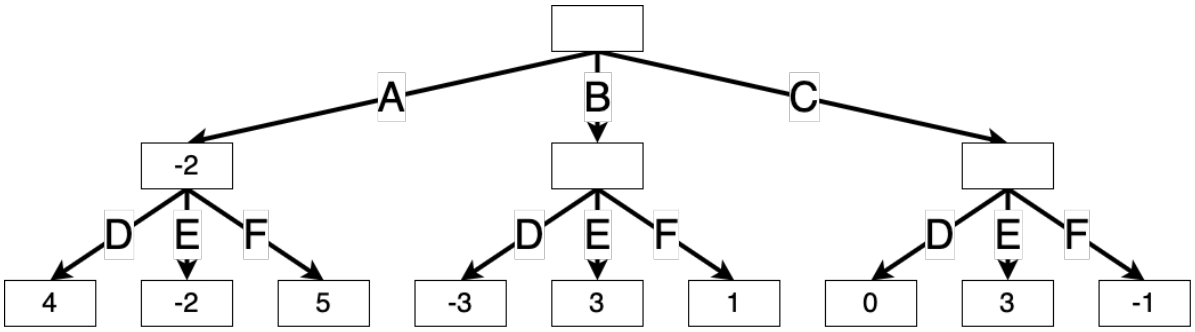
**Example 8.2** (Min-max search for a simple game)**.** Consider a simple game with just two steps: Max chooses one of three possible actions (A, B, C), and then Min chooses one of three possible actions (D, E, F). The combination leads to a certain integer outcome, shown in the table below:

| | D | E | F |
|---|---|---|---|
| A | 4 | -2 | 5 |
| B | -3 | 3 | 1 |
| C | 0 | 3 | -1 |

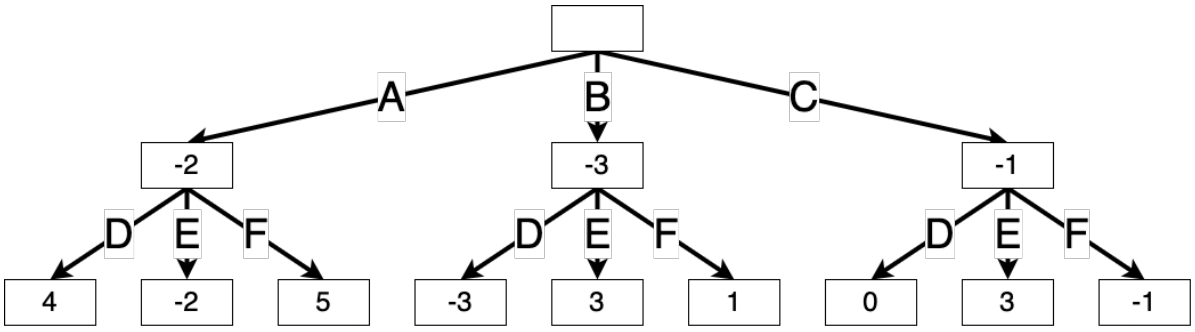
We can visualize this as the following complete game tree, where each box contains the value $V_h^\star(s)$ of that node. The min-max values of the terminal states are already known:



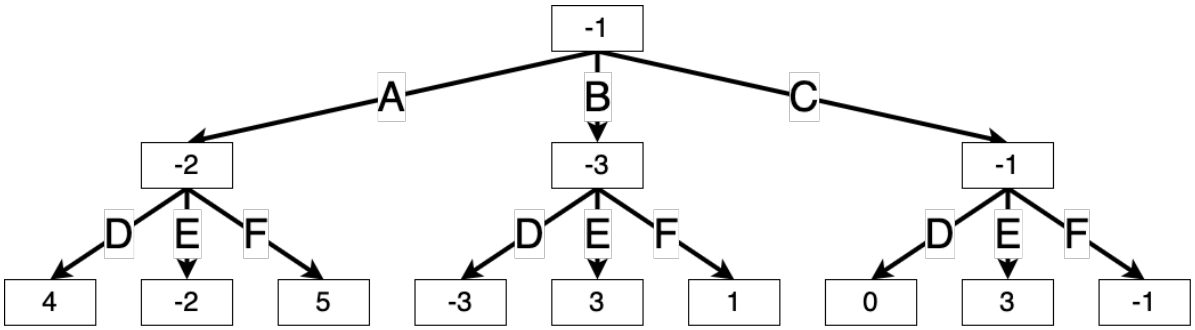

We begin min-max search at the root, exploring each of Max's actions. Suppose Max chooses action A. Then Min will choose action E to minimize the game score, making the value of this game node $\min(4, -2, 5) = -2$.




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 $\max(-2, -3, -1) = -1$.




### 8.3.1 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.

## 8.4 Alpha-beta search

The intuition behind alpha-beta search is as follows: Suppose Max is in state $s$, and considering whether to take action $a$ or $a'$. If at any point they find out that action $a'$ is definitely worse than (or equal to) action $a$, they don't need to evaluate action $a'$ any further.

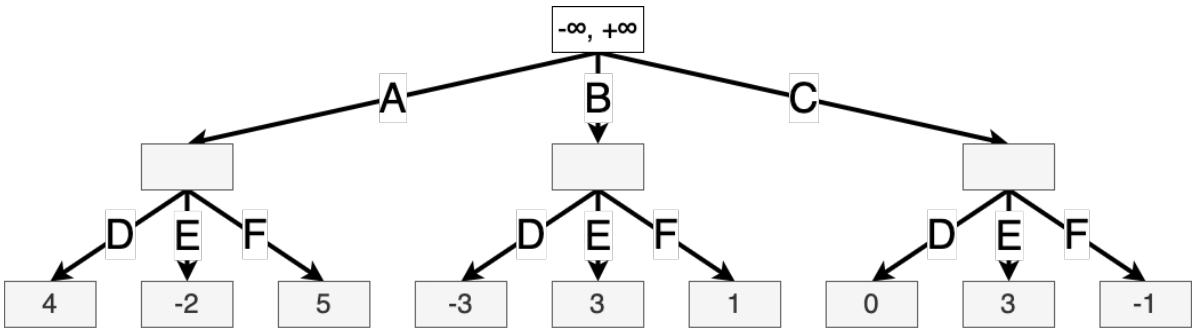
Concretely, we run min-max search as above, except now we keep track of two additional parameters $\alpha(s)$ and $\beta(s)$ while evaluating each state:

- 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.

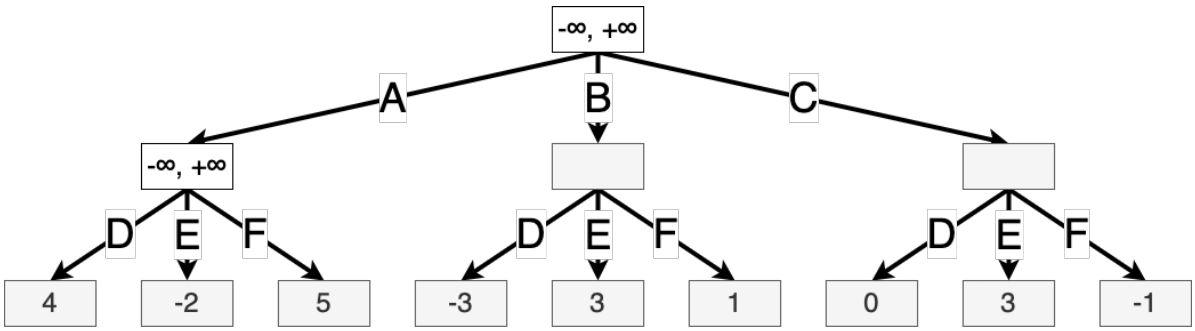
Suppose we are evaluating $V_h^\star(s)$, where it is Max's turn ($h$ is even). We update $\alpha(s)$ to be the *highest* minimax value achievable from $s$ so far. That is, the value of $s$ is *at least* $\alpha(s)$. Suppose Max chooses action $a$, which leads to state $s'$, in which it is Min's turn. 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.

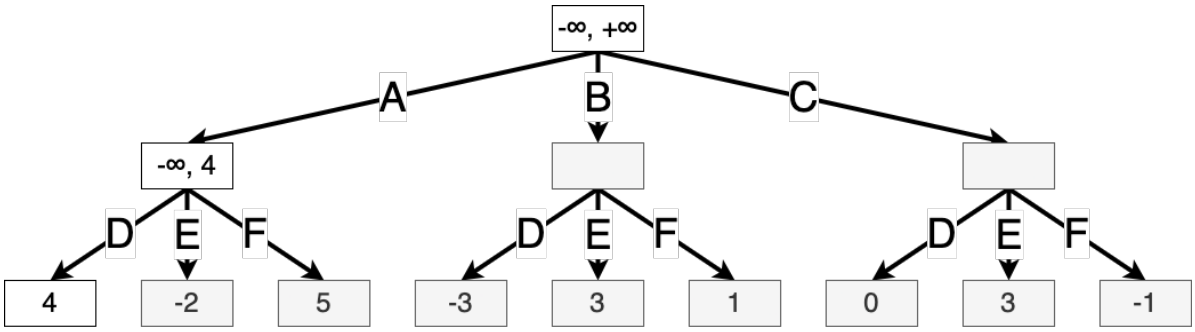
**Example 8.3** (Alpha-beta search for a simple game)**.** Let us use the same simple game from Example 8.2. We list the values of $\alpha(s), \beta(s)$ in each node throughout the algorithm. These values are initialized to $-\infty, +\infty$ respectively. We shade any squares that have not been visited by the algorithm, and we assume that actions are evaluated from left to right.

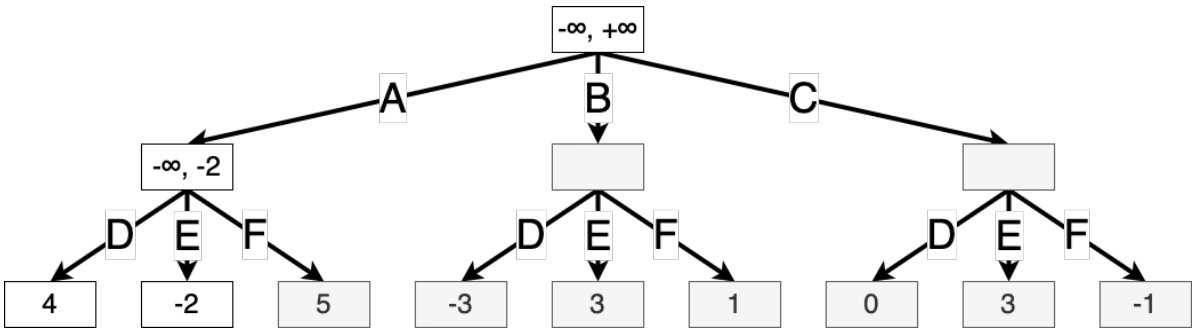



Suppose Max takes action A. Let $s'$ be the resulting game state. The values of $\alpha(s')$ and $\beta(s')$ are initialized at the same values as the root state, since we want to prune a subtree if there exists a better action at any step higher in the tree.

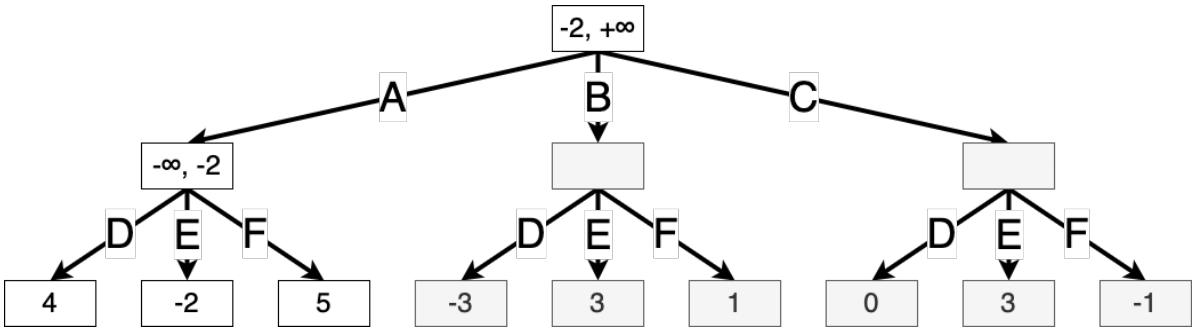



Then we iterate through Min's possible actions, updating the value of $\beta(s')$ as we go.

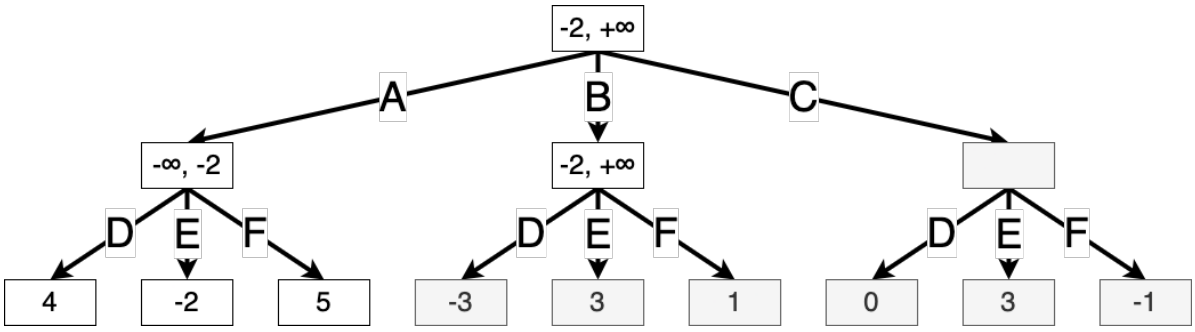




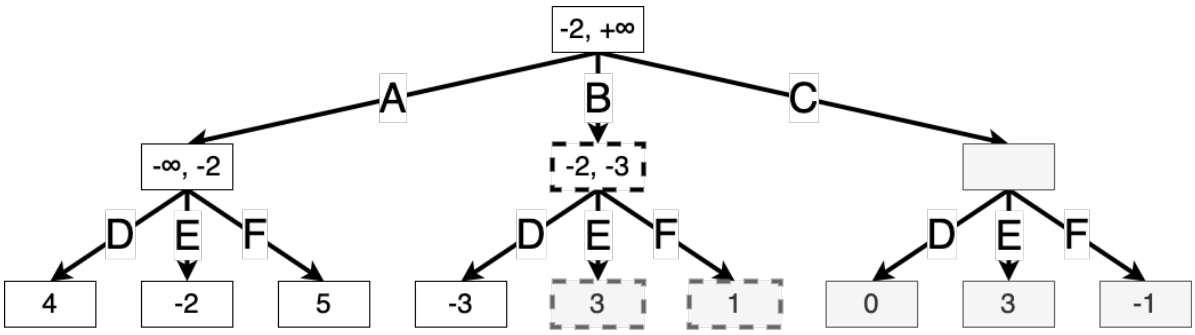

Once the value of state $s'$ is fully evaluated, we know that Max can achieve a value of *at least* $-2$ starting from the root, and so we update $\alpha(s)$, where $s$ is the root state:



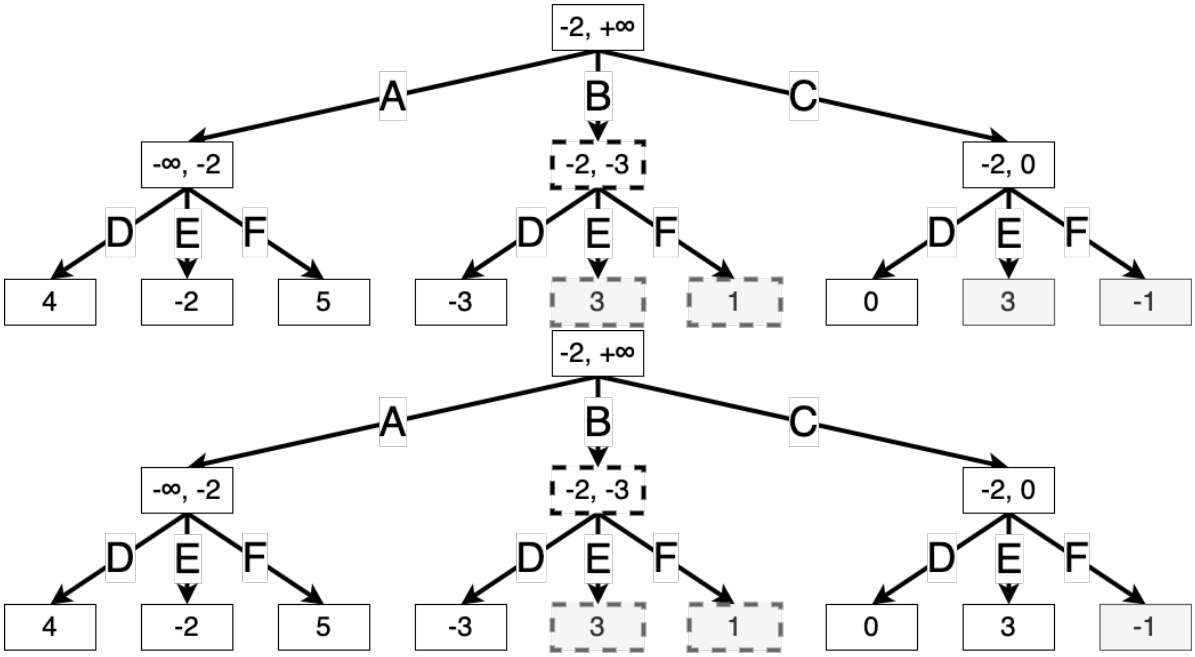

Then Max imagines taking action B. Again, let $s'$ denote the resulting game state. We initialize $\alpha(s')$ and $\beta(s')$ from the root:



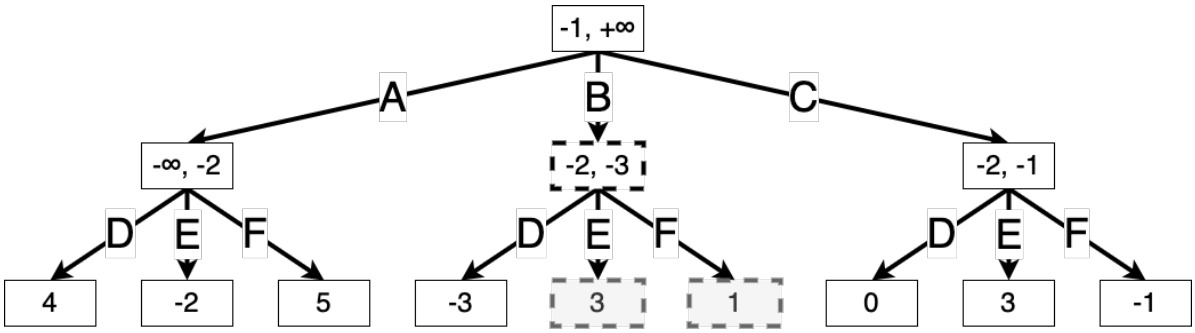

Now suppose Min takes action D, resulting in a value of $-3$. We see that $V_h^\star(s') = \min(-3, x, y)$, where $x$ and $y$ are the values of the remaining two actions. But since $\min(-3, x, y) \leq -3$, we know that the value of $s'$ is at most $-3$. But Max can achieve a better value of $\alpha(s') = -2$ 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.



```
def alpha_beta_search(s, player, alpha, beta) -> tuple["Action", "Value"]:
    # Return the value of the state (for Max) and the best action for Max to take.
    if env.is_terminal(s):
        return None, env.winner(s)

    if player is max:
        a_max, v_max = None, None
        for a in actions:
            _, v = minimax_search(env.step(s, a), min, alpha, beta)
            if v > v_max:
                a_max, v_max = a, v
                alpha = max(alpha, v)
            if v_max >= beta:
                # we know Min will not choose the action that leads to this state
                return a_max, v_max
        return a_max, v_max

    else:
        a_min, v_min = None, None
        for a in actions:
            _, v = minimax_search(env.step(s, a), max)
            if v < v_min:
                a_min, v_min = a, v
                beta = min(beta, v)
            if v_min <= alpha:
                # we know Max will not choose the action that leads to this state
                return a_min, v_min
        return a_min, v_min


latex(alpha_beta_search)
```

```
function alpha_beta_search(s, player, α, β)
  if env.is_terminal(s)
    return (None, env.winner(s))
  end if
  if player ≡ max
    (a_max, v_max) ← (None, None)
    for a ∈ actions do
      (_, v) ← minimax_search(env.step(s, a), min, α, β)
      if v > v_max
        (a_max, v_max) ← (a, v)
        α ← max(α, v)
      end if
      if v_max ≥ β
        return (a_max, v_max)
      end if
    end for
    return (a_max, v_max)
  else
    (a_min, v_min) ← (None, None)
    for a ∈ actions do
      (_, v) ← minimax_search(env.step(s, a), max)
      if v < v_min
        (a_min, v_min) ← (a, v)
        β ← min(β, v)
      end if
      if v_min ≤ α
        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?

## 8.5 Monte Carlo Tree Search

The task of evaluating actions in a complex environment might seem familiar. We've encountered this problem before in both the Chapter 3 setting and the Chapter 1 setting. 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 complete game tree in which each path represents an *entire game*. The goal of MCTS is to assign values to only the game states that are *relevant* to the *current game*; We gradually expand the tree 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 Chapter 3 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 Section 3.6 algorithm.

*Remark* 8.1 (Summary of UCB). Let us quickly review the UCB bandit algorithm. For each arm $k$, we track the sample mean

$$\hat{\mu}_t^k = \frac{1}{N_t^k} \sum_{\tau=0}^{t-1} \mathbf{1}\{a_\tau = k\} r_\tau$$

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 $\delta$ (some fixed parameter we choose), the true mean $\mu^k$ 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 (corresponding to a state-action pair $(s, a)$) in the game tree, 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 ($\sum_{\tau=0}^{t-1} \mathbf{1}\{(s_\tau, a_\tau) = (s, a)\} r_\tau$). 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 8.2** (Monte Carlo tree search algorithm)**.** Inputs: - $T$, the number of iterations per move - $\pi_{\text{rollout}}$, the **rollout policy** for randomly sampling games - $c$, a positive value that encourages exploration

To choose a single move starting at state $s_{\text{start}}$, 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 $\tau$ 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
     $$\text{UCB}^{s,a} = \frac{W^{s,a}}{N^{s,a}} + c\sqrt{\frac{\ln N^s}{N^{s,a}}} \tag{8.1}$$
     - Append $(s, a)$ to $\tau$
     - Set $s \leftarrow P(s, a)$
2. **Expansion**: Let $s_{\text{new}}$ denote the final state in $\tau$ (that has at least one action that hasn't been taken). Choose one of these unexplored actions from $s_{\text{new}}$. Call it $a_{\text{new}}$. Add it to $\tau$.
3. **Simulation**: Simulate a complete game episode by starting with the action $a_{\text{new}}$ and then playing according to $\pi_{\text{rollout}}$. 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 Equation 8.1. 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 $\pi_{\text{rollout}}$. If the distribution $\pi_{\text{rollout}}$ induces over games is very different from the distribution seen during real gameplay, we might end up with a poor value approximation.

### 8.5.1 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 Definition 8.2 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 Equation 8.1 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 8.3** (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 - $\pi_{\text{guide}}$, a guiding policy that encourages certain actions - $c$, a positive value that encourages exploration

To select a move in state $s_{\text{start}}$, we repeat the following four steps $T$ times:

1. **Selection**: We start at $s = s_{\text{start}}$. Let $\tau$ 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

$$\text{UCB}^{s,a} = \frac{W^{s,a}}{N^s} + c \cdot \pi_{\text{guide}}(a \mid s)\sqrt{\frac{\ln N^s}{N^{s,a}}} \tag{8.2}$$

     - Append $(s, a)$ to $\tau$

- Set $s \leftarrow P(s, a)$

2. **Expansion**: Let $s_{\text{new}}$ denote the final state in $\tau$ (that has at least one action that hasn't been taken). Choose one of these unexplored actions from $s_{\text{new}}$. Call it $a_{\text{new}}$. Add it to $\tau$.
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 $a_{\text{new}}$.
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 Equation 8.2. Then play continues. As before, we can reuse the tree across timesteps.

```python
class EdgeStatistics(NamedTuple):
    wins: int = 0
    visits: int = 0

class MCTSTree:
    """A representation of the search tree.

    Maps each state-action pair to its number of wins and the number of visits.
    """

    edges: dict[tuple["State", "Action"], EdgeStatistics]

def mcts_iter(tree, s_init):
    s = s_init
    # while all((s, a) in tree for a in env.action_state(s)):
```

How do we actually compute a useful $\pi_{\text{guide}}$ and $v$? If we have some existing dataset of trajectories, we could use Chapter 7 (that is, imitation learning) to generate a policy $\pi_{\text{guide}}$ via behavioral cloning and learn $v$ by regressing the game outcomes onto states. Then, plugging these into Definition 8.3 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**.

### 8.5.2 Self-play

Recall the Section 1.5.3.2 algorithm from the Chapter 1. Policy iteration alternates between **policy evaluation** (taking $\pi$ and computing $V^\pi$) and **policy improvement** (setting $\pi$ to

be greedy with respect to $V^\pi$). Above, we saw how MCTS can be thought of as a "policy improvement" operation: for a given policy $\pi^0$, we can use it to guide MCTS, resulting in an algorithm that is itself a policy $\pi^0_{\text{MCTS}}$ that maps from states to actions. Now, we can use Chapter 7 to obtain a new policy $\pi^1$ that imitates $\pi^0_{\text{MCTS}}$. We can now use $\pi^1$ to guide MCTS, and repeat.

**Definition 8.4** (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 $\theta^0, \lambda^0$

For $t = 0, \dots, T-1$:

- **Policy improvement**: Let $\pi^t_{\text{MCTS}}$ denote the policy obtained by Definition 8.3 with $\pi - \theta^t$ 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 behavioral cloning to find a set of policy parameters $\theta^{t+1}$ that mimic the behavior of $\pi^t_{\text{MCTS}}$ and a set of value function parameters $\lambda^{t+1}$ that approximate its value function. That is,

$$\theta^{t+1} \leftarrow \arg\min_\theta \sum_{m=0}^{M-1} \sum_{h=0}^{H-1} -\log \pi_\theta(a_h^m \mid s_h^m)$$

$$\lambda^{t+1} \leftarrow \arg\min_\lambda \sum_{m=0}^{M-1} \sum_{h=0}^{H-1} (v_\lambda(s_h^m) - R(\tau_m))^2$$

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).

## 8.6 Summary

In this chapter, we explored tree search-based algorithms for deterministic, zero sum, fully observable two-player games. We began with Section 8.3, 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. Section 8.4 does this by -pruning- away states that we already know to be suboptimal, and Section 8.5 -approximates- the value of states instead of evaluating them exactly.

## 8.7 References

Chapter 5 of Russell and Norvig (2021) 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 from scratch. 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.

# 9 Exploration in MDPs

## 9.1 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. The algorithms we saw in the chapter on fitted DP Chapter 5 suffer from this issue.

In Chapter 3, where the state never changes so all we care about are the actions, we saw algorithms like Section 3.6 and Section 3.7 that incentivize the learner to explore arms that it is uncertain about. In this chapter, we will see how to generalize these ideas to the MDP setting.

**Definition 9.1** (Per-episode regret)**.** To quantify the performance of a learning algorithm, we will consider its per-episode regret over $T$ timesteps/episodes:

$$\text{Regret}_T = \mathbb{E}\left[\sum_{t=0}^{T-1} V_0^\star(s_0) - V_0^{\pi^t}(s_0)\right]$$

where $\pi^t$ is the policy generated by the algorithm at the $t$th iteration.

### 9.1.1 Sparse reward

Exploration is crucial in unknown **sparse reward** problems where reward doesn't come until after many steps, and algorithms which do not *systematically* explore new states may fail to learn anything meaningful (within a reasonable amount of time). We can see this by considering the following simple MDP:

**Example 9.1** (Sparse Reward MDP)**.** Here's a simple example of an MDP with sparse reward:

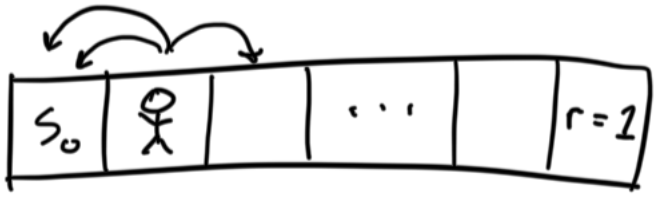



Figure 9.1: image

There are $|\mathcal{S}|$ states. The agent starts in the leftmost state. In every state, there are three possible actions, two of which move the agent left and one which moves the agent right. The reward function assigns $r = 1$ to the rightmost cell.

How well would the algorithms we've covered so far do on this problem? For example, Chapter 6 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. Chapter 5 will run into a similar issue: as we randomly interact with the environment, we will 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.

The rest of this chapter will consider ways to *explicitly* add exploration to these algorithms.

### 9.1.2 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 bandit setting, it was trivial to visit every arm: just pull each arm once. How might I navigate to a particular state $s$ in the MDP setting? Doing so requires planning out a path from the original state. In fact, solving navigation itself is a complex RL problem!

## 9.2 Explore-then-exploit (for deterministic MDPs)

In this simple deterministic setting, the environment never randomly takes us to unseen states, so our strategy must actively explore new states.

**Definition 9.2.** We'll keep a set $\mathcal{D}$ of all the $(s, a, r, s')$ pairs we've observed. Each episode, we'll use $\mathcal{D}$ to construct a fully known MDP, $M_{\mathcal{D}}$, in which unseen state-action pairs are rewarded. We'll then use the planning algorithms from Section 1.3.2 to reach the unknown states in $M - \mathcal{D}$.

We assume that every state can be reached from the initial state within a single episode, and that the action space $\mathcal{A}$ is fixed.

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 Section 1.3.2 on the known MDP $M - \mathcal{D}$ to compute policy $\pi^\star_{\mathcal{D}}$.
    3. Execute $\pi^\star_{\mathcal{D}}$. This will visit some $(s, a)$ not yet observed in $\mathcal{D}$.
    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.

**Theorem 9.1** (Performance of explore-then-exploit)**.** *As long as every state can be reached from $s_0$ within a single episode, i.e. $|\mathcal{S}| \leq H$, this will eventually be able to explore all $|\mathcal{S}||\mathcal{A}|$ state-action pairs, adding one new transition per episode. We know it will take at most $|\mathcal{S}||\mathcal{A}|$ iterations to explore the entire MDP, after which $\pi^t = \pi^\star$, incurring no additional regret. For each $\pi^t$ up until then, corresponding to the shortest-path policies $\tilde{\pi}$, the value of policy $\pi^t$ will differ from that of $\pi^\star$ by at most $H$, since the policies will differ by at most 1 reward at each timestep. So,*

$$\sum_{t=0}^{T-1} V_0^\star - V_0^{\pi^t} \leq |\mathcal{S}||\mathcal{A}|H.$$

*(Note that this MDP and algorithm are deterministic, so the regret is not random.)*

## 9.3 Treating an unknown MDP as a MAB

We also explored the exploration-exploitation tradeoff in Chapter 3. Recall tthat in the MAB setting, we have $K$ arms, each of which has an unknown reward distribution, and we want to learn which of the arms is *optimal*, i.e. has the highest mean reward.

One algorithm that struck a good balance between exploration and exploitation was the **upper confidence bound** algorithm Section 3.6: For each arm, we construct a *confidence interval* for its true mean award, and then choose the arm with the highest upper confidence bound. In summary,

$$k_{t+1} \leftarrow \arg\max_{k \in [K]} \frac{R_t^k}{N_t^k} + \sqrt{\frac{\ln(2t/\delta)}{2N_t^k}}$$

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. How might we extend UCB to the MDP case?

Let us formally describe an unknown MDP as an MAB problem. In an unknown MDP, we want to learn which *policy* is optimal. So if we want to apply MAB techniques to solving an MDP, it makes sense to think of *arms* as *policies*. There are $K = (|\mathcal{A}|^{|\mathcal{S}|})^H$ deterministic policies in a finite MDP. Then, "pulling" arm $\pi$ corresponds to using $\pi$ to act through a trajectory in the MDP, and observing the total reward.

**Exercise 9.1** (Notation for expected reward)**.** Which quantity that we have seen so far equals the expected reward from arm $\pi$?

Recall that UCB incurs regret $\tilde{O}(\sqrt{TK})$, 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

$$\tilde{O}(\sqrt{|\mathcal{A}|^{|\mathcal{S}|H}T}) \tag{9.1}$$

This scales *exponentially* in $|\mathcal{S}|$ and $H$, which quickly becomes intractable. Notably, this method doesn't consider the information that we gain across different policies. We can illustrate this with the following example:

**Example 9.2** (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 $\pi_{\mathrm{Y}}$ and $\pi_{\mathrm{N}}$. 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.

## 9.4 UCB-VI

The approach above is inefficient: 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. Can we borrow ideas from UCB to reduce the regret to a polynomial in $|\mathcal{S}|$, $|\mathcal{A}|$, and $H$?

One way to frame the UCB algorithm is that, when choosing arms, we optimize over a *proxy reward* that is the sum of the estimated mean reward and an **exploration term**. In the **UCB-VI** algorithm, we will extend this idea to the case of an unknown MDP $\mathcal{M}^?$ by modeling a proxy MDP $\widetilde{\mathcal{M}}$ with a reward function that encourages exploration. Then, we will use DP to solve for the optimal policy in $\widetilde{\mathcal{M}}$.

**Assumptions:** For simplicity, here we assume the reward function of $\mathcal{M}^?$ is known, so we only need to model the state transitions, though the rewards can be modelled similarly. We will also consider the more general case of a **time-varying** MDP, where the transition and reward functions may change over time. We take the convention that $P_h$ is the distribution of $s_{h+1} \mid s_h, a_h$ and $r_h$ is applied to $s_h, a_h$.

At a high level, the UCB-VI algorithm can be described as follows:

1. **modeling:** Use previous data to model the transitions $\widehat{P}_0, \dots, \widehat{P}_{H-1}$.
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 DP to compute the optimal policy $\hat{\pi}_h(s)$ in the modelled MDP

$$\tilde{\mathcal{M}} = (\mathcal{S}, \mathcal{A}, \{\widehat{P}_h\}_{h\in[H]}, \{r_h + b_h\}_{h\in[H]}, H).$$

4. **Execution:** Use $\hat{\pi}_h(s)$ to collect a new trajectory, and repeat.

We detail each of these steps below. The full definition follows in Definition 9.3.

### 9.4.1 modeling the transitions

We seek to approximate $P_h(s_{h+1} \mid s_h, a_h) = \frac{\mathbb{P}(s_h, a_h, s_{h+1})}{\mathbb{P}(s_h, a_h)}$. We can estimate these using their sample probabilities from the dataset. That is, define

$$N_h^t(s,a,s') := \sum_{i=0}^{t-1} \mathbf{1}\left\{(s_h^i, a_h^i, s_{h+1}^i) = (s,a,s')\right\}$$
$$N_h^t(s,a) := \sum_{i=0}^{t-1} \mathbf{1}\left\{(s_h^i, a_h^i) = (s,a)\right\}$$

Then we can model

$$\widehat{P}_h^t(s' \mid s,a) = \frac{N_h^t(s,a,s')}{N_h^t(s,a)}.$$

Note that this is also 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.

### 9.4.2 Reward bonus

To motivate the reward bonus term $b_h^t(s,a)$, recall how we designed the reward bonus term for UCB:

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 $\mu^k$.
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 $\delta$ by a factor of $T$.

We'd like to do the same for UCB-VI, 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

$$b_h^t(s,a) = 2H\sqrt{\frac{\log(|\mathcal{S}||\mathcal{A}|HT/\delta)}{N_h^t(s,a)}}. \tag{9.2}$$

We will only provide a heuristic sketch of the proof; see (Agarwal et al. 2022) (Section 7.3) for a full proof.

*Remark* 9.1 (UCB-VI reward bonus construction). 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:

$$\widehat{V}_h^t(s) = \max_{a \in \mathcal{A}} \left[ \tilde{r}_h^t(s,a) + \mathbb{E}_{s' \sim \widehat{P}_h^t(\cdot|s,a)} \left[ \widehat{V}_{h+1}^t(s') \right] \right]$$

where $\tilde{r}_h^t(s,a) = r_h(s,a) + b_h^t(s,a)$ is the reward function of our modelled MDP $\tilde{\mathcal{M}}^t$. On the other hand, we know that $V^\star$ must satisfy

$$V_h^\star(s) = \max_{a \in \mathcal{A}} \left[ \tilde{r}_h^t(s,a) + \mathbb{E}_{s' \sim P_h^?(\cdot|s,a)} [V_{h+1}^\star(s')] \right]$$

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:

$$\text{error} = \left| \mathbb{E}_{s' \sim \widehat{P}_h^t(\cdot|s,a)} \left[ V_{h+1}^\star(s') \right] - \mathbb{E}_{s' \sim P_h^?(\cdot|s,a)} \left[ V_{h+1}^\star(s') \right]. \right| \tag{9.3}$$

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

$$\begin{aligned}
\mathbb{E}_{s' \sim \widehat{P}_h^t(\cdot|s,a)} \left[ V_{h+1}^\star(s') \right] &= \sum_{s' \in \mathcal{S}} \frac{N_h^t(s,a,s')}{N_h^t(s,a)} V_{h+1}^\star(s') \\
&= \frac{1}{N_h^t(s,a)} \sum_{i=0}^{t-1} \sum_{s' \in \mathcal{S}} \mathbf{1}\left\{ (s_h^i, a_h^i, s_{h+1}^i) = (s,a,s') \right\} V_{h+1}^\star(s') \\
&= \frac{1}{N_h^t(s,a)} \sum_{i=0}^{t-1} \underbrace{\mathbf{1}\left\{ (s_h^i, a_h^i) = (s,a) \right\} V_{h+1}^\star(s_{h+1}^i)}_{X^i}
\end{aligned}$$

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 Equation 9.3 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 $h$):

$$\begin{aligned}
\mathbb{E}_{s' \sim P_h^?(\cdot|s,a)} \left[ V_{h+1}^\star(s') \right] &= \sum_{s' \in \mathcal{S}} P_h^?(s' \mid s,a) V_{h+1}^\star(s') \\
&= \sum_{s' \in \mathcal{S}} \frac{1}{N_h^t(s,a)} \sum_{i=0}^{t-1} \mathbf{1}\left\{ (s_h^i, a_h^i) = (s,a) \right\} P_h^?(s' \mid s,a) V_{h+1}^\star(s') \\
&= \frac{1}{N_h^t(s,a)} \sum_{i=0}^{t-1} \mathbb{E}_{s_{h+1}^i \sim P_h^?(\cdot|s_h^i, a_h^i)} X^i.
\end{aligned}$$

Now we can apply Hoeffding's inequality to $X^i - \mathbb{E}_{s_{h+1}^i \sim P_h^?(\cdot|s_h^i,a_h^i)} X^i$, which is bounded by $H$, to obtain that, with probability at least $1-\delta$,

$$\text{error} = \left| \frac{1}{N_h^t(s,a)} \sum_{i=0}^{t-1} \left( X^i - \mathbb{E}_{s_{h+1}^i \sim P_h^?(\cdot|s_h^i,a_h^i)} X^i \right) \right| \leq 2H \sqrt{\frac{\ln(1/\delta)}{N_h^t(s,a)}}.$$

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.

Putting these parts together, we can define the algorithm as follows:

**Definition 9.3** (UCB-VI algorithm)**.** TODO

### 9.4.3 Performance of UCB-VI

How exactly does UCB-VI 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 $\pi^t$ 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 $\pi^t$ is *exploiting* actions that are giving high reward.

If the r.h.s. is *large*, then we have overestimated the value: $\pi^t$, the optimal policy of $\tilde{\mathcal{M}}^t$, 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 UCB-VI achieves a regret of

**Theorem 9.2** (UCB-VI regret)**.**

$$\mathbb{E}\left[\sum_{t=0}^{T-1}\left(V_0^\star(s_0) - V_0^{\pi^t}(s_0)\right)\right] = \tilde{O}(H^2\sqrt{|\mathcal{S}||\mathcal{A}|T})$$

Comparing this to the UCB regret bound $\tilde{O}(\sqrt{TK})$, 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 Equation 9.1) 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:

$$\frac{1}{T}\mathbb{E}[\text{Regret}_T] = \tilde{O}\left(\sqrt{\frac{H^4|\mathcal{S}||\mathcal{A}|}{T}}\right)$$

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.

## 9.5 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 9.4** (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

$$\mathbb{E}_{s' \sim P_h(\cdot|s,a)}[V_{h+1}^\star(s)] = (\mu_h^\star \phi(s, a))^\top V_{h+1}^\star.$$

The $\phi$ 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.

### 9.5.1 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:

$$\begin{aligned}
Q_h^\star(s, a) &= r_h(s, a) + \mathbb{E}_{s' \sim P_h(\cdot|s,a)}[V_{h+1}^\star(s')] \\
&= \phi(s, a)^\top \theta_h^\star + (\mu_h^\star \phi(s, a))^\top V_{h+1}^\star \\
&= \phi(s, a)^\top \underbrace{(\theta_h^\star + (\mu_h^\star)^\top V_{h+1}^\star)}_{w_h} \\
V_h^\star(s) &= \max_a Q_h^\star(s, a) \\
\pi_h^\star(s) &= \arg\max_a Q_h^\star(s, a)
\end{aligned}$$

Show that $Q_h^\pi$ is also linear with respect to $\phi(s, a)$ for any policy $\pi$.

### 9.5.2 UCB-VI in a linear MDP

#### 9.5.2.1 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 $\delta_s$ 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

$$\mathbb{E}_{s' \sim P_h(\cdot|s,a)}[\delta_{s'}] = P_h(\cdot \mid s,a) = \mu_h^\star \phi(s,a).$$

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 $\lambda I$ 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 $\widehat{P}_h^t(\cdot \mid s,a) = \hat{\mu}_h^t \phi(s,a)$.

#### 9.5.2.2 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 Section 3.8.1:

$$b_h^t(s,a) = \beta\sqrt{\phi(s,a)^\top (A_h^t)^{-1} \phi(s,a)}, \quad \beta = \tilde{O}(dH).$$

Note that this isn't explicitly inversely proportional to $N_h^t(s,a)$ as in the original UCB-VI bonus term Equation 9.2. 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 UCB-VI algorithm Definition 9.3.

**Theorem 9.3** (LinUCB-VI regret)**.** *The LinUCB-VI algorithm achieves expected regret*

$$\mathbb{E}[Regret_T] = \mathbb{E}\left[\sum_{t=0}^{T-1} V_0^\star(s_0) - V_0^{\pi^t}(s_0)\right] \leq \tilde{O}(H^2 d^{1.5}\sqrt{T})$$

Comparing this to our bound for UCB-VI in an environment without this linear assumption, we see that we go from a sample complexity of $\tilde{\Omega}(H^4|\mathcal{S}||\mathcal{A}|)$ to $\tilde{\Omega}(H^4 d^3)$. This new sample complexity only depends on the feature dimension and not on the state or action space of the MDP!

## 9.6 Summary

In this chapter, we've explored how to explore in an unknown MDP.

- We first discussed the explore-then-exploit algorithm Definition 9.2, a simple way to explore a deterministic MDP by visiting all state-action pairs.
- We then discussed how to treat an unknown MDP as a MAB Section 9.3, and how this approach is inefficient since it doesn't make use of relationships between policies.
- We then introduced the UCB-VI algorithm Definition 9.3, 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 LinUCB-VI algorithm Section 9.5.2, which has a sample complexity independent of the size of the state and action spaces.

# 10 Appendix: Background

## 10.1 O notation

Throughout this chapter and the rest of the book, we will describe the asymptotic behavior 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 $\tilde{O}(g(t))$ to hide logarithmic factors. That is, $f(t) = \tilde{O}(g(t))$ 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 $O(f(t) + g(t))$.

# References


Agarwal, Alekh, Nan Jiang, Sham M Kakade, and Wen Sun. 2022. *Reinforcement Learning: Theory and Algorithms*. https://rltheorybook.github.io/rltheorybook_AJKS.pdf.

Baydin, Atilim Gunes, Barak A. Pearlmutter, Alexey Andreyevich Radul, and Jeffrey Mark Siskind. 2018. "Automatic Differentiation in Machine Learning: A Survey." February 5, 2018. https://doi.org/10.48550/arXiv.1502.05767.

Boyd, Stephen, and Lieven Vandenberghe. 2004. *Convex Optimization*. Cambridge University Press. https://web.stanford.edu/~boyd/cvxbook/.

Deng, Li. 2012. "The MNIST Database of Handwritten Digit Images for Machine Learning Research [Best of the Web]." *IEEE Signal Processing Magazine* 29 (6): 141–42. https://doi.org/10.1109/MSP.2012.2211477.

Hastie, Trevor, Robert Tibshirani, and Jerome Friedman. 2013. *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*. Springer Science & Business Media. https://books.google.com?id=yPfZBwAAQBAJ.

James, Gareth, Daniela Witten, Trevor Hastie, Robert Tibshirani, and Jonathan Taylor. 2023. *An Introduction to Statistical Learning: With Applications in Python*. Springer Texts in Statistics. Cham: Springer International Publishing. https://doi.org/10.1007/978-3-031-38747-0.

Lai, T. L, and Herbert Robbins. 1985. "Asymptotically Efficient Adaptive Allocation Rules." *Advances in Applied Mathematics* 6 (1): 4–22. https://doi.org/10.1016/0196-8858(85)90002-8.

Nielsen, Michael A. 2015. *Neural Networks and Deep Learning*. Determination Press. http://neuralnetworksanddeeplearning.com/.

Ross, Stéphane, Geoffrey J. Gordon, and J. Bagnell. 2010. "A Reduction of Imitation Learning and Structured Prediction to No-Regret Online Learning." In. https://www.semanticscholar.org/paper/A-Reduction-of-Imitation-Learning-and-Structured-to-Ross-Gordon/79ab3c49903ec8cb339437ccf5cf998607fc313e.

Russell, Stuart J., and Peter Norvig. 2021. *Artificial Intelligence: A Modern Approach*. Fourth edition. Pearson Series in Artificial Intelligence. Hoboken: Pearson.

Schrittwieser, Julian, Ioannis Antonoglou, Thomas Hubert, Karen Simonyan, Laurent Sifre, Simon Schmitt, Arthur Guez, et al. 2020. "Mastering Atari, Go, Chess and Shogi by Planning with a Learned Model." *Nature* 588 (7839, 7839): 604–9. https://doi.org/10.1038/s41586-020-03051-4.

Schulman, John, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. 2017. "Proximal Policy Optimization Algorithms." August 28, 2017. https://doi.org/10.48550/arXiv.1707.06347.

Silver, David, Aja Huang, Chris J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, et al. 2016. "Mastering the Game of Go with Deep Neural Networks and Tree Search." *Nature* 529 (7587, 7587): 484–89. https://doi.org/10.1038/nature16961.

Silver, David, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, et al. 2018. "A General Reinforcement Learning Algorithm That Masters Chess, Shogi, and Go Through Self-Play." *Science* 362 (6419): 1140–44. https://doi.org/10.1126/science.aar6404.

Silver, David, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, et al. 2017. "Mastering the Game of Go Without Human Knowledge." *Nature* 550 (7676, 7676): 354–59. https://doi.org/10.1038/nature24270.

Sussman, Gerald Jay, Jack Wisdom, and Will Farr. 2013. *Functional Differential Geometry*. Cambridge, MA: The MIT Press.

Sutton, Richard S., and Andrew G. Barto. 2018. *Reinforcement Learning: An Introduction*. Second edition. Adaptive Computation and Machine Learning Series. Cambridge, Massachusetts: The MIT Press. http://incompleteideas.net/book/RLbook2020trimmed.pdf.

Vershynin, Roman. 2018. *High-Dimensional Probability: An Introduction with Applications in Data Science*. Cambridge University Press. https://books.google.com?id=NDdqDwAAQBAJ.

Williams, Ronald J. 1992. "Simple Statistical Gradient-Following Algorithms for Connectionist Reinforcement Learning." *Machine Learning* 8 (3): 229–56. https://doi.org/10.1007/BF00992696.