CS/Stat 184
Introduction to Reinforcement Learning

Initially created by **Alexander D. Cai** during the first iteration of the course in Fall 2022.

Contents

Welcome to the study of reinforcement learning! This set of lecture notes accompanies the undergraduate course CS/STAT 184 and is intended to be a friendly yet rigorous introduction to this exciting and active subfield of machine learning. Here are some questions you might have before embarking on this journey:

What is reinforcement learning (RL)? Broadly speaking, RL is a subfield of machine learning that studies how an agent can learn to make sequential decisions in an environment.

Why study RL? RL provides a powerful framework for attacking a wide variety of problems, including robotic control, video games and board games, resource management, language modelling, 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 on top of RL.

Is this book for me? This book assumes familiarity with multivariable calculus, linear algebra, and probability. For Harvard undergraduates, this would be fulfilled by Math 21a, Math 21b, and Stat 110. Stat 111 is strongly recommended but not required. Here is a non-comprehensive list of topics of which this book will assume knowledge:

- **Linear Algebra:** Vectors, matrices, matrix multiplication, matrix inversion, eigenvalues and eigenvectors, and the Gram-Schmidt process.
- **Multivariable Calculus:** Partial derivatives, gradient, directional derivative, and the chain rule.
- **Probability:** Random variables, probability distributions, expectation, variance, covariance, conditional probability, Bayes' rule, and the law of total probability.

How does reinforcement learning differ from other machine learning paradigms? Here is a list of comparisons:

- **Supervised learning.** Supervised learning concerns itself with learning a mapping from inputs to outputs (e.g. image classification). Typically the data takes the form of input-output pairs that are assumed to be sampled independently from some generating distribution. In RL, however, the data is generated by the agent interacting with the environment, meaning the observations depend on the agent's behaviour and are not independent from each other. This requires a more general set of tools.

  Conversely, supervised learning is a well-studied field that provides many useful tools for RL. For example, it may be useful to use supervised learning to predict how valuable a given state is, or to predict the probability of transitioning to a given state.

0.1 Overview

Chapter 1 introduces **Markov Decision Processes**, the dominant mathematical framework for studying RL. We'll discuss **dynamic programming** algorithms for solving MDPs, including

policy evaluation, **policy iteration**, and **value iteration**.

Chapter 2 then discusses **multi-armed bandits**, a simpler problem that is often used as a warm-up to RL.

0.2 Notation

We will use the following notation throughout the book. This notation is inspired by Sutton and Barto and AJKS .

cite

cite

Add notation

s	A state.
a	An action.
r	A reward.
p	A probability.
π	A policy.
V	A value function.
Q	An action-value function.
A	An advantage function.
γ	A discount factor.
τ	A trajectory.
$\mathcal{S}$	A state space.
$\mathcal{A}$	An action space.

0.3 Challenges of reinforcement learning

Exploration-exploitation tradeoff. Should the agent try a new action or stick with the action that it knows is good?

Prediction. The agent might want to predict the value of a state or state-action pair.

Policy computation (control). In a complex environment, even if the dynamics are known, it can still be challenging to compute the best policy.

0.4 Resources

Inspired by the Stat 110 textbook and Stat 111 lecture notes.

This book seeks to provide an intuitive understanding before technical treatment.

Refer to AJK Sutton and Barto, online sources.

Contents

Chapter 1

Bandits

The **multi-armed bandits** (MAB) setting is a simple but powerful setting for studying the basic challenges of RL. 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.

States	Actions	Rewards
None	Finite	Stochastic

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 1.0.1: 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.

examples
ical trials,
e, etc.

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.

1.1 Multi-Armed Bandits

Remark 1.1.1: 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 take money from the player.

Let K denote the number of arms. We'll label them $1, \dots, K$ and use *superscripts* to indicate the arm index; since we seldom need to raise values to a power, this won't cause much confusion. For simplicity, we'll assume rewards are *bounded* between 0 and 1. Then each arm has an unknown reward distribution $\nu^k \in \triangle([0,1])$ with mean $\mu^k = \mathbb{E}_{r\sim\nu^k}[r]$.

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

```
# multi-armed bandits
for timestep in range(0, T):
    # Agent chooses an arm
    k = agent.choose_arm()

    # Environment generates a reward
    r = env.generate_reward(k)

    # Agent observes the reward
    agent.observe_reward(k, r)
```

Maybe switc more traditio pseudocode? Or set up so Python "inte faces" near t start?

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 1.1.1: Regret

The agent's **regret** after T timesteps is the difference between the total reward it observes and the total reward it *would* have received if it had always pulled the optimal arm:

$$\text{Regret}_T := \sum_{t=0}^{T-1} \mu^\star - \mu^{a_t} \tag{1.1}$$

Note that this depends on the *true means* of the pulled arms, *not* the observed rewards.

Often we consider the **expected regret** $\mathbb{E}[\text{Regret}_T]$, where the randomness comes from the agent's strategy.

e Big O
on in ap-
x

Ideally, we'd like to asymptotically achieve **zero regret**, i.e. $\mathbb{E}[\text{Regret}_T] = o(T)$.

1.1.1 Pure exploration (random guessing)

A trivial strategy is to always choose arms at random (i.e. "pure exploration"). What is the expected regret of this strategy?

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

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.

1.1.2 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.

```
# observed_rewards is an array of length K

# exploration phase
for k in range(K):
    observed_rewards[k] = env.generate_reward(k)
k_hat = argmax(observed_rewards)

# exploitation phase
for t in range(T - K):
    r = env.generate_reward(k_hat)
```

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 two arms, with Bernoulli reward distributions given by $\mu^1 > \mu^2$.

Let's let r^1 be the random reward from the first arm and r^2 be the random reward from the second. If $r^1 > r^2$, then we achieve zero regret. Otherwise, we achieve regret $T(\mu^1 - \mu^2)$. Thus, the expected regret is simply:

$$\begin{aligned}\mathbb{E}[\text{Regret}_T] &= \mathbb{P}(r^1 < r^2) \cdot T(\mu^1 - \mu^2) + c \\ &= (1 - \mu^1)\mu^2 \cdot T(\mu^1 - \mu^2) + c\end{aligned}$$

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

Can we do better? For one, we could reduce the variance of the reward estimates by pulling each arm *multiple times*. This is called the **explore-then-commit** strategy.

1.1.3 Explore-then-commit

Let's pull each arm N_{explore} times, and then commit to the arm with the highest observed average reward. What is the expected regret of this strategy?

```
# avg_reward is an array of length K

# exploration phase
for k in range(K):
    total = 0
    for i in range(N_explore):
        total += env.generate_reward(k)
    avg_reward[k] = total / N_explore
k_hat = argmax(avg_reward)

# exploitation phase
for t in range(T):
    r = env.generate_reward(k_hat)
```

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

Let's analyze the expected regret of this strategy by splitting it up into the exploration and exploitation phases.

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 $N_{\text{explore}}K$.

Exploitation phase. This will take a bit more effort. We'll ultimately prove that:

1. For any total time T,
2. We can choose N_{explore} such that
3. With arbitrarily high probability, the regret is sublinear.

Let $\hat{k} := \arg\max_{k \in [K]} \hat{\mu}^k$ be the arm we choose to exploit. 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?

nk to ap-
x

Let's use $\Delta^k = \hat{\mu}^k - \mu^k$ to denote how far the mean estimate for arm k is from the true mean. **Hoeffding's inequality** tells us that, for a given arm k, since the rewards from that arm are i.i.d.,

$$\mathbb{P}\left(|\Delta^k| > \sqrt{\frac{\ln(2/\delta)}{2N_{\text{explore}}}}\right) \leq \delta. \tag{1.2}$$

But note that we can't directly apply this to $\hat{k}$ since $\hat{k}$ is itself a random variable. Instead, we need to "uniform-ize" this bound across *all* the arms, i.e. bound the residual 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: The probability of error (i.e. the l.h.s. of 1.2) for a *single* arm is at most δ, so the probability that *at least one* of the arms is far from the mean is at most $K\delta$. Setting $\delta' := K\delta$ and taking the complement of both sides, we have

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

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

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

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

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

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

$$\text{Regret}_T \leq 3T^{2/3}\sqrt[3]{K \ln(2K/\delta')/2}$$

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.

1.1.4 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.

```
# epsilon-greedy
# random() samples from the uniform distribution on [0, 1]
for t in range(T):
    if random() < epsilon(t):
        # exploration
        k = random_choice(K)
    else:
        # exploitation
        # calculate averages using element-wise division
        k = argmax(total_reward / num_pulls)
    r = env.generate_reward(k)
    total_reward[k] += r
    num_pulls[k] += 1
```

Note that ϵ can vary over time. In particular we might want to gradually *decrease* ϵ as we learn more about the environment over time.

It turns out that setting $\epsilon_t = \sqrt[3]{K \ln(t)/t}$ also achieves a regret of $\widetilde{O}(t^{2/3}\sqrt[3]{K})$ (ignoring the logarithmic factors).

Insert optima epsilon and r analysis / hi probability b

In the ETC case, we had to set N_{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? In particular, what if we could explore more for arms that we're less certain about? This brings us to the **Upper Confidence Bound** (UCB) algorithm.

1.1.5 Upper Confidence Bound (UCB)

Needs revisio structure

We'll estimate *confidence intervals* for the mean of each arm, 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, we'd like to compute some upper confidence bound M_t^k for arm k at time t and then choose $a_t := \arg\max_{k\in[K]} M_t^k$. But how should we compute M_t^k?

In our regret analysis for ETC, we were able to compute this bound using Hoeffding's inequality. Hoeffding's inequality assumes that the number of samples is *fixed*, which was true in ETC. However, 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 upper confidence bound that holds *uniformly* across time and across the different 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,

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

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 $\widetilde{r}_n^k$ to be the nth sample from arm k, and $\widetilde{\mu}_n^k$ 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 $\widetilde{\mu}_n^k$.

So how can we extend our bound on $\widetilde{\mu}_n^k$ to $\hat{\mu}_t^k$? Well, we know $N_t^k \le t$ (which would be the case 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. In particular, we let $\delta' := t\delta$, giving us

$$\mathbb{P}\left(\forall n \le t, |\widetilde{\mu}_n^k - \mu^k| \le \sqrt{\frac{\ln(2t/\delta')}{2n}}\right) \ge 1 - \delta'$$

rate more on

Now we can safely set $n := N_t^k$ to achieve

$$\mathbb{P}\left(|\hat{\mu}_t^k - \mu^k| \le \sqrt{\frac{\ln(2t/\delta')}{2N_t^k}}\right) \ge 1 - \delta'.$$

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 δ' depending on how tight we want the interval to be. A smaller δ' would give us a larger yet "more confident" interval, and vice versa.

Intuitively, this 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?

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 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:

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

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

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

e include? In fact, we can do a more sophisticated analysis to show $\text{Regret}_T = \widetilde{O}(\sqrt{TK})$.

1.2 Thompson sampling

Chapter 2

Markov Decision Processes

Contents

2.1 Introduction

The field of RL studies how an agent can learn to make sequential decisions in an 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** like chess or Go, where the player takes turns with the opponent to make moves on the board.
- **Video games** like Super Mario Bros or Breakout, where the player controls a character to reach the goal.
- **Robotic control**, where the robot can move and interact with the real-world environment to complete some task.

All of these fit into the RL framework! Consider what the agent, state, and possible reward signals are in each example.

These are environments where the **state transitions**, the "rules" of the environment, only depend on the *most recent* state and action. We can formalize such environments using **Markov decision processes** (MDPs). Formally, we say that the state transitions satisfy the **Markov property**:

$$\mathbb{P}(s_{t+1} \mid s_0, a_0, \ldots, s_t, a_t) = P(s_{t+1} \mid s_t, a_t).$$

Where $P : \mathcal{S} \times \mathcal{A} \to \Delta(\mathcal{S})$ is the state transition function. We'll see that this simple assumption leads to a rich set of problems and algorithms.

MDPs are usually classified as **finite-horizon** (aka **episodic**), 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 then discuss the infinite-horizon case.

In each setting, we'll describe how to evaluate different **policies** (strategies for choosing actions) and how to compute or approximate the **optimal policy**. We'll introduce the **Bellman**

consistency condition, which allows us to analyze the whole series of interactions in terms of individual timesteps.

2.2 Finite horizon (episodic) MDPs

The components of a finite-horizon (aka **episodic**) 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** $\mu \in \Delta(\mathcal{S})$.

4. The **state transitions** (a.k.a. **dynamics**) $P : \mathcal{S} \times \mathcal{A} \to \Delta(\mathcal{S})$ 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:

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

Example 2.2.1: Tidying

Let's consider an extremely 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 $\mathcal{A} = \{\text{tidy}, \text{ignore}\}$. The room starts off tidy.

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.

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). 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	tidy	1	0	-1
orderly	ignore	0.7	0.3	1
messy	tidy	1	0	0
messy	ignore	0	1	-1

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

2.2.1 Policies

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

There are two axes along which policies can vary:

- Policies can be **deterministic** or **stochastic**. A deterministic policy maps "situations" to actions while a stochastic policy maps "situations" to *probability distributions* over actions. (The use of "situation" is clarified below.)
- Policies can be **stationary** or **history-dependent**. A stationary policy only depends on the current state, while a history-dependent policy can depend on the entire history of states, actions, and rewards. In the episodic setting, we'll also consider **time-dependent** policies that depend on the time step t, i.e. $\pi = \{\pi_0, \ldots, \pi_{H-1}\}$ where $\pi_t : \mathcal{S} \to \mathcal{A}$ or $\Delta(\mathcal{A})$ for each $t \in [H]$.

A fascinating result is that every finite-horizon MDP has an optimal policy that is time-dependent and deterministic! 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 2.2.2: Tidying policy

Here are some possible policies for the tidying example:

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

2.2.2 Trajectories

A sequence of states, actions, and rewards is called a **trajectory**:

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

(Note that sources differ as to whether to include the reward at the final time step. This is a minor detail.)

Once we've chosen a policy, we can sample trajectories by choosing actions according to the policy and observing the state transitions and rewards. That is, a policy induces a distribution ρ^π over trajectories. (We assume that μ and P are clear from context.)

Example 2.2.3: Trajectories in the tidying environment

Here is a possible trajectory for the tidying example:

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

Could any of the above policies have generated this trajectory?

Note that for a stationary policy, using the Markov property, we can specify this probability distribution in an **autoregressive** way (i.e. one timestep at a time):

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

Exercise: How would you modify this to include stochastic rewards?

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

2.2.3 Value functions

We'll use G_t to denote the cumulative reward from a given time step onwards (called the **return-to-go**):

$$G_t(\tau) := r_t + \cdots + r_{H-1}$$

(The dependence on τ will be omitted for brevity.) The main goal of RL is to find an **optimal policy** $\pi^\star$ that maximizes the expected return:

$$\pi^\star = \arg\max_\pi \mathop{\mathbb{E}}_{\tau\sim\rho^\pi}[G_0].$$

We'll need a concise way to refer to the expected return conditional on *starting in a given state at a given time*. We call this the **value function** of π at time t and denote it by

$$V_t^\pi(s) := \mathop{\mathbb{E}}_{\tau\sim\rho^\pi}[G_t \mid s_t = s]$$

Similarly, we can define the **action-value function** (aka the **Q-function**) as the expected return when starting in a given state and taking a given action:

$$Q_t^\pi(s,a) := \mathop{\mathbb{E}}_{\tau\sim\rho^\pi}[G_t \mid s_t = s, a_t = a]$$

Remark 2.2.1: Connection between value and action-value functions

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

$$V_t^\pi(s) = \mathop{\mathbb{E}}_{a\sim\pi_t(s)}[Q_t^\pi(s,a)]$$

and the action-value can be expressed in terms of the value of the following state:

$$Q_t^\pi(s,a) = r(s,a) + \mathop{\mathbb{E}}_{s'\sim P(s,a)}[V_{t+1}^\pi(s')]$$

The one-step (Bellman) consistency equation

Note that by simply considering the return as the sum of the *current* reward and the *future* return, we can describe the value function recursively (in terms of itself):

$$V_t^\pi(s) = \mathop{\mathbb{E}}_{\substack{a\sim\pi_t(s)\\ s'\sim P(s,a)}}[r(s,a) + V_{t+1}^\pi(s')] \tag{2.1}$$

This is named the **Bellman consistency equation** after **Richard Bellman** (1920–1984), who is credited with introducing dynamic programming in 1953.

Remark 2.2.2: The Bellman consistency equation for deterministic policies

Note that for deterministic policies, the Bellman consistency equation simplifies to

$$V_t^\pi(s) = r(s,\pi_t(s)) + \mathop{\mathbb{E}}_{s'\sim P(s,\pi_t(s))}[V_{t+1}^\pi(s')]$$

Remark 2.2.3: Bellman consistency for the action-value function

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

$$Q_t^\pi(s,a) = r(s,a) + \mathop{\mathbb{E}}_{\substack{s'\sim P(s,a)\\ a'\sim\pi_{t+1}(s')}}[Q_{t+1}^\pi(s',a')]$$

which can also be simplified for deterministic policies:

$$Q_t^\pi(s,a) = r(s,a) + \mathop{\mathbb{E}}_{s'\sim P(s,a)}[Q_{t+1}^\pi(s',\pi_{t+1}(s'))]$$

The Bellman operator

Fix a policy π. 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":

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

We'll call $\mathcal{J}^\pi$ the **Bellman operator** for π. Note that it's defined on any "value function" mapping states to real numbers; v doesn't necessarily have to be a well-defined value function for some policy. It also gives us a concise way to express the Bellman consistency equation:

$$V_t^\pi = \mathcal{J}^\pi(V_{t+1}^\pi)$$

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

Its critical property is that it is a **contraction mapping**. Intuitively, if we start with two "value functions" $v, u : \mathcal{S} \to \mathbb{R}$ and repeatedly apply the Bellman operator to each of them, they will get "closer and closer" at an exponential rate. We'll formalize and prove this fact later when discussing infinite-horizon MDPs and then use it to derive and analyze several useful algorithms.

2.2.4 Policy evaluation

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

Dynamic programming

The Bellman consistency equation gives us a convenient algorithm for evaluating stationary policies: it expresses the value function at time t as a function of the value function at time $t+1$. 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.

Definition 2.2.1: Dynamic programming for policy evaluation in finite-horizon MDPs

```
V[t][s] = 0 for all timesteps t (including t = H) and states s
for t = H-1, H-2, ..., 0:
    for s in S, a in A, s' in S:
        V[t][s] += pi[t][s] * P[s][a][s'] * (r[s][a] + V[t+1][s'])
```

Example 2.2.4: Tidying policy evaluation

Let's evaluate the policy from 2.2.2 that tidies iff the room is messy. We'll use the Bellman consistency equation to compute the value function at each time step.

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

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

alise code
ntions

2.2.5 Optimal policies

We've just seen how to evaluate a given policy. But how can we find the *best* policy for a given environment – the one that does better than all the others (in expectation)? Formally speaking, we call a policy $\pi^\star$ **optimal** if it does at least as well as *any* other policy π (including stochastic and history-dependent ones) in all situations:

$$V_t^{\pi^\star}(s) \geq V_t^{\pi}(s) \quad \forall s \in \mathcal{S}, t \in [H]$$

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

We mentioned earlier that every MDP has an optimal policy that is stationary and deterministic. In particular, we can construct such a policy by acting *greedily* with respect to the optimal action-value function:

$$\pi_t^\star(s) = \arg\max_a Q_t^\star(s, a)$$

Theorem 2.2.1: It is optimal to be greedy w.r.t. the optimal value function

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

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

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 $t \in [H]$.

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

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

$$[\mathcal{J}^{\hat{\pi}}(V_{t+1}^\star)](s) \geq V_t^\star(s)$$

To see this, let's expand the definition of $V^\star$:

$$
\begin{aligned}
V_t^\star(s) &= \max_{\pi \in \Pi} V_t^\pi(s) \\
&= \max_{\pi \in \Pi} \mathop{\mathbb{E}}_{a \sim \pi} \left[r(s,a) + \mathop{\mathbb{E}}_{s' \sim P(s,a)} V_{t+1}^\pi(s') \right] && \text{Bellman consistency} \\
&\leq \max_{\pi \in \Pi} \mathop{\mathbb{E}}_{a \sim \pi} \left[r(s,a) + \mathop{\mathbb{E}}_{s' \sim P(s,a)} V_{t+1}^\star(s') \right] && \text{definition of } V^\star \\
&= \max_{a} \left[r(s,a) + \mathop{\mathbb{E}}_{s' \sim P(s,a)} V_{t+1}^\star(s') \right] && \text{only depends on } \pi \text{ via } a \\
&= [\mathcal{J}^{\hat{\pi}}(V_{t+1}^\star)](s).
\end{aligned}
$$

Note that the expression $a \sim \pi$ above might depend on the past history; this isn't shown in the notation and doesn't affect our result. 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.)

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

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

Dynamic programming

Now that we've shown this greedy policy is optimal, we can work backwards in time to compute the optimal value function and optimal policy. This is called **dynamic programming**.

Theorem 2.2.2: Computing the optimal policy

We can solve for the optimal policy in an episodic MDP using **dynamic programming**.

- *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:

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

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

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

  Then $V^\star_{H-1}(s)$, 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 $t = H-2, \ldots, 0$, we set

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

Analysis

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 $O(H|\mathcal{S}|^2|\mathcal{A}|)$.

2.2.6 Summary

We've just seen the definition of a finite-horizon MDP and how to evaluate and compute the optimal policy for such an MDP. Here's a summary of the key concepts:

- A **finite-horizon** (a.k.a. **episodic**) MDP is a tuple $(\mathcal{S}, \mathcal{A}, \mu, P, r, H)$ where $\mathcal{S}$ is the state space, $\mathcal{A}$ is the action space, μ is the initial state distribution, P is the state transition function, r is the reward function, and H is the time horizon.
- A **policy** π is a strategy for choosing actions. A **stationary** policy is one that only depends

on the current state.

- A **trajectory** τ is a sequence of states, actions, and rewards. A trajectory distribution ρ^π is the distribution over trajectories induced by a policy π.
- The **return-to-go** G_t is the cumulative reward from a given time step onwards.
- The **value function** $V_t^\pi(s)$ is the expected return-to-go conditional on starting in a given state at a given time.
- The **action-value function** $Q_t^\pi(s, a)$ is the expected return-to-go conditional on starting in a given state, taking a given action, and then following the policy.
- The **Bellman consistency equations** express the value function and action-value function recursively in terms of themselves.
- **Policy evaluation** is the task of computing the value function of a given policy. We can do this by starting at the end of the time horizon and working backwards in time.
- An **optimal policy** is one that does at least as well as any other policy in all situations. We can compute the optimal policy using dynamic programming.

2.3 Infinite horizon MDPs

What happens if a trajectory is allowed to continue possibly forever? This is the setting of **infinite horizon** MDPs. We'll need to make a few adjustments to make the problem tractable.

Instead of a time horizon H, we now need a **discount factor** γ such that rewards become less valuable the further into the future they are. Formally, instead of the "undiscounted" return-to-go above (which might blow up to infinity), we work with the **discounted** return-to-go, which is well-defined (assuming the rewards are bounded):

$$G_t = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \cdots = \sum_{k=0}^{\infty} \gamma^k r_{t+k}$$

This also has a nice real-world interpretation: it's better to get a reward now than later, since you can invest it and get more rewards in the future.

The other components of the MDP remain the same:

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

We'll shift our focus to time-independent policies $\pi : \mathcal{S} \to \mathcal{A}$ (deterministic) or $\Delta(\mathcal{A})$ (stochastic). We also consider time-independent value functions $V^\pi : \mathcal{S} \to \mathbb{R}$ and $Q^\pi : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$. We need to adjust the Bellman consistency functions accordingly to account for the discounting:

$$\begin{aligned} V^\pi(s) &= \mathop{\mathbb{E}}_{\tau \sim \rho^\pi}[G_0 \mid s_0 = s] \\ &= \mathop{\mathbb{E}}_{\substack{a \sim \pi(s) \\ s' \sim P(s,a)}}[r(s,a) + \gamma V^\pi(s')] \\ Q^\pi(s,a) &= \mathop{\mathbb{E}}_{\tau \sim \rho^\pi}[G_0 \mid s_0 = s, a_0 = a] \\ &= r(s,a) + \gamma \mathop{\mathbb{E}}_{\substack{s' \sim P(s,a) \\ a' \sim \pi(s')}}[Q^\pi(s', a')] \end{aligned}$$

Here is an outline of the rest of the chapter:

1. Several algorithms rely heavily on the fact that the Bellman operator is a *contracting map*, and so it has a unique attracting fixed point.
2. We'll discuss how to evaluate policies (i.e. compute their corresponding value functions) in this new setting.
3. Then we'll discuss two iterative algorithms for computing the optimal policy: value iteration and policy iteration.

2.3.1 Contracting maps

One 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. A useful fact known as the **Banach fixed-point theorem** tells us that this procedure will converge to the true value function!

Let's make this more rigorous. How can we measure the distance between two value functions? We'll take the **supremum norm** as our distance metric:

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

We aim to show that

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

The following derivation demonstrates this:

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

Then the Banach fixed-point theorem tells us that repeatedly applying the Bellman operator will converge to the true value function exponentially:

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

where $(J^\pi)^{(t)}(v)$ denotes applying J^π to v t times. We'll use this useful fact to prove the convergence of several algorithms later on.

2.3.2 Tabular case (linear algebraic notation)

When the state and action spaces are finite and small, we can think of the value function and Q-function as *lookup tables* with each cell corresponding to the value of a state (or state-action pair). We can neatly express quantities as vectors and matrices:

$$r \in \mathbb{R}^{|\mathcal{S}|\times|\mathcal{A}|} \qquad P \in [0,1]^{(|\mathcal{S}\times\mathcal{A}|)\times|\mathcal{S}|} \qquad \rho \in [0,1]^{|\mathcal{S}|}$$
$$\pi \in [0,1]^{|\mathcal{A}|\times|\mathcal{S}|} \qquad V^\pi \in \mathbb{R}^{|\mathcal{S}|} \qquad Q^\pi \in \mathbb{R}^{|\mathcal{S}|\times|\mathcal{A}|}.$$

(Verify that these types make sense!) Note that when the policy π is deterministic, the actions can be determined from the states, and so we can chop off the action dimension for the rewards and state transitions:

$$r^\pi \in \mathbb{R}^{|\mathcal{S}|} \qquad P^\pi \in [0,1]^{|\mathcal{S}|\times|\mathcal{S}|} \qquad \rho \in [0,1]^{|\mathcal{S}|}$$
$$\pi \in \mathcal{A}^{|\mathcal{S}|} \qquad V^\pi \in \mathbb{R}^{|\mathcal{S}|} \qquad Q^\pi \in \mathbb{R}^{|\mathcal{S}|\times|\mathcal{A}|}.$$

2.3.3 Policy evaluation

Note that the previous DP technique no longer works since there is no "final timestep" to start from. We'll need another approach to policy evaluation. Once again, the Bellman consistency conditions give a convenient way to evaluate a 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 solution.

Simplified Bellman consistency equations

The Bellman consistency equations for a deterministic policy can be jointly written in this linear-algebraic notation as

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

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

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

Note that 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.) clarify this h

Iterative policy evaluation

The matrix inversion above takes roughly $O(|\mathcal{S}|^3)$ time. Can we trade off the requirement of finding the *exact* value function for a faster *approximate* algorithm?

Let's use the Bellman operator to define an iterative algorithm for computing the value function. We'll start with an initial guess $v^{(0)}$ and then iterate the Bellman operator:

$$v^{(t+1)} = \mathcal{J}^\pi(v^{(t)}) = r^\pi + \gamma P^\pi v.$$

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

Then as we showed before, by the Banach fixed-point theorem:

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

How many iterations do we need for an ϵ-accurate estimate? We can work backwards to solve for t:

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

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

$$O\left(|\mathcal{S}|^2|\mathcal{A}| \cdot \frac{\log(1/(\epsilon(1-\gamma)))}{1-\gamma}\right).$$

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

2.3.4 Optimal policies

Now let's move on to solving for the optimal policy. Once again, we can't use the DP approach from the episodic case. Instead, we'll exploit the observation that the Bellman consistency equations for the optimal value function doesn't depend on any policy:

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

To see this, recall we showed that the greedy policy w.r.t. the optimal value function is optimal; substituting this policy into the standard Bellman consistency equations gives the above expression.

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

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

Since the greedy-w.r.t.-$V^\star$ 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**.

Value iteration

Write pseudo

As the final step of the algorithm, to return an actual policy, we can simply act greedily w.r.t. the final iteration $v^{(T)}$ of our above algorithm. We must be careful, though: The value function of this greedy policy is *not* the same as $v^{(T)}$!

Formally, if $\|v^{(T)} - V^\star\|_\infty \le \epsilon$, then the greedy policy $\hat{\pi}$ satisfies $\|V^{\hat{\pi}} - V^\star\|_\infty \le \frac{2\gamma}{1-\gamma}\epsilon$, which might potentially be very large, i.e. a very loose bound!

Theorem 2.3.1: Greedy policy value degradation

We aim to show that

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

where $\hat{\pi}(s) = \arg\max_a q(s,a)$ is the greedy policy w.r.t. $q(s,a) = r(s,a) + \mathbb{E}_{s' \sim P(s,a)} v(s')$.

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

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

The second one is bounded by

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

and thus

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

So in order to 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 we need to run the algorithm for

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

Policy iteration

Can we mitigate this "greedy worsening"? Instead

Theorem 2.3.2: Policy Iteration

Remember, for now we're only considering policies that are *stationary and deterministic*. There's $|\mathcal{S}|^{|\mathcal{A}|}$ of these, so let's start off by choosing one at random. Let's call this initial policy π^0, using the superscript to indicate the time step.

Now for $t = 0, 1, \ldots$, we perform the following:

1. *Policy Evaluation*: First use the exact policy evaluation algorithm from earlier to calculate $V^{\pi^t}(s)$ for all states s. Then use this to calculate the state-action values:

$$Q^{\pi^t}(s,a) = r(s,a) + \gamma \sum_{s'} P(s' \mid s,a) V^{\pi^t}(s')$$

2. *Policy Improvement*: Update the policy so that, at each state, it chooses the action with the highest action-value (according to the current iterate):

$$\pi^{t+1}(s) = \arg\max_a Q^{\pi^t}(s, a)$$

In other words, we're setting it to act greedily with respect to the current Q-function.

What's the computational complexity of this?

Let's analyse this algorithm.

finish policy tion

Chapter 3

Linear Quadratic Regulators

3.1 Motivation

Have you ever tried balancing a pen upright on your palm? If not, try it! It's a lot harder than seems. Unlike the cases we studied the previous chapter, the state space and action space aren't *finite*, or even *discrete*. Instead, they are *continuous* and *uncountably infinite*. In addition, the state transitions governing the system – that is, the laws of physics – are nonlinear and complex.

We'll keep this motivating example in mind throughout the chapter, although reframing it in terms of the following classic *control problem:*

Example 3.1.1: CartPole

Consider a pole balanced on a cart. The state s consists of just four continuous values:

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* [a] the cart by applying a horizontal force a.

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

[a] *Controls* are the continuous analogue to *actions* in the discrete setting. In control theory, the state and controls are typically denoted as x and u, but we'll stick with the s and a notation to highlight the similarity with the discrete case.

Beyond this simple scenario, there are many real-world examples that involve continuous control:

- **Robotics.** Autonomous driving; Controlling a drone's position; Automation in warehouses and manufacturing; Humanoid robots with joints.

- **Temperature.** Controlling the temperature in a room; Keeping greenhouses warm; Understanding weather patterns.

- **Games.** Sports; MMORPGs (Massively Multiplayer Online Role-Playing Games); Board games.

- **Finance.** Stock trading; Portfolio management; Risk management.

How can we teach computers to solve these kinds of problems?

In the last chapter, we developed efficient dynamic programming algorithms (*value iteration* and *policy iteration*) for calculating $V^\star$ and $\pi^\star$ in the finite setting. In this chapter, we'll derive similar results in the continuous case by imposing some additional structure on the problem.

Note that we're still assuming that the entire environment is *known* – that is, we understand 'how the world works'. We'll get to the unknown case in the next chapter.

3.2 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 γ or time horizon T. What are the equivalents in the control setting?

- The state and action spaces are *continuous* rather than finite. That is, $\mathcal{S} = \mathbb{R}^{n_s}$ and $\mathcal{A} = \mathbb{R}^{n_a}$, where n_s and n_a are 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_t. We denote these dynamics as the function f_t, such that $s_{t+1} = f_t(s_t, a_t, w_t)$. Of course, we can simplify to cases where the dynamics are *deterministic/noise-free* (no w_t term) or are *stationary/time-homogeneous* (the same function f across timesteps).

- Instead of a reward function, it's more intuitive to consider a **cost function** $c_t : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$ that describes *how far away* we are from our **goal 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.

- We seek to minimize the *undiscounted* cost with a *time horizon* T. Note that we end an episode at s_T – there is no a_T, and so we denote the cost for the final state as $c_T(s_T)$.

With all of these components, we can now formulate the **optimal control problem:** *find a time-dependent policy to minimize the expected undiscounted cost over T timesteps.*

Definition 3.2.1: Optimal control problem

$$\begin{aligned} \min_{\pi_0,\ldots,\pi_{T-1}:\mathcal{S}\to\mathcal{A}} \quad & \mathop{\mathbb{E}}_{s_0,w_t}\left[\left(\sum_{t=0}^{T-1} c_t(s_t,a_t)\right) + c_T(s_T)\right] \\ \text{where} \quad & s_{t+1} = f_t(s_t,a_t,w_t), \\ & a_t = \pi_t(s_t) \\ & s_0 \sim \mu_0 \\ & w_t \sim \text{noise} \end{aligned} \tag{3.1}$$

3.2.1 Discretization

How does this relate to the finite horizon case? If s_t and a_t were discrete, then we'd be able to work backwards using the DP algorithms we saw before. As a matter of fact, let's consider what happens if we *discretize* the problem. For intuition, suppose $n_s = n_a = 1$ (that is, states and actions are real numbers). To make $\mathcal{S}$ and $\mathcal{A}$ discrete, let's choose some small positive ϵ, and simply round states and actions to the nearest multiple of ϵ. For example, if $\epsilon = 0.01$, then we're just rounding s and a to two decimal spaces. [1] If both these state and action spaces can be bounded, then the resulting sets are actually finite, so now we can use our previous tools for MDPs.

But is this actually a feasible solution? Even if our $\mathcal{S}$ and $\mathcal{A}$ are finite, the existing algorithms might take unfeasibly long to complete. Suppose our state and action spaces are bounded by some constants $\max_{s\in\mathcal{S}} \|s\| \le B_s$ and $\max_{a\in\mathcal{A}} \|a\| \le B_a$. Then using our rounding method, we must divide *each dimension* into intervals of length ε, resulting in $(B_s/\varepsilon)^{n_s}$ and $(B_a/\varepsilon)^{n_a}$ total points. To get a sense of how quickly this grows, let's consider $\varepsilon = 0.01, n_s = n_a = 10$. Then the number of elements in our transition matrix is $|\mathcal{S}|^2|\mathcal{A}| = (100^{10})^2(100^{10}) = 10^{60}$! Try finding a computer that'll fit that in memory! (For reference, 32 GB of memory can store 10^9 32-bit floating point numbers.)

So as we've seen, discretizing the problem isn't a feasible solution as soon as our action and state spaces are even moderately high-dimensional. How can we do better?

Note that by discretizing the state and action spaces, we implicitly assumed that rounding each state or action vector by some tiny amount ε wouldn't change the behavior much; namely, that the functions involved were relatively *continuous*. Can we use this continuous structure in other ways? This brings us to the topic of **Linear Quadratic Regulators,** a widely used and studied tool in control theory.

[1] Formally, we can consider an ϵ-net over the original continuous space. Let V be some normed space. A subset $V_\epsilon \subseteq V$ is called an ϵ-net if for all $v \in V$, there exists a $v_\epsilon \in V_\epsilon$ such that $\|v - v_\epsilon\| \le \epsilon$. The rounding example given is technically a 0.005-net.

3.3 The Linear Quadratic Regulator Problem

The optimal control problem stated above seems very difficult to solve. The cost function might not be convex, making optimization difficult, and the state transitions might be very complex, making it difficult to satisfy the constraints. Is there a relevant simplification that we can analyze?

We'll show that a natural structure to impose is *linear dynamics* and a *quadratic cost function* (in both arguments). This is called the **linear quadratic regulator** (LQR) model, and is a popular tool in control theory. In fact, some people even design systems to be linear in order to use results from LQR!

Why are these assumptions useful? As we'll see later in the chapter, it lets us *locally approximate* nonlinear dynamics and cost functions using their *Taylor approximations* (up to first and second order respectively). We'll also find that even for more complex setups, we can generalize the algorithms for LQR to get surprisingly good solutions.

Definition 3.3.1: The linear quadratic regulator

Linear, time-homogeneous dynamics:

$$s_{t+1} = f(s_t, a_t, w_t) = As_t + Ba_t + w_t$$

Quadratic, time-homogeneous cost function: [a]

$$c(s_t, a_t) = \begin{cases} s_t^\top Q s_t + a_t^\top R a_t & t < T \\ s_T^\top Q s_T & t = T \end{cases}$$

We want c to be a convex function (easy to optimize) in both s_t and a_t, so we'll set Q and R to both be symmetric. [b]

Intuitively, the cost function punishes states and actions that are far away from the origin (i.e. both the state and action are zero vectors). More generally, we'll want to replace the origin with a *goal* state and action $(s^\star, a^\star)$. This can easily be done by replacing s_t with $(s_t - s^\star)$ and a_t with $(a_t - a^\star)$ in the expression above.

Isotropic Gaussian noise:

$$w_t \sim \mathcal{N}(0, \sigma^2 I)$$

Putting everything together, the optimization problem we want to solve is:

$$\begin{aligned} \min_{\pi_0,\ldots,\pi_{T-1}:\mathcal{S}\to\mathcal{A}} \quad & \mathbb{E}\left[\left(\sum_{t=0}^{T-1} s_t^\top Q s_t + a_t^\top R a_t\right) + s_T^\top Q s_T\right] \\ \text{where} \quad & s_{t+1} = As_t + Ba_t + w_t \\ & a_t = \pi_t(s_t) \\ & w_t \sim \mathcal{N}(0, \sigma^2 I) \\ & s_0 \sim \mu_0. \end{aligned}$$

[a] For some intuition into this expression, consider the simple case where a_t and s_t are scalars (and so are Q and R), so $c(s_t, a_t) = Qs_t^2 + Ra_t^2$. If this notation is unfamiliar to you, we recommend this tutorial on quadratic forms from Khan Academy!

[b] Note that it suffices for them to be positive definite, but symmetry makes some later expressions much nicer.

So how do we go about analyzing this system? A good first step might be to introduce *value functions,* analogous to those in the previous chapter, to reason about the behavior of the system over the time horizon.

Definition 3.3.2: Value functions for LQR

Given a policy $\pi = (\pi_0, \dots, \pi_{t-1})$, we can define the value function $V_t^\pi : \mathcal{S} \to \mathbb{R}$ as

$$\begin{aligned} V_t^\pi(s) &= \mathbb{E}\left[\left(\sum_{i=t}^{T-1} c(s_i, a_i)\right) + c(s_T)\right] \\ &= \mathbb{E}\left[\left(\sum_{i=t}^{T-1} s_i^\top Q s_i + a_i^\top R a_i\right) + s_T^\top Q s_T\right] \\ \text{conditional on} \quad s_t &= s \\ a_i &= \pi_i(s_i) \quad \forall i \geq t. \end{aligned}$$

The expression inside the expectation is called the **cost-to-go**, since it's just the total cost starting from timestep t.

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

$$\begin{aligned} Q_t^\pi(s, a) &= \mathbb{E}\left[\left(\sum_{i=t}^{T-1} c(s_i, a_i)\right) + c(s_T)\right] \\ &= \mathbb{E}\left[\left(\sum_{i=t}^{T-1} s_i^\top Q s_i + a_i^\top R a_i\right) + s_T^\top Q s_T\right] \\ \text{conditional on} \quad (s_t, a_t) &= (s, a) \\ a_i &= \pi_i(s_i) \quad \forall i > t \end{aligned}$$

As in the previous chapter, these will be instrumental in constructing optimal policy π via dynamic programming.

3.4 Optimality and the Ricatti Equation

In this section, we'll derive the optimal policy in the LQR setting. We'll do this through induction, which directly translates into a recursive dynamic programming algorithm for actually calculating the optimal policy. Along the way, we'll prove that the optimal value function is *quadratic,* and

that the optimal policy is a *linear function* of the state.

Definition 3.4.1: Optimal value functions for LQR

The **optimal value function** is the one that, at any time, in any state, achieves *minimum cost* across all policies:

$$\begin{aligned} V_t^\star(s) &= \min_{\pi_t,\dots,\pi_{T-1}:\mathcal{S}\to\mathcal{A}} V_t^\pi(s) \\ &= \min_{\pi_t,\dots,\pi_{T-1}} \mathbb{E}\left[\left(\sum_{i=t}^{T-1} s_t^\top Q s_t + a_t^\top R a_t\right) + s_T^\top Q s_T\right] \\ \text{conditional on} \quad & a_i = \pi_i(s_i) \quad \forall i \geq t \\ & s_t = s \end{aligned}$$

Theorem 3.4.1: Optimal value function in LQR is quadratic and convex

$$V_t^\star(s) = s^\top P_t s + p_t$$

for some time-dependent $P_t \in \mathbb{R}^{n_s \times n_s}$ and $p_t \in \mathbb{R}^{n_s}$ where P_t is symmetric. Note that there is no linear term.

Theorem 3.4.2: Optimal policy in LQR is linear

$$\pi_t^\star(s) = -K_t s$$

for some $K_t \in \mathbb{R}^{k \times d}$. (The negative is due to convention.)

We'll derive both of these theorems simultaneously via an *inductive* proof. We'll start from the final timestep T as our base case. Then, we'll show that if the theorems hold at time $t+1$, they must hold for time t. This is called the *inductive hypothesis,* and by proving it, the theorems will 'ripple down' to all earlier timesteps, and we'll have shown that these theorems are always true. As an additional bonus, as mentioned above, our proof will naturally produce a DP algorithm that allows us to calculate the optimal value function and policy.

Base case: $V_T^\star(s) = s^\top P_T s + p_T$. At the final timestep, there are no possible actions to take, and so $V_T^\star(s) = c(s) = s^\top Q s$. Thus $P_T = Q$ and p_T is the zero vector.

Inductive hypothesis: We seek to show that the inductive step holds for both theorems: If $V_{t+1}^\star(s)$ is quadratic and convex, then $V_t^\star(s)$ must also be quadratic and convex, and $\pi_t^\star(s)$ must be linear. We'll break this down into the following steps:

Step 1. Show that $Q_t^\star(s,a)$ is quadratic and convex (in both s and a).

Step 2. Derive the optimal policy $\pi_t^\star(s) = \arg\min_a Q_t^\star(s,a)$ and show that it's linear.

Step 3. Show that $V_t^\star(s)$ is quadratic and convex.

This is essentially the same proof that we wrote in the finite-horizon MDP setting, except

now the state and action are *continuous* instead of finite.

We first assume our theorems are true at time $t+1$. That is,

$$V^\star_{t+1}(s) = s^\top P_{t+1} s + p_{t+1} \quad \text{for all states} \quad s \in \mathcal{S}.$$

Step 1. We'll start off by demonstrating that $Q^\star_t(s)$ is quadratic and convex. Recall that the definition of $Q^\star_t : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$ is

$$Q^\star_t(s,a) = c(s,a) + \mathop{\mathbb{E}}_{s' \sim f(s,a,w_{t+1})} V^\star_{t+1}(s').$$

We know $c(s,a) := s^\top Q s + a^\top R a$. Let's consider the average value over the next timestep. The only randomness in the dynamics comes from the noise w_{t+1}, so we can write out this expected value as:

$$\begin{aligned}
&\mathop{\mathbb{E}}_{s' \sim f(s,a,w_{t+1})} V^\star_{t+1}(s') \\
&= \mathop{\mathbb{E}}_{w_{t+1} \sim \mathcal{N}(0,\sigma^2 I)} V^\star_{t+1}(As + Ba + w_{t+1}) && \text{definition of } f \\
&= \mathop{\mathbb{E}}_{w_{t+1}} \left[(As + Ba + w_{t+1})^\top P_{t+1}(As + Ba + w_{t+1}) + p_{t+1}\right]. && \text{inductive hypothesis}
\end{aligned}$$

Summing and combining like terms, we get

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

Note that the terms that are linear in w_t 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

$$\mathop{\mathbb{E}}_{w_{t+1}} \left[w_{t+1}^\top P_{t+1} w_{t+1}\right] = \sum_{i=1}^{d} \sum_{j=1}^{d} (P_{t+1})_{ij} \mathop{\mathbb{E}}_{w_{t+1}} \left[(w_{t+1})_i (w_{t+1})_j\right].$$

When dealing with these *quadratic forms,* it's often helpful to consider the terms on the diagonal ($i = j$) separately from those off the diagonal. On the diagonal, the expectation becomes

$$(P_{t+1})_{ii} \, \mathbb{E}(w_{t+1})_i^2 = (P_{t+1})_{ii} \operatorname{Var}\left((w_{t+1})_i\right) = \sigma^2 (P_{t+1})_{ii}.$$

Off the diagonal, since the elements of w_{t+1} are independent, the expectation factors, and since each element has mean zero, the term disappears: $(P_{t+1})_{ij} \, \mathbb{E}(w_{t+1})_i \, \mathbb{E}(w_{t+1})_j = 0$. Thus, the only terms left are the ones on the diagonal, so the sum of these can be expressed as the trace of $\sigma^2 P_{t+1}$:

$$\mathop{\mathbb{E}}_{w_{t+1}} w_{t+1}^\top P_{t+1} w_{t+1} = \operatorname{Tr}(\sigma^2 P_{t+1}).$$

Substituting this back into the expression for $Q^\star_t$, we have:

$$\boxed{\begin{aligned} Q_t^\star(s,a) = s^\top(Q + A^\top P_{t+1} A)s + a^\top(R + B^\top P_{t+1} B)a + 2s^\top A^\top P_{t+1} B a \\ + \operatorname{Tr}(\sigma^2 P_{t+1}) + p_{t+1}. \end{aligned}} \tag{3.2}$$

As we hoped, this expression is quadratic in s and a. Furthermore, we'd like to show that it's also convex w.r.t. a in order to make the optimization much easier. This is fairly straightforward:

Theorem 3.4.3: $Q_t^\star$ is convex

Consider the part of Equation 3.2 that is quadratic in a, namely $a^\top(R+B^\top P_{t+1}B)a$. Then $Q_t^\star$ is convex w.r.t. a if $R + B^\top P_{t+1} B$ is positive definite (PD).

It suffices to show that it is symmetric, which we do as follows. Recall that in our definition of LQR, we assumed that R is symmetric (see Definition 3.3.1). Also note that since P_{t+1} is symmetric (by the inductive hypothesis), so too must be $B^\top P_{t+1} B$. (If this isn't clear, try proving it as an exercise!) Since the sum of two symmetric matrices is also symmetric, we have that $R + B^\top P_{t+1} B$ is symmetric, and so $Q_t^\star$ is convex w.r.t. a. A similar proof shows that $Q + A^\top P_{t+1} A$ is symmetric, and so $Q_t^\star$ is also convex in s.

Step 2. Now let's move on to the next part of the next part of proving the inductive hypothesis: showing that $\pi_t^\star(s) = \arg\min_a Q_t^\star(s,a)$ is linear. Since $Q_t^\star$ is convex, finding its minimum over a is easy: we can just take the gradient w.r.t. a and set it to zero. First, we calculate the gradient:

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

Setting this to zero, we get

$$\begin{aligned} 0 &= (R + B^\top P_{t+1} B)a + B^\top P_{t+1} A s \\ \pi_t^\star(s) := a &= -(R + B^\top P_{t+1} B)^{-1} B^\top P_{t+1} A s \\ &= -K_t s, \end{aligned} \tag{3.3}$$

where $K_t = (R + B^\top P_{t+1} B)^{-1} B^\top P_{t+1} A$.

Note that this optimal policy has an interesting property: in addition to being independent of the starting distribution μ_0 (which also happened for our finite-horizon MDP solution), it's also fully deterministic and isn't affected by noise! (Compare this with the discrete MDP case, where calculating our optimal policy required taking an expectation over the state transitions.)

Step 3. To complete our inductive proof, we must show that the inductive hypothesis is true at time t; that is, we must prove that $V_t^\star(s)$ is quadratic. Using the identity $V_t^\star(s) = Q_t^\star(s, \pi^\star(s))$, we have:

$$\begin{aligned} V_t^\star(s) &= Q_t^\star(s, \pi^\star(s)) \\ &= s^\top(Q + A^\top P_{t+1} A)s + (-K_t s)^\top (R + B^\top P_{t+1} B)(-K_t s) + 2s^\top A^\top P_{t+1} B(-K_t s) \\ &\quad + \operatorname{Tr}(\sigma^2 P_{t+1}) + p_{t+1} \end{aligned}$$

Note that w.r.t. s, this is the sum of a quadratic term and a constant, which is exactly what we were aiming for!

To conclude our proof, let's concretely specify the values of P_t and p_t. The constant term is clearly $p_t = \mathrm{Tr}(\sigma^2 P_{t+1}) + p_{t+1}$. We can simplify the quadratic term by substituting in K_t. Notice that when we do this, the $(R + B^\top P_{t+1} B)$ term in the expression is cancelled out by its inverse, and the remaining terms combine to give what is known as the *Riccati equation:*

Theorem 3.4.4: Riccati equation

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

There are several nice things to note about the Riccati equation:

1. It's defined **recursively.** Given P_T, the dynamics defined by A and B, and the state coefficients Q, we can recursively calculate P_t across all timesteps.

2. P_t often appears in calculations surrounding optimality, such as $V_t^\star, Q_t^\star$, and $\pi_t^\star$.

3. Together with A, B, and the action coefficients R, it fully defines the optimal policy.

Now we've shown that $V_t^\star(s) = s^\top P_t s + p_t$, which is quadratic and convex, and this concludes our proof. ■

In summary, we just demonstrated that:

- The optimal value function $V_t^\star$ is convex at all t.
- The optimal Q-function $Q_t^\star$ is convex (in both arguments) at all t.
- The optimal policy $\pi_t^\star$ is linear at all t.
- All of these quantities can be calculated using a symmetric matrix P_t for each timestep, which can be defined recursively using the Riccati equation.

Before we move on to some extensions of LQR, let's consider how the state at time t behaves when we act according to this optimal policy.

3.4.1 Expected state at time t

Suppose you're about to go to bed, and the thermostat in your room is controlled by an optimal policy under LQR. A very reasonable question to ask would be: what's the expected state (i.e. temperature) of the room at a given time t? Certainly you don't want to freeze or boil overnight!

To answer this question, let's first express the state at time t in a cleaner way in terms of the

history. Note that having linear dynamics makes it easy to expand terms backwards in time:

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

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

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

If we choose actions according to our optimal policy, this becomes

$$\mathbb{E}[s_t \mid s_0, a_t = -K_t s_t] = \left(\prod_{i=0}^{t-1} (A - BK_i) \right) s_0.$$

This introdces the quantity $A - BK_i$, which shows up frequently in control theory. For example, one important question is: will s_t remain bounded, or will it go to infinity as time goes on? To answer this, let's imagine that these K_is are equal (call this matrix K). Then the expression above becomes $(A - BK)^t s_0$. 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

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

By then, your room is *definitely* on fire! Otherwise, if $|\lambda_{\max}| < 1$, then it's impossible for your original state to explode as dramatically (assuming it's properly normalized).

We've now formulated an optimal solution for the typical, time-homogeneous case of LQR, and considered the expected state under the optimal policy. However, this simple case is insufficient for more complex tasks. In the following sections, we'll consider some motivating examples, and extensions of LQR where some assumptions are relaxed.

3.5 Extensions

In this section, we'll consider settings 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.

3.5.1 Time-dependent dynamics and cost function

So far, we've considered the *time-homogeneous* case, where the dynamics and cost function stay the same at every timestep. However, this might not always be the case. For example, if we want to preserve the temperature in a greenhouse, the outside forces are going to change depending on the time of day. As another example, in many sports or video games, the rules and scoring system might change during overtime. 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. (As an exercise, walk through the derivation and verify this claim!)

The modified problem is now defined as follows:

$$\begin{aligned} \arg \min_{\pi_0,\dots,\pi_{T-1}:\mathcal{S}\to\mathcal{A}} \quad & \mathbb{E}\left[\left(\sum_{t=0}^{T-1}(s_t^\top Q_t s_t) + a_t^\top R_t a_t\right) + s_T^\top Q_T s_T\right] \\ \text{where} \quad & s_{t+1} = f_t(s_t, a_t, w_t) = A_t s_t + B_t a_t + w_t \\ & s_0 \sim \mu_0 \\ & a_t = \pi_t(s_t) \\ & w_t \sim \mathcal{N}(0, \sigma^2 I). \end{aligned}$$

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

Theorem 3.5.1: Time-dependent Riccati Equation

$$P_t = Q_t + A_t^\top P_{t+1} A_t - A_t^\top P_{t+1} B_t (R_t + B_t^\top P_{t+1} B_t)^{-1} B_t^\top P_{t+1} A_t.$$

Note that this is just the time-homogeneous Riccati equation (Theorem 3.4.4), but with the time index added to each of the relevant matrices.

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

3.5.2 More general quadratic cost functions

Our original cost function had only second-order terms w.r.t. the state and action. 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_t for the cross term, linear coefficients q_t and r_t for the state and action respectively, and a constant term c_t:

$$c_t(s_t, a_t) = (s_t^\top Q_t s_t + s_t^\top M_t a_t + a_t^\top R_t a_t) + (s_t^\top q_t + a_t^\top r_t) + c_t. \tag{3.4}$$

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

$$s_{t+1} = f_t(s_t, a_t, w_t) = A_t s_t + B_t a_t + v_t + w_t.$$

The derivation of the optimal solution in this case will be left as a homework exercise.

3.5.3 Tracking a predefined trajectory

So far, we've been trying to get the robot to stay as close as possible to the origin, or more generally a goal state-action pair $(s^\star, a^\star)$. However, consider applying LQR to autonomous driving. Now, we want the desired state to change over time, instead of remaining in one location. Otherwise, it wouldn't be a very useful vehicle! In these cases, we want the robot to follow a predefined *trajectory* of states and actions $(s_t^\star, a_t^\star)_{t=0}^{T-1}$. To do this, we'll modify the cost function accordingly:

$$c_t(s_t, a_t) = (s_t - s_t^\star)^\top Q(s_t - s_t^\star) + (a_t - a_t^\star)^\top R(a_t - a_t^\star).$$

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 we discussed above (Equation 3.4):

$$M_t = 0, \qquad q_t = -2Qs_t^\star, \qquad r_t = -2Ra_t^\star, \qquad c_t = (s_t^\star)^\top Q(s_t^\star) + (a_t^\star)^\top R(a_t^\star).$$

3.6 The infinite-horizon setting

Another assumption we've made is that the task has a *finite horizon* T. How about tasks that might matter indefinitely, where we want to minimize the expected cost over *all future timesteps?* Consider, for example, controlling the long-term value of a portfolio, or managing acidity levels in a lake.

In the previous chapter, we dealt with such **infinite-horizon** cases by *discounting* future rewards. This time, we'll take a different approach by considering the limit of a *finite*-horizon task as $T \to \infty$. As it turns out, our derivation is exactly analogous to value iteration from the previous chapter! However, here the structure is nice enough that we don't need a discount factor γ to deal with limits analytically. (Note that we must normalize by $1/T$ to keep the total cost bounded.)

In the discounted case, analogously to taking the limit as $T \to \infty$, we consider the limit as the discount factor γ approaches 1. This is because as $\gamma \to 1$, time discounting becomes less and less important – just like the horizon T vanishing into the distance – and we're left with the undiscounted case. (Note that for the total reward to remain bounded, we must normalize the sum by $1 - \gamma$.) Let's consider value iteration in this setting, which uses the Bellman operator $\mathcal{J}$ to update V:

$$V_{t+1}(s) = \lim_{\gamma \to 1} (\mathcal{J} V_t)(s) = \lim_{\gamma \to 1} \max_a \left(r(s,a) + \gamma \mathop{\mathbb{E}}_{s' \sim P(s,a)} V_t(s') \right).$$

This is exactly analogous to the *Riccati equation* (3.4.4)! Instead of thinking of P_{t+1} as defining the value function for the *next timestep,* though, we think of it as the *next version* of the value function in an iterative algorithm. Then both of these algorithms are doing the same thing: iteratively refining the value function by acting greedily w.r.t. the current iteration.

By going through the same derivation in section 3.4, we'll see that P_t (defined recursively by the Riccati equation) converges to a fixed value P. For an intuitive perspective on why this happens, let's suppose you have an upcoming project deadline. When it's still a few months away, you might not pay much attention to it, and behave as you normally do. But as the deadline gets closer and closer, suddenly the horizon becomes more and more relevant, and you'll spend more time thinking about it. The infinite-horizon case is just where the deadline is infinitely far away, so you can just behave "like normal" all the time!

	Control	**Finite MDP**
State and action spaces	Continuous	Finite
Optimization problem	Minimize finite-horizon undiscounted cost	Maximize infinite-horizon discounted reward *or* finite-horizon discounted reward
Approaching the infinite-horizon, undiscounted setting	$\lim_{T\to\infty} \frac{1}{T} \mathbb{E}\left(\sum_{t=0}^{T-1} c(s_t, a_t)\right)$	$\lim_{\gamma\to 1}(1-\gamma)\,\mathbb{E}\sum_{t=0}^{\infty} \gamma^t r(s_t, a_t)$
Iterative algorithm for optimal value	Riccati equations $P \leftarrow Q + A^\top PA - A^\top PB(R + B^\top PB)^{-1}B^\top PA$	Value iteration $V(s) \leftarrow \max_a \left[r(s,a) + \gamma\,\mathbb{E}_{s'\sim P(s,a)}\,V(s')\right]$

Figure 3.1: A comparison between the continuous control and finite MDP settings.

3.7 Approximating nonlinear dynamics

As its name might suggest, LQR works best when the dynamics are linear and the cost function is quadratic. In these settings, we've shown a way to analytically derive the optimal policy. However, let's return to the CartPole example from the start of the chapter (Example 3.1.1). The dynamics (physics) aren't linear, and we might also want to specify a cost function that's more complex than just quadratic.

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

Definition 3.7.1: Nonlinear control problem

$$\begin{aligned} \min_{\pi_0,\ldots,\pi_{T-1}:\mathcal{S}\to\mathcal{A}} \quad & \mathop{\mathbb{E}}_{s_0}\left[\sum_{t=0}^{T-1} c(s_t, a_t)\right] \\ \text{where} \quad & s_{t+1} = f(s_t, a_t) \\ & a_t = \pi_t(s_t) \\ & s_0 \sim \mu_0 \\ & c(s, a) = d(s, s^\star) + d(a, a^\star). \end{aligned}$$

Here, d denotes some general measure of distance to the goal state and action $(s^\star, a^\star)$.

This is now only slightly simplified from the general optimal control problem (see 3.2.1). 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. How can we adapt LQR to this more general nonlinear case?

3.7.1 Local linearization

As we briefly mentioned in the introduction, part of the reason we designed the LQR problem the way we did was because we can take any *locally continuous* function, and approximate it using a Taylor expansion of low-order polynomials. By taking a linear approximation of f and a quadratic approximation of c, we're back to the regime of LQR with these derived matrices in terms of their gradients and Hessians accordingly.

Assumptions. This approach assumes that f is differentiable and that c is twice-differentiable, both around $(s^\star, a^\star)$. Additionally, since a Taylor expansion generally gets less and less accurate the further you stray from the point of expansion, this means that we also assume that all states are close to the optimal state $s^\star$, and that we can stay close using actions that are close to $a^\star$. If this seems like a strong set of restrictions, it is! But we'll save this discussion for the next section.

If you're unfamiliar with Taylor expansions, we recommend taking a calculus course; we'll use them here without much further introduction. 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:

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

where the gradients and Hessians are defined as

$$(\nabla_s c(s,a))_i = \frac{dc(s,a)}{ds_i}, \quad i \le n_s \qquad (\nabla_a c(s,a))_i = \frac{dc(s,a)}{da_i}, \quad i \le n_a$$

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

$$(\nabla_{sa} c(s,a))_{ij} = \frac{d^2c(s,a)}{ds_i da_j}. \quad i \le n_s, j \le n_a$$

We note that this cost can be expressed in the general quadratic form seen in Equation 3.4. We leave it as an exercise to derive the corresponding matrices and vectors Q, R, M, q, r, c. 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 definition of derivative, and see how the function changes as we add or subtract a tiny δ to the input.

However, simply taking the second-order approximation of the cost function is insufficient; we also need to ensure that it is locally convex, that is, the Hessians $\nabla_{ss} c(s^\star, a^\star)$ and $\nabla_{aa} c(s^\star, a^\star)$ are positive definite.

Local convexification. Recall that an equivalent definition of positive definite matrices is that all of their eigenvalues are positive. One way to naively *force* $\nabla_{ss} c(s^\star, a^\star)$ and $\nabla_{aa} c(s^\star, a^\star)$ to be positive definite is to simply remove any negative eigenvalues. Note that both of these Hessians are symmetric, and so they can be decomposed in terms of their eigenbasis. Thus by removing their eigenvalues and adding a small 'lower bound' to the eigenvalues (so that the surface will have some minimum amount of curvature), we obtain

$$\nabla_{ss} c(s^\star, a^\star) = \sum_{i=1}^{n_s} \sigma_i u_i u_i^\top$$

$$\nabla_{ss} \widetilde{c}(s^\star, a^\star) = \sum_{\substack{i=1 \\ \sigma_i > 0}}^{n_s} \sigma_i u_i u_i^\top + \lambda I.$$

We can use a similar approach to convexify $\nabla_{aa} \widetilde{c}(s^\star, a^\star)$. Now that we have a convex quadratic approximation to the cost function, and a linear approximation to the state transitions, we can simply apply the time-homogenous LQR methods we derived before.

But what happens when our assumptions break down, namely when we enter states far away from $s^\star$ and want to use actions far from $a^\star$? As we mentioned above, our Taylor approximation will typically become less accurate. To address this, we'll need to do a Taylor approximation around *different points at each time step.*

3.7.2 Iterative LQR

Iterative LQR is a way to resolve the issues with local linearization. The key idea is to linearize around different points at each timestep, making creating a time-dependent approximation of the dynamics. We'll break it into a few steps:

Step 1. Form an LQR around the current candidate trajectory $(\bar{s}_t^i, \bar{a}_t^i)_{t=0}^{T-1}$ using local approximation.

Step 2. Apply the solution to time-dependent LQR from subsection 3.5.1 to obtain an optimal policy π^i.

Step 3. Generate a new trajectory $(\widetilde{a}_t)_{t=0}^{T-1}$ using π^i.

Step 4. Compute a better candidate trajectory $\mathbf{a}^{i+1}$ by interpolating between $\mathbf{a}^i$ and $\widetilde{\mathbf{a}}$.

Now the question becomes: How do we choose the best *waypoints* $(\bar{s}_t, \bar{a}_t)$ to get the best approximation?

We'll once again use an iterative approach, similarly to value iteration, where at each step we update the waypoints *greedily* w.r.t. the current iteration. Let's use a superscript to denote the iteration of the algorithm. We can start off by initializing some (bad) sequence of waypoints $\bar{a}_0^0, \dots, \bar{a}_{T-1}^0$ by some approximate method such as local linearization. Applying these actions at their respective timesteps gives a sample trajectory

$$\bar{s}_0, \bar{a}_0, \bar{s}_1, \bar{a}_1, \dots, \bar{s}_{T-1}, \bar{a}_{T-1} \quad \text{where} \quad \bar{s}_0^0 = \bar{s}_0 = \mathop{\mathbb{E}}_{s_0 \sim \mu_0}[s_0], \quad \bar{s}_{t+1}^0 = f(\bar{s}_t^0, \bar{a}_t^0).$$

Now, at each timestep t, we linearize f and quadratize (and convexify) c around the point generated in the sample trajectory, using the same Taylor expansion techniques we saw in the previous section:

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

Now let's use the time-dependent LQR solution to compute an optimal policy $\pi_0^i, \dots, \pi_{T-1}^i$. We can then generate a new sample trajectory by taking actions according to this optimal policy:

$$\bar{s}_0^{i+1} = \bar{s}_0, \qquad \widetilde{a}_t = \pi_t^i(\bar{s}_t^{i+1}), \qquad \bar{s}_{t+1}^{i+1} = f(\bar{s}_t^{i+1}, \widetilde{a}_t).$$

Note that the states are drawn by sampling from the *true* dynamics, and also that we've denoted these actions as $\widetilde{a}_t$ and aren't directly using them for the next iteration a_t^{i+1}. Rather, we want

to interpolate between them and the actions from the previous iteration $\bar{a}_0^i, \ldots, \bar{a}_{T-1}^i$. 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:

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

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

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

3.8 Programming and Implementation

Not sure how much to include here yet. WIP. Walk through a basic Python solution to OpenAI Gym and CartPole? (Or save this for homework?)

3.9 Exercises

1. Consider a cleaning robot with one wheel on each side of its body. Your pet has made a mess nearby, and you want to steer the robot to go clean it up. Let's represent the state of the robot as a 3-dimensional vector containing its (x, y) coordinates and its angle θ, relative to some global reference frame. We can control the robot using its linear velocity v (change in x, y) and angular velocity ω (change in θ). For simplicity, we'll assume the robot is perfectly ideal and there's no noise in the system.

   (a) Formally describe the true of the system as a function $f : \mathcal{S} \times \mathcal{A} \to \mathcal{S}$. Assume the system works in timesteps of $dt = 1s$ and that all distances are measured in meters.

   (b) Linearize the dynamics that you derived using a first-order Taylor approximation. Is this approximation stationary or time-dependent?

   (c) Suppose the mess is at location $(x^\star, y^\star) = (2, 2)$. We want to reach that state using a small amount of energy. Write down the quadratic cost function using $Q = \text{diag}(1, 1, 0.5)$, expressing that we care about the final position more than the state, and $R = 0.2I$ to penalize large action steps.

   Now that our LQR is set up, let's find the optimal policy.

(a) Let the horizon be $T = 50$ timesteps. Write down the recursive update formula for P_t (Hint: use the Riccati equation).

(b) Write the closed form solution of the optimal policy at time $T - 1$. Verify that this is a linear function of the state.

2. (Todo.)

Chapter 4

Policy Gradients

4.1 Motivation

The scope of our problem has been gradually expanding.

1. In the first chapter, we considered *bandits* with a finite number of arms, where the only stochasticity involved was their rewards.
2. In the second chapter, we considered *MDPs* more generally, involving a finite number of states and actions, where the state transitions are Markovian.
3. In the third chapter, we considered *continuous* state and action spaces and developed the *Linear Quadratic Regulator*. We then showed how to use it to find *locally optimal solutions* to problems with nonlinear dynamics and non-quadratic cost functions.

Now, we'll continue to investigate the case of finding optimal policies in large MDPs using the self-explanatory approach of *policy optimization*. This is a general term encompassing many specific algorithms we've already seen:

- *Policy iteration* for finite MDPs,
- *Iterative LQR* for locally optimal policies in continuous control.

Here we'll see some general algorithms that allow us to optimize policies for general kinds of problems. These algorithms have been used in many groundbreaking applications, including AlphaGo, OpenAI Five. These methods also bring us into the domain where we can use *deep learning* to approximate complex, nonlinear functions.

4.2 (Stochastic) Policy Gradient Ascent

Let's suppose our policy can be *parameterized* by some parameters θ. For example, these might be a preferences over state-action pairs, or in a high-dimensional case, the weights and biases of a deep neural network. We'll talk more about possible parameterizations in section 4.5

Remember that in reinforcement learning, the goal is to *maximize reward.* Specifically, we seek the parameters that maximize the expected total reward, which we can express concisely using the value function we defined earlier:

$$\begin{aligned} J(\theta) := \mathop{\mathbb{E}}_{s_0 \sim \mu_0} V^{\pi_\theta}(s_0) = \mathbb{E} \sum_{t=0}^{T-1} r_t \\ \text{where} \quad s_0 &\sim \mu_0 \\ s_{t+1} &\sim P(s_t, a_t), \\ a_h &= \pi_\theta(s_h) \\ r_h &= r(s_h, a_h). \end{aligned} \tag{4.1}$$

We call a sequence of states, actions, and rewards a **trajectory** $\tau = (s_i, a_i, r_i)_{i=0}^{T-1}$, and the total time-discounted reward is also often called the **return** $R(\tau)$ of a trajectory. Note that the above is the *undiscounted, finite-horizon case,* which we'll continue to use throughout the chapter, but analogous results hold for the *discounted, infinite-horizon case.*

Note that when the state transitions are Markov (i.e. s_t only depends on s_{t-1}, a_{t-1}) and the policy is stationary (i.e. $a_t \sim \pi_\theta(s_t)$), we can write out the *likelihood of a trajectory* under the policy π_θ:

$$\begin{aligned} \rho_\theta(\tau) = \mu(s_0)\pi_\theta(a_0|s_0) \\ &\times P(s_1|s_0, a_0)\pi_\theta(a_1|s_1) \\ &\times \cdots \\ &\times P(s_{H-1}|s_{H-2}, a_{H-2})\pi_\theta(a_{H-1}|s_{H-1}). \end{aligned} \tag{4.2}$$

This lets us rewrite $J(\theta) = \mathbb{E}_{\tau \sim \rho_\theta} R(\tau)$.

Now how do we optimize for this function? One very general optimization technique is *gradient ascent.* Namely, the **gradient** of a function at a given point answers: At this point, which direction should we move to increase the function the most? By repeatedly moving in this direction, we can keep moving up on the graph of this function. Expressing this iteratively, we have:

$$\theta_{t+1} = \theta_t + \eta \nabla_\theta J(\pi_\theta)\Big|_{\theta=\theta_t},$$

Where η is a *hyperparameter* that says how big of a step to take each time.

In order to apply this technique, we need to be able to evaluate the gradient $\nabla_\theta J(\pi_\theta)$. How can we do this?

In practice, it's often impractical to evaluate the gradient directly. For example, in supervised learning, $J(\theta)$ might be the sum of squared prediction errors across an entire **training dataset.** However, if our dataset is very large, we might not be able to fit it into our computer's memory!

Instead, we can *estimate* a gradient step using some estimator $\widetilde{\nabla} J(\theta)$. This is called **stochastic gradient descent** (SGD). Ideally, we want this estimator to be **unbiased,** that is, on average, it matches a single true gradient step:

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

If J is defined in terms of some training dataset, we might randomly choose a *minibatch* of samples and use them to estimate the prediction error across the *whole* dataset. (This approach is known as **minibatch SGD**.)

Notice that our parameters will stop changing once $\nabla J(\theta) = 0$. This implies that 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.*

We can actually show that in a finite number of steps, SGD will find a θ that is "close" to a local optimum. More formally, suppose we run SGD for T steps, using an unbiased gradient estimator. Let the step size η_t scale as $O(1/\sqrt{t})$. Then if J is bounded and β-smooth, and the norm of the gradient estimator has a finite variance, then after T steps:

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

In another perspective, the local "landscape" of J around θ becomes flatter and flatter the longer we run SGD.

4.3 REINFORCE and Importance Sampling

Note that the objective function above, $J(\theta) = \mathbb{E}_{\tau\sim\rho_\theta} R(\tau)$, is very difficult to compute! It requires playing out every possible trajectory, which is clearly infeasible for slightly complex state and action spaces. Can we rewrite this in a form that's more convenient to implement? Specifically, suppose there is some distribution, given by a likelihood $\rho(\tau)$, that's easy to sample from (e.g. a database of existing trajectories). We can then rewrite the objective function as follows (all gradients are being taken w.r.t. θ):

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

Note that setting $\rho = \rho_\theta$ gives us an alternative form of J that's easier to implement. (Notice the swapped order of ∇ and $\mathbb{E}$!)

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

Consider expanding out ρ_θ. Note that taking its $\log$ turns it into a sum of $\log$ terms, of which only the $\pi_\theta(a_t|s_t)$ terms depend on θ, so we can simplify even further to obtain

$$\nabla J(\theta) = \mathop{\mathbb{E}}_{\tau\sim\rho_\theta} \left[\sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta(a_t|s_t) R(\tau)\right]$$

In fact, we can perform one more simplification. Intuitively, the action at step t does not affect the reward at previous timesteps. 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:

$$\begin{aligned}\nabla J(\theta) &= \mathop{\mathbb{E}}_{\tau\sim\rho_\theta}\left[\sum_{t=0}^{T-1}\nabla_\theta \log \pi_\theta(a_t|s_t)\sum_{t'=t}^{T-1} r(s_{t'}, a_{t'})\right] \\ &= \mathop{\mathbb{E}}_{\tau\sim\rho_\theta}\left[\sum_{t=0}^{T-1}\nabla_\theta \log \pi_\theta(a_t|s_t) Q^{\pi_\theta}(s_t, a_t)\right]\end{aligned} \tag{4.3}$$

Note that in the discounted case, the Q^{π_θ} term must becomes $\lambda^t Q^{\pi_\theta}$. (Make sure this makes sense!) **Exercise:** Prove that this is equivalent to the previous definitions. Also show that this works in the undiscounted case and for infinite horizon.

This expression allows us to estimate the gradient by sampling a few sample trajectories from π_θ, calculating the likelihoods of the chosen actions, and substituting these into the expression above.

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

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

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

This is an example of **importance sampling:** updating a distribution to put more density on "more important" samples (in this case trajectories).

4.4 Baselines and advantages

A central idea from supervised learning is the bias-variance tradeoff. So far, our method is *unbiased,* meaning that its average is the true policy gradient. Can we find ways to reduce the variance of our estimator as well?

We can instead subtract a **baseline function** $b_t : \mathcal{S} \to \mathbb{R}$ at each timestep t. This modifies the policy gradient as follows:

$$\nabla J(\theta) = \mathop{\mathbb{E}}_{\tau\sim\rho_\theta}\left[\sum_{t=0}^{H-1}\nabla \log \pi_\theta(a_t|s_t)\left(\left(\sum_{t'=t}^{H-1} r_t\right) - b_t(s_t)\right)\right].$$

For example, we might want b_t to estimate the average reward-to-go at a given timestep: $b_t^\theta = \mathbb{E}_{\tau\sim\rho_\theta} R_t(\tau)$. This way, the random variable $R_t(\tau) - b_t^\theta$ is centered around zero, making certain algorithms more stable.

As a better baseline, we could instead choose the *value function.* Note that the random variable $Q_t^\pi(s,a) - V_t^\pi(s)$, where the randomness is taken over the actions, is also centered around zero. (Recall $V_t^\pi(s) = \mathbb{E}_{a\sim\pi} Q_t^\pi(s,a)$.) In fact, this quantity has a particular name: the **advantage function.** In a sense, it measures how much better this action does than the average for that policy. We can now alternatively and concisely express the policy gradient as follows. Note that the advantage function effectively replaces the Q-function from Equation 4.3:

$$\nabla J(\theta) = \mathop{\mathbb{E}}_{\tau\sim\rho_\theta}\left[\sum_{t=0}^{T-1} \nabla \log \pi_\theta(a_t|s_t) A_t^{\pi_\theta}(s_t,a_t)\right]$$

Additionally, note that for an optimal policy $\pi^\star$, the advantage of a given state-action pair is always nonpositive. (Why?)

4.5 Policy parameterizations

What are some different ways we could parameterize our policy?

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 exponentiate each of them, and divide by the total:

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

However, this doesn't preserve any structure in the states or actions. While this is flexible, it is also prone to overfitting.

4.5.1 Linear in features

Instead, what if we 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 $\theta \in \mathbb{R}^p$ 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 θ represents the feature vector of the "ideal" state-action pair, as state-action pairs whose features align closely with θ are given higher probability.

The score for this parameterization is also quite elegant:

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

Plugging this into our policy gradient expression, we get

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

Why can we drop the $\mathbb{E}\,\phi(s_t,a')$ term? By linearity of expectation, consider the dropped term at a single timestep: $\mathbb{E}_{\tau\sim\rho_\theta}\left[\left(\mathbb{E}_{a'\sim\pi(s_t)}\phi(s,a')\right)A_t^{\pi_\theta}(s_t,a_t)\right]$. By Adam's Law, we can wrap the advantage term in a conditional expectation on the state s_t. Then we already know that $\mathbb{E}_{a\sim\pi(s)}A_t^{\pi}(s,a)=0$, and so this entire term vanishes.

4.5.2 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:

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

The score can then be written as

$$\nabla\log\pi_\theta(a|s)=\nabla f_\theta(s,a)-\mathop{\mathbb{E}}_{a\sim\pi_\theta(s)}\nabla f_\theta(s,a')$$