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 .

We will use *lowercase letters* to index over *uppercase letters*.

notation.

0.2.1 Multi-Armed Bandits

$[N]$	The set $\{0, 1, \ldots, N-1\}$.
K	The number of arms
T	The number of time steps (i.e. algorithm iterations).

0.2.2 MDPs

$\mathcal{S}$	The state space.
$\mathcal{A}$	The action space.
$s \in \mathcal{S}$	A state.
$a \in \mathcal{A}$	An action.
$r : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$	The reward function.
$P : \mathcal{S}times\mathcal{A} \to \triangle(\mathcal{S})$	The transition probabilities.
$\gamma \in (0, 1)$	The discount factor.
$\pi : \mathcal{S} \to \mathcal{A}$	A policy.
$V^\pi(s) \in \mathbb{R}$	The value function of policy π.
$Q^\pi(s, a) \in \mathbb{R}$	The action-value function of policy π.
$\pi^\star, V^\star, Q^\star$	The optimal policy, value function, and action-value function.

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.

Example 1.0.2: Clinical trials

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

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

1.1 Introduction

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 $0, \dots, K-1$ and use *superscripts* to indicate the arm index; since we seldom need to raise a number to a power, this hopefully 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 $\nu^k \in \Delta([0,1])$ with mean $\mu^k = \mathbb{E}_{r\sim\nu^k}[r]$.

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

for $t = 0, \dots, T$ **do**
  Agent chooses $a_t \in [K]$
  Agent receives $r_t \sim \nu^{a_t}$
  Agent updates its internal state
end for

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 defined as

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

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

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

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

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

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

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

1.2 Pure exploration (random guessing)

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

Definition 1.2.1: Pure exploration

for $t \leftarrow 0$ to $T-1$ **do**
  Choose $a_t \sim \text{Unif}([K])$
  Observe $r_t \sim \nu^{a_t}$
end for

Note that

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

so the expected regret is simply

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

This scales as $\Theta(T)$, i.e. *linear* in the number of timesteps T. There's no learning here: the agent doesn't use any information about the environment to improve its strategy.

1.3 Pure greedy

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

Definition 1.3.1: Pure greedy

for $k \leftarrow 0$ to $K-1$ **do** ▷ Exploration phase
  Observe $r^k \sim \nu^k$
end for
$\widehat{k} \leftarrow \arg\max_{k \in [K]} r^k$
for $t \leftarrow K$ to $T-1$ **do** ▷ Exploitation phase

Observe $r_t \sim \nu^{\widehat{k}}$
end for

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

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

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

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

Which is still $\Theta(T)$, the same as pure exploration! Can we do better?

1.4 Explore-then-commit

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

Definition 1.4.1: Explore-then-commit

Input: $N_{\text{explore}} \leq T/K$
for $k \leftarrow 0$ to $K - 1$ **do** ▷ Exploration phase
  for $i \leftarrow 0$ to $N_{\text{explore}} - 1$ **do**
    $r_i^k \sim \nu^k$
  end for
  $\widehat{\mu}^k \leftarrow \frac{1}{N_{\text{explore}}} \sum_{i=0}^{N_{\text{explore}}-1} r_i^k$
end for
$\widehat{k} \leftarrow \arg\max_k(\widehat{\mu}^k)$
for $t \leftarrow N_{\text{explore}}K$ to $T - 1$ **do** ▷ Exploitation phase
  $r_t \sim \nu^{\widehat{k}}$
end for

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

1.4.1 ETC regret analysis

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 prove that for any total time T, we can choose N_{explore} such that with arbitrarily high probability, the regret is sublinear. We know the regret from the exploitation phase is

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

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

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

Theorem 1.4.1: Hoeffding's inequality

Let $X_0, \ldots, X_{n-1}$ be i.i.d. random variables with $X_i \in [0,1]$ almost surely for each $i \in [n]$. Then for any $\delta > 0$,

$$\mathbb{P}\left(\left|\frac{1}{n}\sum_{i=1}^{n}(X_i - \mathbb{E}[X_i])\right| > \sqrt{\frac{\ln(2/\delta)}{2n}}\right) \leq \delta. \tag{1.2}$$

(The proof of this inequality is beyond the scope of this book.) We can apply this directly to the rewards for a given arm k, since the rewards from that arm are i.i.d.:

$$\mathbb{P}\left(|\Delta^k| > \sqrt{\frac{\ln(2/\delta)}{2N_{\text{explore}}}}\right) \leq \delta. \tag{1.3}$$

But note that we can't apply this to arm $\widehat{k}$ directly since $\widehat{k}$ is itself a random variable. Instead, we need to "uniform-ize" this bound across *all* the arms, i.e. bound the error across all the arms simultaneously, so that the resulting bound will apply *no matter what* $\widehat{k}$ "crystallizes" to.

The **union bound** provides a simple way to do this:

Theorem 1.4.2: Union bound

Consider a set of events $A_0, \ldots, A_{n-1}$. Then

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

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

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

Exercise: Prove the second statement above.

Applying the union bound across the arms for the l.h.s. event of 1.3, we have

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

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

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

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

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

Note that it suffices for N_{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:

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

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

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

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

1.5 Epsilon-greedy

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

Definition 1.5.1: Epsilon-greedy

Input: $\epsilon : \mathbb{N} \to [0, 1]$
$S^k \leftarrow 0$ for each $k \in [K]$ ▷ Total reward for arm k
$N^k \leftarrow 0$ for each $k \in [K]$ ▷ Number of pulls for arm k
for $t \leftarrow 1$ to T **do**
  if random() $< \epsilon(t)$ **then**
    $k \sim \text{Unif}([K])$ ▷ Exploration
  else
    $k \leftarrow \arg\max_k \left(\frac{S^k}{N^k}\right)$ ▷ Exploitation
  end if
  $r_t \sim \nu^k$
  $S^k \leftarrow S^k + r_t$
  $N^k \leftarrow N^k + 1$
end for

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

It turns out that setting $\epsilon_t = \sqrt[3]{K \ln(t)/t}$ also achieves a regret of $\widetilde{O}(t^{2/3}K^{1/3})$ (ignoring the logarithmic factors). (We will not prove this here.)

In ETC, 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, and explore *more* for arms that we're less certain about?

1.6 Upper Confidence Bound (UCB)

To quantify how *certain* we are about the mean of each arm, we'll compute *confidence intervals* for our estimators, and then choose the arm with the highest *upper confidence bound*. This operates on the principle of **the benefit of the doubt (i.e. optimism in the face of uncertainty)**: we'll choose the arm that we're most optimistic about.

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

In subsection 1.4.1, we were able to compute this bound using Hoeffding's inequality, which assumes that the number of samples is *fixed*. This was the case in ETC (where we pull each arm N_{explore} times), but in UCB, the number of times we pull each arm depends on the agent's actions, which in turn depend on the random rewards and are therefore stochastic. So we *can't* use Hoeffding's inequality directly.

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

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

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

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

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

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

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

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

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

where we can choose δ' depending on how tight we want the interval to be. A smaller δ' would give us a larger yet higher-confidence interval, and vice versa. We can now use this to define the UCB algorithm.

Definition 1.6.1: Upper Confidence Bound (UCB)

Input: $\delta' \in (0, 1)$
for $t \leftarrow 0$ to $T - 1$ **do**
  $k \leftarrow \arg\max_{k' \in [K]} \frac{S^{k'}}{N^{k'}} + \sqrt{\frac{\ln(2t/\delta')}{2N^{k'}}}$
  $r_t \sim \nu^k$
  $S^k \leftarrow S^k + r_t$
  $N^k \leftarrow N^k + 1$
end for

Exercise: As written, this ignores the issue that we divide by $N^k = 0$ for all arms at the beginning. How should we resolve this issue?

Intuitively, UCB prioritizes arms where:

1. $\widehat{\mu}_t^k$ 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?

1.6.1 UCB regret analysis

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

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

$$\mathbb{P}\left(\forall k \leq K, t < T. |\widehat{\mu}_t^k - \mu^k| \leq B_t^k\right) \geq 1 - \delta''$$
$$\text{where} \quad B_t^k := \sqrt{\frac{\ln(2TK/\delta'')}{2N_t^k}}.$$

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:

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

Summing this across timesteps gives

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

Putting everything together gives

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

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

1.6.2 Lower bound on regret (intuition)

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

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

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

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

1.7 Thompson sampling and Bayesian bandits

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

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

Definition 1.7.1: Thompson sampling

Input: the prior distribution $\pi \in \Delta([0,1]^K)$
for $t \leftarrow 0$ to $T-1$ **do**
  $\boldsymbol{\mu} \sim \pi(\cdot \mid a_0, r_0, \ldots, a_{t-1}, r_{t-1})$
  $a_t \leftarrow \arg\max_{k \in [K]} \mu^k$
  $r_t \sim \nu^{a_t}$ ▷ Observe reward
end for

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

Example 1.7.1: Bayesian Bernoulli bandit

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

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

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

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

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

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

It turns out that asymptotically, Thompson sampling is optimal in the following sense. Lai and Robbins [2] prove an *instance-dependent* lower bound that says for *any* bandit algorithm,

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

where

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

measures the **Kullback-Leibler divergence** from the Bernoulli distribution with mean μ^k to the Bernoulli distribution with mean $\mu^\star$. It turns out that Thompson sampling achieves this lower bound with equality! That is, not only is the error *rate* optimal, but the *constant factor* is optimal as well.

1.8 Contextual bandits

In the above MAB environment, the reward distribution of the arms remains constant for each action that we take. However, in many real-world settings, we might receive additional information

that affects the reward distributions of the arms. For example, in the online advertising case where each arm corresponds to an ad we could show the user, we might receive information about the user's preferences that changes how likely they are to click on a given ad. We can model such environments using **contextual bandits**. At each timestep t, a new *context* x_t is drawn from some distribution ν_{x}. The learner gets to observe the context, and choose an action a_t according to some policy $\pi_t(x_t)$. Then, the learner observes the reward $\nu^{a_t}(x_t)$.

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

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

Recall that running UCB for T timesteps on an MAB with K arms achieves a regret bound of $\widetilde{O}(\sqrt{TK})$. So in this problem, we would achieve regret $\widetilde{O}(\sqrt{TK|\mathcal{X}|})$ in the contextual MAB, which has a polynomial dependence on $|\mathcal{X}|$. But in a situation where we have large, or even infinitely many contexts (e.g. in the case where our context is a continuous value), this becomes intractable. Note that this treats the different contexts as entirely unrelated to each other, while in practice, often contexts are *related* to each other in some way: for example, we might want to advertise to users with similar preferences. How can we incorporate this structure into our solution?

1.8.1 Linear contextual bandits

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

$$\widehat{\theta}_t^k = \arg\min_{\theta \in \mathbb{R}^d} \sum_{i=0}^{t-1} (r_i - x_i^\top \theta)^2 \mathbf{1}\left\{a_i = k\right\}.$$

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

$$\begin{aligned} \widehat{\theta}_t^k &= (A_t^k)^{-1} \sum_{i=0}^{t-1} x_i r_i \mathbf{1}\left\{a_i = k\right\} \\ \text{where} \quad A_t^k &= \sum_{i=0}^{t-1} x_i x_i^\top \mathbf{1}\left\{a_i = k\right\}. \end{aligned} \tag{1.4}$$

We can now apply the UCB algorithm in this environment in order to balance *exploration* of new arms and *exploitation* of arms that we believe to have high reward. But how should we construct the upper confidence bound? Previously, we treated the pulls of an arm as i.i.d. samples and used Hoeffding's inequality to bound the distance of the sample mean, our estimator, from the true mean. However, now our estimator is not a sample mean, but rather depends on the OLS estimator above (1.4). Instead, we'll use **Chebyshev's inequality** to construct an upper confidence bound.

Theorem 1.8.1: Chebyshev's inequality

For a random variable Y such that $\mathbb{E}Y = 0$ and $\mathbb{E}Y^2 = \sigma^2$,

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

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

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

Note that we haven't explained why $x_t^\top (A_t^k)^{-1} x_t$ is the correct expression for the covariance matrix of $x_t^\top \widehat{\theta}_t^k$. This result follows from performing some algebra on the definition of the OLS estimator (1.4). The first term is exactly our predicted reward $\widehat{\mu}_t^k(x_t)$. To interpret the second term, note that

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

where

$$\Sigma_t^k = \frac{1}{N_t^k} \sum_{i=0}^{t-1} \mathbf{1}\{a_i = k\} x_i x_i^\top$$

is the empirical covariance matrix of the contexts. That is, the learner is encouraged to choose arms when x_t is *not aligned* with the data seen so far, or if arm k has not been explored much so N_t^k is small. The LinUCB algorithm follows as a direct generalization:

Definition 1.8.1: LinUCB

for $t \in [T]$ **do**
  for $k \in [K]$ **do**
    $A_t^k \leftarrow \sum_{i=0}^{t-1} \mathbf{1}\{a_i = k\} x_i x_i^\top + \lambda I$
    $\theta_t^k \leftarrow (A_t^k)^{-1} \sum_{i=0}^{t-1} x_i r_i \mathbf{1}\{a_t = k\}$
  end for
  Given context x_t
  Choose $a_t = \arg\max_k x_t^\top \widehat{\theta}_t^k + c_t\sqrt{x_t^\top (A_t^k)^{-1} x_t}$
  Observe reward $r_t \sim \nu^{a_t}(x_t)$
end for

Note that we include a λI regularization term to ensure that A_t^k is invertible.

c_t is similar to the $\log(2t/\delta')$ term of UCB: It depends logarithmically on

- $\frac{1}{\delta}$, where δ is the probability with which the bound holds;
- t and d, which we uniformize over, involving $\det A_t^k$.

actually defi
fully

Using similar tools for UCB, we can also prove an $\widetilde{O}(\sqrt{T})$ regret bound, which does much better

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 interactive environment. This is a very general problem! How can we *formalize* this task in a way that is both *sufficiently general* yet also tractable enough for *fruitful analysis*?

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

- **Board games** 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. Furthermore, these are environments where the **state transitions**, the "rules" of the environment, only depend on the *most recent* state and action. This is called the **Markov property**.

Definition 2.1.1: Markov property

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

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

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

We'll see that this simple assumption leads to a rich set of problems and algorithms. Environments with the Markov property are called **Markov decision processes** (MDPs) and will be the focus of this chapter.

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

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

In each setting, we'll describe how to evaluate different **policies** (strategies for choosing actions) and how to compute (or approximate) the **optimal policy** for a given MDP. We'll introduce the **one-step consistency condition**, which allows us to analyze the whole series of interactions in terms of individual timesteps.

2.2 Finite horizon MDPs

Definition 2.2.1: Finite-horizon Markov decision process

The components of a finite-horizon Markov decision process are:

1. The **state** that the agent interacts with. We use $\mathcal{S}$ to denote the set of possible states, called the **state space**.
2. The **actions** that the agent can take. We use $\mathcal{A}$ to denote the set of possible actions, called the **action space**.
3. Some **initial state distribution** $\mu \in \Delta(\mathcal{S})$.
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 MDP

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 orderly.

The state transitions are as follows: if you tidy the room, it becomes (or remains) orderly; if you ignore the room, it might become messy.

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 $t = 6$ correspond to Sunday.

2.2.1 Policies

Definition 2.2.2: Policies

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

There are three axes along which policies can vary: their outputs, inputs, and time-dependence. We'll discuss each of these in turn.

1. **Deterministic or stochastic.** A deterministic policy outputs actions while a stochastic policy outputs *distributions* over actions.
2. **State-dependent or history-dependent.** A state-dependent (a.k.a. "Markovian") policy only depends on the current state, while a history-dependent policy depends on the sequence of past states, actions, and rewards. We'll only consider state-dependent policies in this course.
3. **Stationary or time-dependent.** A stationary policy remains the same function at all time steps, while a time-dependent policy $\pi = \{\pi_0, \ldots, \pi_{H-1}\}$ specifies a different function π_h at each time step h.

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

Example 2.2.2: Tidying policies

Here are some possible policies for the tidying MDP (2.2.1):

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

2.2.2 Trajectories

Definition 2.2.3: Trajectories

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

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

where $r_h = r(s_h, a_h)$. (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 repeatedly choosing actions according to the policy, transitioning according to the state transitions, and observing the 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 policies in 2.2.2 have generated this trajectory?

Note that for a state-dependent policy, using the Markov property (2.1.1), we can specify this probability distribution in an **autoregressive** way (i.e. one timestep at a time):

Definition 2.2.4: Autoregressive trajectory distribution

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

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

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

2.2.3 Value functions

The main goal of RL is to find a policy that maximizes the average total reward $r_0 + \cdots + r_{H-1}$. (Note that this is a random variable that depends on the policy.) Let's introduce some notation for analyzing this quantity.

A policy's **value function** is its expected total reward *starting in a given state at a given*

time:

Definition 2.2.5: Value function

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

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

Definition 2.2.6: Action-value function

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

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

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

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

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

The one-step (Bellman) consistency equation

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

Definition 2.2.7: one-step consistency equation for the value function

$$V_h^\pi(s) = \mathop{\mathbb{E}}_{\substack{a\sim\pi_h(s)\\ s'\sim P(s,a)}}[r(s,a) + V_{h+1}^\pi(s')] \tag{2.1}$$

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

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

Definition 2.2.8: one-step consistency equation for action-values

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

Theorem 2.2.1: The one-step consistency equation for deterministic policies

Note that for deterministic policies, the one-step consistency equations simplify to

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

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":

Definition 2.2.9: Bellman operator

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

We'll call $\mathcal{J}^\pi : (\mathcal{S} \to \mathbb{R}) \to (\mathcal{S} \to \mathbb{R})$ the **Bellman operator** of π. Note that it's defined on any "value function" mapping states to real numbers; v doesn't have to be a well-defined value function for some policy (hence the lowercase notation). The Bellman operator also gives us a concise way to express the one-step consistency equation (2.1):

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

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

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

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 one-step consistency equation (2.1) gives us a convenient algorithm for evaluating stationary policies: it expresses the value function at timestep h as a function of the value function at timestep $h + 1$. This means we can start at the end of the time horizon, where the value is known, and work backwards in time, using the one-step consistency equation to compute the value function at each time step.

Definition 2.2.10: Dynamic programming for policy evaluation

$V_h(s) \leftarrow 0$ for all $t \in \{0, \dots, H\}, s \in S$
for $t = H-1, \dots, 0$ **do**
  for $s \in \mathcal{S}, a \in \mathcal{A}, s' \in \mathcal{S}$ **do**
    $V_h(s) \leftarrow V_h(s) + \pi_h(a \mid s) P(s' \mid s, a)[r(s,a) + V_{h+1}(s')]$
  end for
end for

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

Exercise: Do you see where we compute Q_h^π along the way? Make this step explicit.

Example 2.2.4: Tidying policy evaluation

Let's evaluate the policy from 2.2.2 that tidies if and only if the room is messy. We'll use the one-step consistency equation to compute the value function at each time step.

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

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

2.2.5 Optimal policies

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

Definition 2.2.11: Optimal policies

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

$$\begin{aligned} V_h^{\pi^\star}(s) &= \mathop{\mathbb{E}}_{\tau\sim\rho^{\pi^\star}}[r_h + \cdots + r_{H-1} \mid s_h = s] \\ &\geq \mathop{\mathbb{E}}_{\tau\sim\rho^{\pi}}[r_h + \cdots + r_{H-1} \mid \tau_h] \quad \forall \pi, \tau_h, h \in [H] \end{aligned} \tag{2.2}$$

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

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

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

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

Theorem 2.2.2: 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

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

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

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

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

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

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

Proof:

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

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

We can now apply this result recursively to get

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

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

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

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

Dynamic programming

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

Definition 2.2.12: DP for optimal policy

We can solve for the optimal policy in an finite-horizon 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

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

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

Note that this algorithm is identical to the policy evaluation algorithm 2.2.10, but instead of *averaging* over the actions chosen by a policy, we instead simply take a *maximum* over the action-values. We'll see this relationship between **policy evaluation** and **optimal policy computation** show up again in the infinite-horizon setting.

Example 2.2.5: Optimal policy for the tidying MDP

Left as an exercise.

2.3 Infinite horizon MDPs

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

In this chapter, we'll describe the necessary adjustments from the finite-horizon case to make the problem tractable. We'll show that the Bellman operator (2.2.3) in the discounted reward setting is a **contraction mapping** for any policy. We'll discuss how to evaluate policies (i.e. compute their corresponding value functions). Finally, we'll present and analyze two iterative algorithms, based on the Bellman operator, for computing the optimal policy: **value iteration** and **policy iteration**.

2.3.1 Differences from finite-horizon

Discounted rewards

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

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

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

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

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

The other components of the MDP remain the same:

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

Stationary policies

The time-dependent policies from the finite-horizon case become difficult to handle in the infinite-horizon case. In particular, many of the DP approaches we saw required us to start at the end of the trajectory, which is no longer possible. We'll shift to **stationary** policies $\pi : \mathcal{S} \to \mathcal{A}$ (deterministic) or $\Delta(\mathcal{A})$ (stochastic).

Exercise: Which of the policies in 2.2.2 are stationary?

Value functions and one-step consistency

We also consider stationary value functions $V^\pi : \mathcal{S} \to \mathbb{R}$ and $Q^\pi : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$. We need to insert a factor of γ into the one-step consistency equation (2.1) to account for the discounting:

$$\begin{aligned} V^\pi(s) &= \mathop{\mathbb{E}}_{\tau\sim\rho^\pi}[r_h + \gamma r_{h+1} + \gamma^2 r_{h+2} \cdots \mid s_h = s] && \text{for any } h \in \mathbb{N} \\ &= \mathop{\mathbb{E}}_{\substack{a\sim\pi(s) \\ s'\sim P(s,a)}}[r(s,a) + \gamma V^\pi(s')] \\ Q^\pi(s,a) &= \mathop{\mathbb{E}}_{\tau\sim\rho^\pi}[r_h + \gamma r_{h+1} + \gamma^2 r_{h+2} + \cdots \mid s_h = s, a_h = a] && \text{for any } h \in \mathbb{N} \\ &= r(s,a) + \gamma \mathop{\mathbb{E}}_{\substack{s'\sim P(s,a) \\ a'\sim\pi(s')}}[Q^\pi(s',a')] \end{aligned} \tag{2.3}$$

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

2.3.2 The Bellman operator is a contraction mapping

Recall from 2.2.3 that the Bellman operator $\mathcal{J}^\pi$ for a policy π takes in a "value function" $v : \mathcal{S} \to \mathbb{R}$ and returns the r.h.s. of the Bellman equation for that "value function". In the infinite-horizon setting, this is

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

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

Definition 2.3.1: Contraction mapping

Let X be some space with a norm $\|\cdot\|$. We call an operator $f : X \to X$ a **contraction mapping** if for any $x, y \in X$,

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

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

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

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

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

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

$$\|f^{(t)}(x) - x^\star\| \leq \gamma^t \|x - x^\star\|. \tag{2.4}$$

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

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

i.e. we compare the "value functions" on the state that causes the biggest gap between them. Then (2.4) implies that if we repeatedly apply $\mathcal{J}^\pi$ to any starting "value function", we will eventually converge to V^π:

$$\|(\mathcal{J}^\pi)^{(t)}(v) - V^\pi\|_\infty \leq \gamma^t \|v - V^\pi\|_\infty. \tag{2.5}$$

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

Theorem 2.3.1: The Bellman operator is a contraction mapping

We aim to show that

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

Proof: for all states $s \in \mathcal{S}$,

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

2.3.3 Tabular case (linear algebraic notation)

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

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

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

(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:

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

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

Example 2.3.1: Tidying MDP

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

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

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

2.3.4 Policy evaluation

The backwards DP technique we used in the finite-horizon case (2.2.4) no longer works since there is no "final timestep" to start from. We'll need another approach to policy evaluation.

The one-step consistency conditions yield a system of equations we can solve 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 value function.

Tabular case for deterministic policies

The one-step consistency equation for a deterministic policy can be written in tabular notation as

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

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

$$V^\pi = (I - \gamma P^\pi)^{-1} r^\pi. \tag{2.6}$$

Note we've assumed that $I - \gamma P^\pi$ is invertible. Can you see why this is the case?

(Recall that a linear operator, i.e. a square matrix, is invertible if and only if its null space is trivial; that is, it doesn't map any nonzero vector to zero. In this case, we can see that $I - \gamma P^\pi$ is invertible because it maps any nonzero vector to a vector with at least one nonzero element.)

Example 2.3.2: Tidying policy evaluation

Let's use the same policy π that tidies if and only if the room is messy. Setting $\gamma = 0.95$, we must invert

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

The inverse to two decimal points is

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

Thus the value function is

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

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

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 $v^{(0)}$ with elements in $[0, 1/(1-\gamma)]$ and then iterate the Bellman operator:

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

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

Then, as we showed in (2.5), 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:

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

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

$$T = O\left(\frac{1}{1-\gamma}\log\left(\frac{1}{\epsilon(1-\gamma)}\right)\right). \tag{2.7}$$

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

2.3.5 Optimal policies

Now let's move on to solving for an optimal policy in the infinite-horizon case. As in the finite-horizon case (2.2), an **optimal policy** $\pi^\star$ is one that does at least as well as any other policy in all situations. That is, for all policies π, states $s \in \mathcal{S}$, times $h \in \mathbb{N}$, and initial trajectories $\tau_h = (s_0, a_0, r_0, \ldots, s_h)$ where $s_h = s$,

$$\begin{aligned} V^{\pi^\star}(s) &= \mathop{\mathbb{E}}_{\tau\sim\rho^{\pi^\star}}[r_h + \gamma r_{h+1} + \gamma^2 r_{h+2} + \cdots \mid s_h = s] \\ &\geq \mathop{\mathbb{E}}_{\tau\sim\rho^{\pi}}[r_h + \gamma r_{h+1} + \gamma^2 r_{h+2} + \cdots \mid \tau_h] \end{aligned} \tag{2.8}$$

Once again, all optimal policies share the same **optimal value function** $V^\star$, and the greedy policy w.r.t. this value function is optimal.

Exercise: Verify this by modifying the proof 2.2.2 from the finite-horizon case.

So how can we compute such an optimal policy? We can't use the backwards DP approach from the finite-horizon case (2.2.12) since there's no "final timestep" to start from. Instead, we'll exploit the fact that the one-step consistency equation (2.3) for the optimal value function doesn't depend on any policy:

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

Exercise: Verify this by substituting the greedy policy into the one-step consistency equation.

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

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

Value iteration

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

Definition 2.3.2: Value iteration

$v^{(0)} \leftarrow 0$
for $t = 0, 1, 2, \ldots, T-1$ **do**
  $v^{(t+1)} \leftarrow \mathcal{J}^\star(v^{(t)})$
end for
return $v^{(T)}$

Note that the runtime analysis for an ϵ-optimal value function is exactly the same as iterative policy evaluation (2.3.4)! This is because value iteration is simply the special case of applying iterative policy evaluation to the *optimal* value function.

As the final step of the algorithm, to return an actual policy $\widehat{\pi}$, we can simply act greedily w.r.t. the final iteration $v^{(T)}$ of our above algorithm:

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

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

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

Theorem 2.3.2: Greedy policy value worsening

We aim to show that

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

where $\widehat{\pi}(s) = \arg\max_a q(s,a)$ is the greedy policy w.r.t.

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

Proof: We first have

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

Let's bound these two quantities separately.

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

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

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

$$Q^\star(s, \pi^\star(s)) - Q^\star(s, \widehat{\pi}(s)) \leq [Q^\star(s, \pi^\star(s)) - q(s, \pi^\star(s))] + [q(s, \widehat{\pi}(s)) - Q^\star(s, \widehat{\pi}(s))]$$

$$= \gamma \mathop{\mathbb{E}}_{s' \sim P(s,\pi^\star(s))}[V^\star(s') - v(s')] + \gamma \mathop{\mathbb{E}}_{s' \sim P(s,\widehat{\pi}(s))}[v(s') - V^\star(s')]$$
$$\leq 2\gamma \|v - V^\star\|_\infty.$$

The second quantity is bounded by

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

and thus

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

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

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

This means, using (2.7), we need to run value iteration for

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

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

Policy iteration

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

Definition 2.3.3: Policy Iteration

$\pi^{(0)} : \mathcal{S} \to \mathcal{A}$ arbitrary
for $t = 0, \ldots, T-1$ **do**
  $V^{\pi^{(t)}} \leftarrow (I - \gamma P^{\pi^{(t)}})^{-1} r^{\pi^{(t)}}$ ▷ (Exact) Policy Evaluation (2.6)
  $Q^{\pi^{(t)}}(s,a) \leftarrow r(s,a) + \gamma \mathbb{E}_{s' \sim P(s,a)}[V^{\pi^{(t)}}(s')]$
  $\pi^{(t+1)}(s) \leftarrow \arg\max_a Q^{\pi^{(t)}}(s,a)$ ▷ Policy Improvement
end for

Although PI appears more complex than VI, we'll use the same contraction property (2.3.1) to show convergence. This will give us the same runtime bound as value iteration and iterative

policy evaluation for an ϵ-optimal value function (2.7), although in practice, PI often converges much faster.

Theorem 2.3.3: Policy Iteration runtime and convergence

We aim to show that the number of iterations required for an ϵ-accurate estimate of the optimal value function is

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

This bound follows from the contraction property (2.5):

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

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

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

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

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

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

$$\begin{aligned}V^{\pi^{t+1}}(s) - V^{\pi^t}(s) &\geq V^{\pi^{t+1}}(s) - \mathcal{J}^\star(V^{\pi^t})(s) \\ &= \gamma \mathop{\mathbb{E}}_{s'\sim P(s,\pi^{t+1}(s))}\left[V^{\pi^{t+1}}(s') - V^{\pi^t}(s')\right].\end{aligned} \tag{2.9}$$

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

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

which implies that $V^{\pi^{t+1}}(s) \geq V^{\pi^t}(s)$ for all s (since the r.h.s. converges to zero). We can then plug this back into (2.9) to get the desired result:

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

$$\geq 0$$
$$V^{\pi^{t+1}}(s) \geq [\mathcal{J}^\star(V^{\pi^t})](s)$$

This means we can now apply the Bellman convergence result (2.5) to get

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

2.4 Summary

- Markov decision processes (MDPs) are a framework for sequential decision making under uncertainty. They consist of a state space $\mathcal{S}$, an action space $\mathcal{A}$, an initial state distribution $\mu \in \Delta(\mathcal{S})$, a transition function $P(s' \mid s, a)$, and a reward function $r(s, a)$. They can be finite-horizon (ends after H timesteps) or infinite-horizon (where rewards scale by $\gamma \in (0, 1)$ at each timestep).
- Our goal is to find a policy π that maximizes expected total reward. Policies can be **deterministic** or **stochastic**, **state-dependent** or **history-dependent**, **stationary** or **time-dependent**.
- A policy induces a distribution over **trajectories**.
- We can evaluate a policy by computing its **value function** $V^\pi(s)$, which is the expected total reward starting from state s and following policy π. We can also compute the **state-action value function** $Q^\pi(s, a)$, which is the expected total reward starting from state s, taking action a, and then following policy π. In the finite-horizon setting, these also depend on the timestep h.
- The **one-step consistency equation** is an equation that the value function must satisfy. It can be used to solve for the value functions exactly. Thinking of the r.h.s. of this equation as an operator on value functions gives the **Bellman operator**.
- In the finite-horizon setting, we can compute the optimal policy using **dynamic programming**.
- In the infinite-horizon setting, we can compute the optimal policy using **value iteration** or **policy iteration**.

Contents

Chapter 3

Linear Quadratic Regulators

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

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

(b) Boston Dynamics's Spot robot.

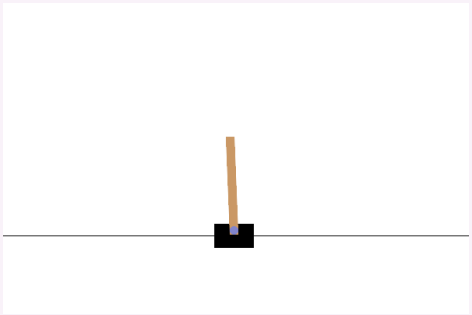
Figure 3.1: Examples of control tasks.

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

This chapter introduces a fundamental tool to solve a simple class of continuous control problems: the **linear quadratic regulator**. We will then extend this basic method to more complex settings.

Example 3.0.1: CartPole

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



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

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

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

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

3.1 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 H. These have equivalents in the control setting:

- The state and action spaces are *continuous* rather than finite. That is, $\mathcal{S} \subseteq \mathbb{R}^{n_s}$ and $\mathcal{A} \subseteq \mathbb{R}^{n_a}$, where n_s and n_a are the corresponding dimensions of these spaces, i.e. the number of coordinates to specify a single state or action respectively.
- We call the state transitions the **dynamics** of the system. In the most general case, these might change across timesteps and also include some stochastic **noise** w_h at each timestep. We denote these dynamics as the function f_h such that $s_{h+1} = f_h(s_h, a_h, w_h)$. Of course, we can simplify to cases where the dynamics are *deterministic/noise-free* (no w_h term) and/or *time-homogeneous* (the same function f across timesteps).
- Instead of maximizing the reward function, we seek to minimize the **cost function** c_h :

$\mathcal{S} \times \mathcal{A} \to \mathbb{R}$. Often, the cost function describes *how far away* we are from a **target state-action pair** $(s^\star, a^\star)$. An important special case is when the cost is *time-homogeneous*; that is, it remains the same function c at each timestep h.

- We seek to minimize the *undiscounted* cost within a *finite time horizon* H. Note that we end an episode at the final state s_H – there is no a_H, and so we denote the cost for the final state as $c_H(s_H)$.

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

Definition 3.1.1: General optimal control problem

$$\begin{aligned} \min_{\pi_0,\dots,\pi_{H-1}:\mathcal{S}\to\mathcal{A}} \quad & \mathbb{E}\left[\left(\sum_{h=0}^{H-1} c_h(s_h, a_h)\right) + c_H(s_H)\right] \\ \text{where} \quad & s_{h+1} = f_h(s_h, a_h, w_h), \\ & a_h = \pi_h(s_h) \\ & s_0 \sim \mu_0 \\ & w_h \sim \text{noise} \end{aligned} \tag{3.1}$$

3.1.1 A first attempt: Discretization

Can we solve this problem using tools from the finite MDP setting? If $\mathcal{S}$ and $\mathcal{A}$ were finite, then we'd be able to work backwards using the DP algorithm for computing the optimal policy in an MDP (2.2.12). This inspires us to try *discretizing* the problem.

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

However, the discretized $\widetilde{\mathcal{S}}$ and $\widetilde{\mathcal{A}}$ may be finite, but they may be infeasibly large: we must divide *each dimension* into intervals of length ε, resulting in $|\widetilde{\mathcal{S}}| = (B_s/\varepsilon)^{n_s}$ and $|\widetilde{\mathcal{A}}| = (B_a/\varepsilon)^{n_a}$. To get a sense of how quickly this grows, consider $\varepsilon = 0.01, n_s = n_a = 10$. Then the number of elements in the transition matrix would be $|\widetilde{\mathcal{S}}|^2|\widetilde{\mathcal{A}}| = (100^{10})^2(100^{10}) = 10^{60}$! (That's a trillion trillion trillion trillion trillion trillion.)

What properties of the problem could we instead make use of? Note that by discretizing the state and action spaces, we implicitly assumed that rounding each state or action vector by some tiny amount ε wouldn't change the behavior of the system by much; namely, that the cost and dynamics were relatively *continuous*. Can we use this continuous structure in other ways? This leads us to the **linear quadratic regulator**.

3.2 The Linear Quadratic Regulator

The optimal control problem (3.1.1) seems highly complex in its general case. Is there a relevant simplification that we can analyze?

Let us consider *linear dynamics* and an *upward-curved quadratic cost function* (in both arguments). We will also consider a time-homogenous cost function that targets $(s^\star, a^\star) = (0, 0)$. This model is called the **linear quadratic regulator** (LQR) and is a fundamental tool in control theory. Solving the LQR problem will additionally enable us to *locally approximate* more complex setups using *Taylor approximations*.

Definition 3.2.1: The linear quadratic regulator

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

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

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

Upward-curved quadratic, time-homogeneous cost function:

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

We require Q and R to both be positive definite matrices so that c has a well-defined unique minimum. We can furthermore assume without loss of generality that they are both symmetric (see exercise below).

This results in the LQR optimization problem:

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

Exercise: We've set Q and R to be *symmetric* positive definite (SPD) matrices. Here we'll show that the symmetry condition can be imposed without loss of generality. Show that replacing Q with $(Q + Q^\top)/2$ (which is symmetric) yields the same cost function.

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

Definition 3.2.2: Value functions for LQR

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

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

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

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

3.3 Optimality and the Riccati Equation

In this section, we'll compute the optimal value function $V_h^\star$, Q-function $Q_h^\star$, and policy $\pi_h^\star$ in the LQR setting (3.2.1) using **dynamic programming** in a very similar way to the DP algorithms in the MDP setting (2.2.4):

1. We'll compute $V_H^\star$ (at the end of the horizon) as our base case.
2. Then we'll work backwards in time, using $V_{h+1}^\star$ to compute $Q_h^\star$, $\pi_h^\star$, and $V_h^\star$.

Along the way, we will prove the striking fact that the solution has very simple structure: $V_h^\star$ and $Q_h^\star$ are *upward-curved quadratics* and $\pi_h^\star$ is *linear* and furthermore does not depend on the noise!

Definition 3.3.1: Optimal value functions for LQR

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

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

Theorem 3.3.1: Optimal value function in LQR is a upward-curved quadratic

At each timestep $h \in [H]$,

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

for some symmetric positive definite matrix $P_h \in \mathbb{R}^{n_s \times n_s}$ and vector $p_h \in \mathbb{R}^{n_s}$.

Theorem 3.3.2: Optimal policy in LQR is linear

At each timestep $h \in [H]$,

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

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

Base case: At the final timestep, there are no possible actions to take, and so $V_H^\star(s) = c(s) = s^\top Q s$. Thus $V_H^\star(s) = s^\top P_h s + p_h$ where $P_H = Q$ and p_H is the zero vector.

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

Step 1. Show that $Q_h^\star(s,a)$ is a upward-curved quadratic (in both s and a).

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

Step 3. Show that $V_h^\star(s)$ is a upward-curved quadratic.

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

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

Step 1. We aim to show that $Q_h^\star(s)$ is a upward-curved quadratic. Recall that the definition of $Q_h^\star : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$ is

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

Recall $c(s,a) = s^\top Q s + a^\top R a$. Let's consider the average value over the next timestep. The only randomness in the dynamics comes from the noise $w_{h+1} \sim \mathcal{N}(0, \sigma^2 I)$, so we can write out this expected value as:

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

Summing and combining like terms, we get

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

Note that the terms that are linear in w_h have mean zero and vanish. Now consider the remaining expectation over the noise. By expanding out the product and using linearity of expectation, we can write this out as

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

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

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

Off the diagonal, since the elements of w_{h+1} are independent, the expectation factors, and since each element has mean zero, the term disappears:

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

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

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

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

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

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

$$\nabla_{aa} Q^\star_h(x,u) = R + B^\top P_{h+1} B$$

This is fairly straightforward: recall that in our definition of LQR, we assumed that R is SPD (see Definition 3.2.1). Also note that since P_{h+1} is SPD (by the inductive hypothesis), so too must be $B^\top P_{h+1} B$. (If this isn't clear, try proving it as an exercise.) Since the sum of two SPD matrices is also SPD, we have that $R + B^\top P_{h+1} B$ is SPD, and so $Q_h^\star$ is indeed a upward-curved quadratic with respect to a.

Step 2. Now we aim to show that $\pi_h^\star$ is linear. Since $Q_h^\star$ is a upward-curved quadratic, finding its minimum over a is easy: we simply set the gradient with respect to a equal to zero and solve for a. First, we calculate the gradient:

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

Setting this to zero, we get

$$\begin{aligned}0 &= (R + B^\top P_{h+1} B) \pi_h^\star(s) + B^\top P_{h+1} A s \\ \pi_h^\star(s) &= (R + B^\top P_{h+1} B)^{-1} (-B^\top P_{h+1} A s) \\ &= -K_h s, \end{aligned} \tag{3.3}$$

where $K_h = (R + B^\top P_{h+1} B)^{-1} B^\top P_{h+1} A$. Note that this optimal policy doesn't depend on the starting distribution μ_0. It's also fully **deterministic** and isn't affected by the noise terms $w_0, \ldots, w_{H-1}$.

Step 3. To complete our inductive proof, we must show that the inductive hypothesis is true at time h; that is, we must prove that $V_h^\star(s)$ is a upward-curved quadratic. Using the identity $V_h^\star(s) = Q_h^\star(s, \pi^\star(s))$, we have:

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

Note that with respect to s, this is the sum of a quadratic term and a constant, which is exactly what we were aiming for! The constant term is clearly $p_h = \operatorname{Tr}(\sigma^2 P_{h+1}) + p_{h+1}$. We can simplify the quadratic term by substituting in K_h. Notice that when we do this, the $(R + B^\top P_{h+1} B)$ term in the expression is cancelled out by its inverse, and the remaining terms combine to give the **Riccati equation**:

Definition 3.3.2: Riccati equation

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

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

1. It's defined **recursively.** Given the dynamics defined by A and B, and the state cost matrix Q, we can recursively calculate P_h across all timesteps starting from $P_H = Q$.
2. P_h often appears in calculations surrounding optimality, such as $V_h^\star, Q_h^\star$, and $\pi_h^\star$.

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

Now we've shown that $V_h^\star(s) = s^\top P_h s + p_h$, which is a upward-curved quadratic, and this concludes our proof. ■

In summary, we just demonstrated that at each timestep $h \in H$, the optimal value function $V_h^\star$ and optimal Q-function $Q_h^\star$ are both upward-curved quadratics and the optimal policy $\pi_h^\star$ is linear. We also showed that all of these quantities can be calculated using a sequence of symmetric matrices $P_0, \dots, P_H$ that can be defined recursively using the Riccati equation (3.3.2).

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

3.3.1 Expected state at time h

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

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

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

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

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

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

Exercise: Verify this.

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

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

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

3.4 Extensions

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

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

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

3.4.1 Time-dependent dynamics and cost function

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

Exercise: Walk through the above derivation to verify this claim.

The modified problem is now defined as follows:

Definition 3.4.1: Time-dependent LQR

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

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

Definition 3.4.2: Time-dependent Riccati Equation

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

Note that this is just the time-homogeneous Riccati equation (Definition 3.3.2), 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.

3.4.2 More general quadratic cost functions

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

$$c_h(s_h, a_h) = (s_h^\top Q_h s_h + s_h^\top M_h a_h + a_h^\top R_h a_h) + (s_h^\top q_h + a_h^\top r_h) + c_h. \tag{3.4}$$

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

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

Exercise: Derive the optimal solution. (You will need to slightly modify the above proof.)

3.4.3 Tracking a predefined trajectory

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

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

Note that this punishes states and actions that are far from the intended trajectory. By expanding out these multiplications, we can see that this is actually a special case of the more general quadratic cost function above (Equation 3.4):

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

3.5 Approximating nonlinear dynamics

The LQR algorithm solves for the optimal policy when the dynamics are *linear* and the cost function is an *upward-curved quadratic*. However, real settings are rarely this simple! Let's return

to the CartPole example from the start of the chapter (Example 3.0.1). The dynamics (physics) aren't linear. How can we approximate this by an LQR problem?

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

Definition 3.5.1: Nonlinear control problem

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

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

This is now only slightly simplified from the general optimal control problem (see 3.1.1). Here, we don't know an analytical form for the dynamics f or the cost function c, but we assume that we're able to *query/sample/simulate* them to get their values at a given state and action. To clarify, consider the case where the dynamics are given by real world physics. We can't (yet) write down an expression for the dynamics that we can differentiate or integrate analytically. However, we can still *simulate* the dynamics and cost function by running a real-world experiment and measuring the resulting states and costs. How can we adapt LQR to this more general nonlinear case?

3.5.1 Local linearization

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

Linearizing the dynamics around $(s^\star, a^\star)$ gives:

$$f(s,a) \approx f(s^\star,a^\star) + \nabla_s f(s^\star,a^\star)(s-s^\star) + \nabla_a f(s^\star,a^\star)(a-a^\star)$$

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

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

$$c(s,a) \approx c(s^\star,a^\star) \quad \text{constant term}$$

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

where the gradients and Hessians are defined as

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

Exercise: Note that this cost can be expressed in the general quadratic form seen in Equation 3.4. Derive the corresponding quantities Q, R, M, q, r, c.

3.5.2 Finite differencing

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

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

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

3.5.3 Local convexification

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

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

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

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

Note that Hessian matrices are generally symmetric, so we can apply this process to Q and R to obtain the positive definite approximations $\widetilde{Q}$ and $\widetilde{R}$. Now that we have a upward-curved quadratic approximation to the cost function, and a linear approximation to the state transitions, we can simply apply the time-homogenous LQR methods from section 3.3.

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

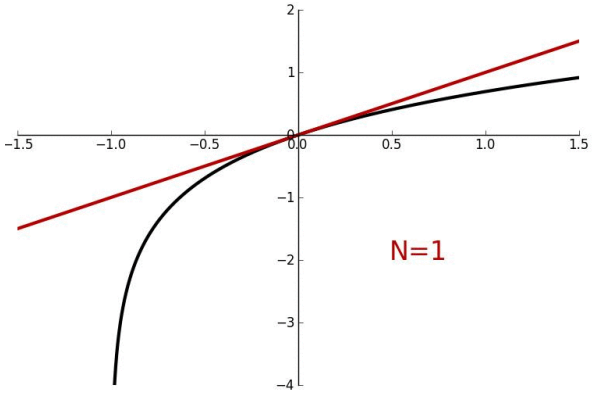


Figure 3.2: Local linearization might only be accurate in a small region around the point of linearization.

3.5.4 Iterative LQR

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

Definition 3.5.2: Iterative LQR (high-level)

For each iteration of the algorithm:

Step 1. Form a time-dependent LQR problem around the current candidate trajectory using local linearization.

Step 2. Compute the optimal policy using subsection 3.4.1.

Step 3. Generate a new series of actions using this policy.

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

Now let's go through the details of each step. We'll use superscripts to denote the iteration of the algorithm. We'll also denote $\bar{s}_0 = \mathbb{E}_{s_0 \sim \mu_0}[s_0]$ as the expected initial state.

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

Step 1: Form a time-dependent LQR problem. At each timestep $h \in [H]$, we use the techniques from section 3.5 to linearize the dynamics and quadratize the cost function around $(\bar{s}_h^i, \bar{a}_h^i)$:

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

Step 2: Compute the optimal policy. We can now solve the time-dependent LQR problem using the Riccati equation from subsection 3.4.1 to compute the optimal policy $\pi_0^i, \ldots, \pi_{H-1}^i$.

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

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

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

Step 4: Compute a better candidate trajectory., Note that we've denoted these actions as $\widetilde{a}_h$ and aren't directly using them for the next iteration $\bar{a}_h^{i+1}$. Rather, we want to *interpolate* between them and the actions from the previous iteration $\bar{a}_0^i, \ldots, \bar{a}_{H-1}^i$. This is so that the cost will *increase monotonically,* since if the new policy turns out to actually be worse, we can stay closer to the previous trajectory. (Can you think of an intuitive example where this might happen?)

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

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

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

The final output of this algorithm is a policy $\pi^{n_{\text{steps}}}$ derived after n_{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).

3.6 Summary

This chapter introduced some approaches to solving different variants of the optimal control problem (3.1.1). We began with the simple case of linear dynamics and an upward-curved quadratic cost. This model is called the LQR and we solved for the optimal policy using dynamic programming. We then extended these results to the more general nonlinear case via local linearization. We finally saw the iterative LQR algorithm for solving nonlinear control problems.

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 (the expected total reward)? 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, or even intractable, to compute exactly! This is because it involves taking an expectation over all possible trajectories τ. Can we rewrite this in a form that's more convenient to implement?

Specifically, suppose there is some distribution over trajectories $\rho(\tau)$ that's easy to sample from (e.g. a database of existing trajectories). We can then rewrite the gradient of objective function, a.k.a. the *policy gradient*, as follows (all gradients are being taken w.r.t. θ):

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

Note that setting $\rho = \rho_\theta$ allows us to express ∇J as an expectation. (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 the following expression for the policy gradient, known as the "REINFORCE" policy gradient:

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

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.

In fact, we can perform one more simplification. Intuitively, the action taken at step t does not affect the reward from previous timesteps, since they're already in the past! You can also show rigorously that this is the case, and that we only need to consider the present and future rewards to calculate the policy gradient:

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

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

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:

$$\nabla J(\theta) = \mathop{\mathbb{E}}_{\tau\sim\rho_\theta}\left[\sum_{t=0}^{H-1} \nabla \log \pi_\theta(a_t|s_t)\left(\left(\sum_{t'=t}^{H-1} r_t\right) - b_t(s_t)\right)\right]. \tag{4.4}$$

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

We can now express the policy gradient as follows. Note that the advantage function effectively replaces the Q-function from Equation 4.3:

$$\nabla J(\theta) = \mathop{\mathbb{E}}_{\tau\sim\rho_\theta}\left[\sum_{t=0}^{T-1} \nabla \log \pi_\theta(a_t|s_t) A_t^{\pi_\theta}(s_t,a_t)\right]. \tag{4.5}$$

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

Definition 4.4.1: Policy gradient with a learned baseline

Require: Learning rate $\eta_0, \ldots, \eta_{K-1}$
Require: Initialization θ^0
for $k = 0, \ldots, K-1$ **do**
Sample N trajectories from π_{θ^k} to estimate a baseline $\widetilde{b}$ such that $\widetilde{b}_h(s) \approx V_h^{\theta^k}(s)$
Sample M trajectories $\tau_0, \ldots, \tau_{M-1} \sim \rho_{\theta^k}$
Compute the policy gradient estimate

$$\widetilde{\nabla}_\theta J(\theta^k) = \frac{1}{M}\sum_{m=0}^{M-1}\sum_{h=0}^{H-1} \nabla \log \pi_{\theta^k}(a_h \mid s_h)(R_h(\tau_m) - \widetilde{b}_h(s_h))$$

Gradient ascent update $\theta^{k+1} \leftarrow \theta^k + \widetilde{\nabla}_\theta J(\theta^k)$
end for

The baseline estimation step can be done using any appropriate supervised learning algorithm. Note that the gradient estimator will be unbiased regardless of the baseline.

4.5 Example 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 perform a "softmax" operation: 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 make use of any structure in the states or actions, so 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 function 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

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

4.5.3 Continuous action spaces

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

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

Exercise: Can you extend the "linear in features" policy to continuous action spaces in a similar way?

4.6 Local policy optimization

4.6.1 Motivation for policy gradient

Recall the policy iteration algorithm discussed in the MDP section: We alternate between these two steps:

- Estimating the Q-function of the current policy
- Updating the policy to be greedy w.r.t. this approximate Q-function.

(Note that we could equivalently estimate the advantage function.)

What advantages does the policy gradient algorithm have over policy iteration? Both policy gradient and policy iteration are iterative algorithms.

To analyze the difference between them, we'll make use of the **performance difference lemma**.

Theorem 4.6.1: Performance difference lemma

Let $\rho_{\pi,s}$ denote the distribution induced by the policy π over trajectories starting in state s.

Given two policies $\pi, \widetilde{pi}$, the PDL allows us to express the difference between their value functions as follows:

$$V_0^{\widetilde{\pi}}(s) - V_0^{\pi}(s) = \mathop{\mathbb{E}}_{\tau \sim \rho_{\widetilde{\pi},s}} \left[\sum_{h=0}^{H-1} A_h^{\pi}(s_h, a_h) \right] \tag{4.6}$$

Some intuition: Recall that $A_h^{\pi}(s,a)$ tells us how much better the action a is in state s than average, supposing actions are chosen according to π. How much better is $\widetilde{\pi}$ than π? To answer this, we break down the trajectory step-by-step. At each step, we compute how much better actions from $\widetilde{\pi}$ are than the actions from π. But this is exactly the average π-advantage, where the expectation is taken over actions from $\widetilde{\pi}$. This is exactly what the PDL describes.

Let's analyze why fitted approaches such as PI don't work as well in the RL setting. To start, let's ask, where *do* fitted approaches work well? They are commonly seen in *supervised learning*, where a prediction rule is fit using some labelled training set, and then assessed on a test set from the same distribution. Does this assumption still hold when doing PI?

Let's consider a single iteration of PI. Suppose the new policy $\widetilde{\pi}$ chooses some action with a negative advantage w.r.t. π. Define $\Delta_\infty = \min_{s \in \mathcal{S}} A_h^{\pi}(s, \widetilde{\pi}(s))$. If this is negative, then the PDL shows that there may exist some state s and time h such that

$$V_h^{\widetilde{\pi}}(s) \geq V_h^{\pi}(s) - H \cdot |\Delta_\infty|.$$

In general, PI cannot avoid particularly bad situations where the new policy $\widetilde{\pi}$ often visits these bad states, causing an actual degradation. It does not enforce that the trajectory distributions ρ_π and $\rho_{\widetilde{\pi}}$ be close to each other. In other words, the "training distribution" that our prediction rule is fitted on, ρ_π, may differ significantly from the "evaluation distribution" $\rho_{\widetilde{\pi}}$ — we must address this issue of *distributional shift*.

How can we enforce that the *trajectory distributions* do not change much at each step? In fact, policy gradient already does this to a small extent: Supposing that the mapping from parameters to trajectory distributions is relatively smooth, then, by adjusting the parameters a small distance from the current iterate, we end up at a new policy with a similar trajectory distribution. But this is not very rigorous, and in practice the parameter-to-distribution mapping may not be smooth. Can we constrain the distance between the resulting distributions more explicitly? This brings us to the next two methods: **trust region policy optimization** (TRPO) and the **natural policy gradient** (NPG).

4.6.2 Trust region policy optimization

TRPO is another iterative algorithm for policy optimization. It is similar to policy iteration, except we constrain the updated policy to be "close to" the current policy in terms of the trajectory distributions they induce.

To formalize "close to", we typically use the **Kullback-Leibler divergence (KLD)**:

Definition 4.6.1: Kullback-Leibler divergence

For two PDFs p, q,

$$\mathrm{KL}\left(p \parallel q\right) := \mathop{\mathbb{E}}_{x\sim p}\left[\log \frac{p(x)}{q(x)}\right] \tag{4.7}$$

This can be interpreted in many different ways, many stemming from information theory. Note that $\mathrm{KL}\left(p \parallel q\right) = 0$ if and only if $p = q$. Also note that it is generally not symmetric.

Additionally, rather than estimating the Q-function of the current policy, we can use the RHS of the Performance Difference Lemma (4.6.1) as our optimization target.

Definition 4.6.2: Trust region policy optimization (exact)

Require: Trust region radius δ
  Initialize θ^0
  for $k = 0, \ldots, K-1$ **do**
    $\theta^{k+1} \leftarrow \arg\max_\theta \mathbb{E}_{s_0,\ldots,s_{H-1}\sim\pi^k}\left[\sum_h \mathbb{E}_{a_h\sim\pi_\theta(s_h)} A^{\pi^k}(s_h, a_h)\right]$ ▷ See below
    where $\mathrm{KL}\left(\rho_{\pi^k} \parallel \rho_{\pi_\theta}\right) \le \delta$
  end for
  return π^K

Note that the objective function is not identical to the r.h.s. of the Performance Difference Lemma. Here, we still use the *states* sampled from the old policy, and only use the *actions* from the new policy. This is because it would be computationally infeasible to sample entire trajectories from π_θ as we are optimizing over θ. This approximation is also reasonable in the sense that it matches the r.h.s. of the Performance Difference Lemma to first order in θ. (We will elaborate more on this later.)

Both the objective function and the KLD constraint involve a weighted average over the space of all trajectories. This is intractable in general, so we need to estimate the expectation. As before, we can do this by taking an empirical average over samples from the trajectory distribution. However, the inner expectation over $a_h \sim \pi_\theta$ involves the optimizing variable θ, and we'd like an expression that has a closed form in terms of θ to make optimization tractable. Otherwise, we'd need to resample many times each time we made an update to θ. To address this, we'll use a common technique known as **importance sampling**.

Definition 4.6.3: Importance sampling

Suppose we want to estimate $\mathbb{E}_{x\sim\widetilde{p}}[f(x)]$. However, $\widetilde{p}$ is difficult to sample from, so we can't take an empirical average directly. Instead, there is some other distribution p that is easier to sample from, e.g. we could draw samples from an existing dataset, as in the case of **offline RL**.

Then note that

$$\mathop{\mathbb{E}}_{x\sim\widetilde{p}}[f(x)] = \mathop{\mathbb{E}}_{x\sim p}\left[\frac{\widetilde{p}(x)}{p(x)}f(x)\right]$$

so, given i.i.d. samples $x_0, \ldots, x_{N-1} \sim p$, we can construct an unbiased estimate of $\mathbb{E}_{x\sim\widetilde{p}}[f(x)]$ by *reweighting* these samples according to the likelihood ratio $\widetilde{p}(x)/p(x)$:

$$\frac{1}{N}\sum_{n=0}^{N-1}\frac{\widetilde{p}(x_n)}{p(x_n)}f(x_n)$$

Doesn't this seem too good to be true? If there were no drawbacks, we could use this to estimate *any* expectation of any function on any arbitrary distribution! The drawback is that the variance may be very large due to the likelihood ratio term. If the sampling distribution p assigns low probability to any region where $\widetilde{p}$ assigns high probability, then the likelihood ratio will be very large and cause the variance to blow up.

Applying importance sampling allows us to estimate the TRPO objective as follows:

Definition 4.6.4: Trust region policy optimization (implementation)

Initialize θ^0
for $k = 0, \ldots, K-1$ **do**
Sample N trajectories from ρ^k to learn a value estimator $\widetilde{b}_h(s) \approx V_h^{\pi^k}(s)$
Sample M trajectories $\tau_0, \ldots, \tau_{M-1} \sim \rho^k$

$$\theta^{k+1} \leftarrow \arg\max_\theta \frac{1}{M}\sum_{m=0}^{M-1}\sum_{h=0}^{H-1}\frac{\pi_\theta(a_h \mid s_h)}{\pi^k(a_h \mid s_h)}[R_h(\tau_m) - \widetilde{b}_h(s_h)]$$
$$\text{where } \sum_{m=0}^{M-1}\sum_{h=0}^{H-1}\log\frac{\pi_k(a_h^m \mid s_h^m)}{\pi_\theta(a_h^m \mid s_h^m)} \le \delta$$

end for

4.6.3 Natural policy gradient

Instead, we can solve an approximation to the TRPO optimization problem. This will link us back to the policy gradient from before. We take a first-order approximation to the objective function and a second-order approximation to the KLD constraint. This results in the optimization problem

$$\begin{aligned}&\max_\theta \nabla_\theta J(\pi_{\theta^k})^\top(\theta - \theta^k)\\ &\text{where } \frac{1}{2}(\theta-\theta^k)^\top F_{\theta^k}(\theta-\theta^k) \le \delta\end{aligned} \tag{4.8}$$

where F_{θ^k} is the **Fisher information matrix** defined below.

Definition 4.6.5: Fisher information matrix

Let p_θ denote a parameterized distribution. Its Fisher information matrix F_θ can be defined equivalently as:

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

Recall that the Hessian of a function describes its curvature: That is, for a vector $\delta \in \Theta$, the quantity $\delta^\top F_\theta \delta$ describes how rapidly the negative log-likelihood changes if we move by δ.

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

$$F_\theta = \mathop{\mathbb{E}}_{\tau\sim\rho_\theta}\left[\sum_{h=0}^{H-1}(\nabla \log \pi_\theta(a_h \mid s_h))(\nabla \log \pi_\theta(a_h \mid s_h))^\top\right] \tag{4.9}$$

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

This is a convex optimization problem, and so we can find the global optima by setting the gradient of the Lagrangian to zero:

$$\begin{aligned} \mathcal{L}(\theta,\eta) &= \nabla_\theta J(\pi_{\theta^k})^\top(\theta-\theta^k) - \eta\left[\frac{1}{2}(\theta-\theta^k)^\top F_{\theta^k}(\theta-\theta^k) - \delta\right] \\ \nabla_\theta \mathcal{L}(\theta^{k+1},\eta) &= 0 \\ \nabla_\theta J(\pi_{\theta^k}) &= \eta F_{\theta^k}(\theta^{k+1}-\theta^k) \\ \theta^{k+1} &= \theta^k + \eta F_{\theta^k}^{-1}\nabla_\theta J(\pi_{\theta^k}) \\ \text{where } \eta &= \sqrt{\frac{\delta}{\nabla_\theta J(\pi_{\theta^k})^\top F_{\theta^k}\nabla_\theta J(\pi_{\theta^k})}} \end{aligned}$$

Definition 4.6.6: Natural policy gradient

Require: Learning rate $\eta > 0$
  Initialize θ^0
  for $k = 0, \ldots, K-1$ **do**
    Estimate the policy gradient $\widehat{g} \approx \nabla_\theta J(\pi_{\theta^k})$ ▷ See (4.5)
    Estimate the Fisher information matrix $\widehat{F} \approx F_{\theta^k}$ ▷ See (4.9)
    $\theta^{k+1} \leftarrow \theta^k + \eta\widehat{F}^{-1}\widehat{g}$ ▷ Natural gradient update
  end for

How many trajectory samples do we need to accurately estimate the Fisher information

matrix? As a rule of thumb, the sample complexity should scale with the dimension of the parameter space. This makes this approach intractable in the deep learning setting where we might have a very large number of parameters.

For some intuition: The typical gradient descent algorithm treats the parameter space as "flat", treating the objective function as some black box value. However, in the case here where the parameters map to a *distribution*, using the natural gradient update is equivalent to optimizing over distribution space rather than distribution space.

Example 4.6.1: Natural gradient on a simple problem

Let's step away from reinforcement learning specifically and consider the following optimization problem over Bernoulli distributions $\pi \in \Delta(\{0, 1\})$:

$$J(\pi) = 100 \cdot \pi(1) + 1 \cdot \pi(0)$$

Clearly the optimal distribution is the constant one $\pi(1) = 1$. Suppose we optimize over the parameterized family $\pi_\theta(1) = \frac{\exp(\theta)}{1+\exp(\theta)}$. Then our optimization algorithm should set θ to be unboundedly large. Then the vanilla gradient is

$$\nabla_\theta J(\pi_\theta) = \frac{99 \exp(\theta)}{(1 + \exp(\theta))^2}.$$

Note that as $\theta \to \infty$ that the increments get closer and closer to 0. However, if we compute the Fisher information scalar

$$\begin{aligned} F_\theta &= \mathop{\mathbb{E}}_{x \sim \pi_\theta}[(\nabla_\theta \log \pi_\theta(x))^2] \\ &= \frac{\exp(\theta)}{(1 + \exp(\theta))^2} \end{aligned}$$

resulting in the natural gradient update

$$\begin{aligned} \theta^{k+1} &= \theta^k + \eta F_{\theta^k}^{-1} \nabla_\theta J(\theta^k) \\ &= \theta^k + 99\eta \end{aligned}$$

which increases at a constant rate, i.e. improves the objective more quickly than vanilla gradient ascent.

4.6.4 Proximal policy optimization

Can we improve on the computational efficiency of the above methods?

We can relax the TRPO objective in a different way: Rather than imposing a hard constraint on the KL distance, we can instead impose a *soft* constraint by incorporating it into the objective:

Definition 4.6.7: Proximal policy optimization (exact)

Require: Regularization parameter λ
Initialize θ^0
for $k = 0, \dots, K-1$ **do**
$\theta^{k+1} \leftarrow \arg\max_\theta \mathbb{E}_{s_0,\dots,s_{H-1}\sim\pi^k}\left[\sum_h \mathbb{E}_{a_h\sim\pi_\theta(s_h)} A^{\pi^k}(s_h, a_h)\right] - \lambda \mathrm{KL}\left(\rho_{\pi^k} \parallel \rho_{\pi_\theta}\right)$
end for
return θ^K

Note that like the original TRPO algorithm 4.6.2, PPO is not gradient-based; rather, at each step, we try to maximize local advantage relative to the current policy.

Let us now turn this into an implementable algorithm, assuming we can sample trajectories from π_{θ^k}.

Let us simplify the $\mathrm{KL}\left(\rho_{\pi^k} \parallel \rho_{\pi_\theta}\right)$ term first. Expanding gives

$$\begin{aligned}
\mathrm{KL}\left(\rho_{\pi^k} \parallel \rho_{\pi_\theta}\right) &= \mathop{\mathbb{E}}_{\tau\sim\rho_{\pi^k}}\left[\log \frac{\rho_{\pi^k}(\tau)}{\rho_{\pi_\theta}(\tau)}\right] \\
&= \mathop{\mathbb{E}}_{\tau\sim\rho_{\pi^k}}\left[\sum_{h=0}^{H-1} \log \frac{\pi^k(a_h \mid s_h)}{\pi_\theta(a_h \mid s_h)}\right] && \text{state transitions cancel} \\
&= \mathop{\mathbb{E}}_{\tau\sim\rho_{\pi^k}}\left[\sum_{h=0}^{H-1} \log \frac{1}{\pi_\theta(a_h \mid s_h)}\right] + c
\end{aligned}$$

where c is some constant relative to θ.

As we did for TRPO (4.6.4), we can use importance sampling (4.6.3) to rewrite the inner expectation. Combining the expectations together, this gives the (exact) objective

$$\max_\theta \mathop{\mathbb{E}}_{\tau\sim\rho_{\pi^k}}\left[\sum_{h=0}^{H-1}\left(\frac{\pi_\theta(a_h \mid s_h)}{\pi^k(a_h \mid s_h)} A^{\pi^k}(s_h, a_h) - \lambda \log \frac{1}{\pi_\theta(a_h \mid s_h)}\right)\right]$$

Now we can use gradient ascent on the parameters θ until convergence to maximize this function, completing a single iteration of PPO (i.e. $\theta^{k+1} \leftarrow \theta$).

Chapter 5

Fitted Dynamic Programming

Contents

5.1 Introduction

Contents

Chapter 6

Exploration in MDPs

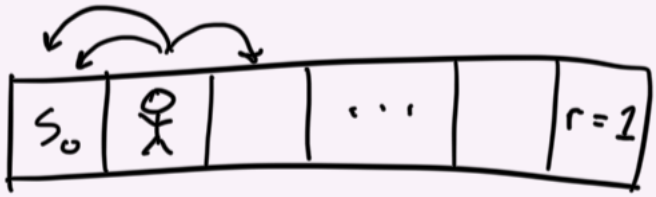
6.1 Introduction

In the chapter on fitted DP (5), we explored algorithms for finding the optimal value and policy in an MDP when the transition and reward functions are unknown. However, we swept the issue of *exploration* under the hood. Namely, our algorithms might easily *overfit* to certain areas of the state space, missing out on possible better paths. This issue is especially relevant in **sparse reward** problems where reward might not be achieved until after many steps, and algorithms which do not *systematically* explore new states may entirely fail to learn anything meaningful.

For example, policy gradient algorithms require *signal* in the gradient to learn. In other words, if we never observe any reward, the gradient will always be zero, and the policy will never improve.

Example 6.1.1: Sparse Reward MDP

Here's a simple example of an MDP with sparse reward:



There are $|\mathcal{S}|$ states. The agent starts in the leftmost state. In every state, there are three possible actions, two of which move the agent left and one which moves the agent right. The reward function assigns $r = 1$ to the rightmost cell.

How can we address this issue?

Let's start by assuming that the MDP is *deterministic*, that is, taking action a in state s will always take you to the state $P(s, a) \in \mathcal{S}$. We will save issues of randomness for later.

Then, one algorithm to trade off exploration and exploitation is:

1. **Explore:** Visit every possible state-action pair to fully understand the MDP.
2. **Exploit:** Now the MDP is fully known, so we can use a planning algorithm like policy iteration (2.3.3) to solve for the optimal policy.

Definition 6.1.1: Explore-then-exploit (for deterministic MDPs)

We'll keep a set K of all the (s, a, r, s') pairs we've observed. Each episode, we'll choose an unseen pair (s, a) for which the reward $r(s, a)$ and the next state $s' = P(s, a)$ are unknown, and take the shortest path there.

$K \leftarrow \emptyset$
while $\exists(s, a)$ s.t. there is no $(s, a, r, s') \in K$ **do**
  Using our known transitions K, compute the shortest path to (s, a)
  Take the shortest path to (s, a)
  $K \leftarrow K \cup \{(s, a)\}$
end while
Compute the optimal policy $\pi^\star$ in the MDP K (e.g. using policy iteration).
return $\pi^\star$.

We leave it to the reader to design an efficient implementation of the shortest-path algorithm.

Theorem 6.1.1: Performance of explore-then-exploit

As long as every state can be reached from s_0 within a single episode, i.e. $|\mathcal{S}| \leq H$, this will eventually be able to explore all $|\mathcal{S}||\mathcal{A}|$ state-action pairs, adding one new transition per episode/trajectory.

We can measure the performance using the **regret** across episodes. That is, let K_t denote the value of K at iteration t, and let us compare the value of the optimal policy π_t derived from K_t with the value of the *true* optimal policy $\pi^\star$.

As we described above, we know that it will take at most SA iterations to explore the entire MDP, after which $\pi_t = \pi^\star$. For the π_t up until then, however, we can construct a very loose bound as follows. At each interaction, the reward from π_t might differ from $\pi^\star$ by at most 1. Since an episode is H steps long, this means the value of policy π_t will differ from that of $\pi^\star$ by at most H. So,

$$\sum_{t=1}^{T} V_0(\pi^\star) - V_0(\pi_t) \leq |\mathcal{S}||\mathcal{A}|H.$$

6.2 Treating an unknown MDP as a MAB

We also explored the exploration-exploitation tradeoff in the chapter on multi-armed bandits (**??**). Recall the overall framework of MAB: We have K arms, each of which has an unknown reward distribution, and we want to learn which of the arms is *optimal*, i.e. gives the highest mean reward.

One algorithm that struck a good balance between exploration and exploitation was the **upper confidence bound** algorithm (1.6.1), where for each arm we construct a *confidence interval* for its true mean award, and then choose the arm that achieves the highest upper confidence bound on its mean reward. In summary,

$$k_{t+1} \leftarrow \arg\max_{k \in [K]} \frac{S_t^k}{N_t^k} + \sqrt{\frac{\ln(2t/\delta)}{2N_t^k}}$$

where N_t^k indicates the number of times arm k has been pulled up until time t, S_t^k indicates the total reward obtained by pulling arm k up until time t, and $\delta > 0$ controls the width of the confidence interval. How might we extend UCB to the MDP case?

Let us formally describe an unknown MDP as an MAB problem. In an unknown MDP, we want to learn which *policy* is optimal. So if we want to apply MAB techniques to solving an MDP, it makes sense to think of *arms* as *policies*. This gives us $(|\mathcal{A}|^{\mathcal{S}})^H$ arms for deterministic policies in a finite MDP. Then, "pulling" arm π corresponds to using π to act through a trajectory in the MDP, and observing the total reward.

Exercise: Which quantity that we have seen so far equals the mean reward from arm π?

Recall that UCB incurs regret $\widetilde{O}(\sqrt{TK})$, where T is the number of pulls and K is the number of arms. Substituting in the values above, we see that treating policies as arms and running UCB incurs regret

$$\widetilde{O}(|\mathcal{A}|^{|\mathcal{S}|H/2}N)$$

where N is the number of trajectories we get to observe. This scales *exponentially* in $|\mathcal{S}|$ and H, which quickly becomes intractable. Notably, this method doesn't consider the information that we gain across different policies. We can illustrate this with the following example:

Example 6.2.1: Treating an MDP as a MAB is ineffective

Consider a "coin MDP" with two states "heads" and "tails", two actions "Y" and "N", and a time horizon of $H = 2$. The state transition flips the coin, and doesn't depend on the action. The reward only depends on the action: Taking action Y gives reward 1, and taking action N gives reward 0.

Suppose we collect a data from the two constant policies $\pi_{\mathsf{Y}}(s) = \mathsf{Y}$ and $\pi_{\mathsf{N}}(s) = \mathsf{N}$. Now we want to learn about policy $\widetilde{\pi}$ that takes action Y and then N. Do we need to collect data about $\widetilde{\pi}$? No! We can infer its behaviour on timestep 1 from our data on policy π_{Y} and its behaviour on timestep 2 from our data on policy π_{N}. But if we treat the

MDP as a bandit, we treat $\widetilde{\pi}$ as a new arm about which we know nothing.

6.3 UCB-VI

One way to frame the UCB algorithm is that, when choosing arms, we optimize over a *proxy reward* that is the sum of the estimated mean reward and an exploration term.

Can we extend this idea to the case of an unknown MDP $\mathcal{M}^?$? That is, can we model a proxy MDP $\widetilde{\mathcal{M}}$ with a reward function that encourages exploration, and then use DP to solve for the optimal policy in $\widetilde{\mathcal{M}}$? This brings us to the **UCB-VI** algorithm.

Assumptions: For simplicity, here we assume the reward function of $M_?$ is known, so we only need to model the state transitions. We will consider the more general case of a **time-varying** MDP, where the transition and reward functions can change over time. We take the convention that P_h is the distribution of $s_{h+1} \mid s_h, a_h$ and r_h is applied to s_h, a_h.

Definition 6.3.1: UCB-VI

For $t \in [T]$:

1. **Modelling:** We use previous data to model the transitions $\widehat{P}_0, \dots, \widehat{P}_{H-2}$. We design a reward bonus $b_h(s,a) \in \mathbb{R}$ to encourage exploration, analogous to the UCB term.
2. **Optimistic planning:** Using DP, We solve for the optimal policy π_h in the modelled MDP
$$\widetilde{M} = (\mathcal{S}, \mathcal{A}, \{\widehat{P}_h\}_{h\in[H-1]}, \{r_h + b_h\}_{h\in[H-1]}, H).$$
Then we use π_h to collect a new trajectory, and repeat.

Note that the bonuses also *propagate backwards* via the DP algorithm. This effectively enables us to *plan to explore* unknown states.

We detail each of these steps below.

6.3.1 Modeling the transitions

Let $\mathcal{D}_h^t$ denote the dataset of transitions collected at timestep h from the first t trajectories. That is, $\mathcal{D}_h^t = \{s_h^i, a_h^i, s_{h+1}^i\}_{i\in[t]}$.

We seek to approximate $P_h(s_{h+1} \mid s_h, a_h) = \frac{\mathbb{P}(s_h, a_h, s_{h+1})}{\mathbb{P}(s_h, a_h)}$. We can estimate these using their sample probabilities from the dataset. That is, define

$$N_h^t(s,a,s') := \sum_{i=0}^{t-1} \mathbf{1}\left\{s_h^i = s, a_h^i = a, s_{h+1}^i = s'\right\}$$

$$N_h^t(s,a) := \sum_{s' \in \mathcal{S}} N_h^t(s,a,s')$$

Then we can model

$$\widehat{P}_h^t(s' \mid s,a) = \frac{N_h^t(s,a,s')}{N_h^t(s,a)}.$$

Remark: Note that this is also a fairly naive estimate, and doesn't assume any underlying structure of the MDP. Thus it would be considered a *nonparametric* model. We'll see how to improve this in the following section, and use underlying structure to improve our estimates.

6.3.2 Reward bonus

To motivate the reward bonus term $b_h^t(s,a)$, recall how we designed the reward bonus term for UCB:

1. We used Hoeffding's inequality to bound, with high probability, how far the sample mean $\widehat{\mu}_k^t$ deviated from the true mean μ_k.
2. By inverting this inequality, we obtained a $(1-\delta)$-confidence interval for the true mean, centered at our estimate.
3. To make this bound *uniform* across all timesteps $t \in [T]$, we applied the union bound and multiplied δ by a factor of T.

We'd like to do the same for UCB-VI, and construct the bonus term such that $V_h^\star \le \widehat{V}_h^t(s)$ with high probability. However, our construction will be more complex than the MAB case, since $\widehat{V}_h^t(s)$ depends on $b_h^t(s,a)$ implicitly via the DP algorithm. We claim that the bonus term that satisfies this property is

$$b_h^t(s,a) = 2H\sqrt{\frac{\log(|\mathcal{S}||\mathcal{A}|HT/\delta)}{N_h^t(s,a)}}.$$

We will only provide a heuristic sketch of the proof; see [1, Section 7.3] for a full proof.

Derivation 6.3.1: UCB-VI reward bonus construction

We aim to show that, with high probability,

$$V_h^\star(s) \le \widehat{V}_h^t(s) \quad \forall t \in [T], h \in [H].$$

We'll do this by bounding the error incurred at each step of DP. Recall that DP solves for $\widehat{V}_h^t(s)$ recursively as follows:

$$\widehat{V}_h^t(s) = \max_{a \in \mathcal{A}} \left[\widetilde{r}_h^t(s,a) + \mathop{\mathbb{E}}_{s' \sim \widehat{P}_h^t(\cdot|s,a)} \left[\widehat{V}_{h+1}^t(s') \right] \right]$$

where $\widetilde{r}_h^t(s,a) = r_h(s,a) + b_h^t(s,a)$ is the reward function of our modelled MDP $\widetilde{M}^t$. There are two possible sources of error:

1. The value functions $\widehat{V}_{h+1}^t$ v.s. $V_{h+1}^\star$
2. The transition probabilities $\widehat{P}_h^t$ v.s. $P_h^?$.

For the former, we can simply bound the difference by H, assuming that the rewards are within $[0,1]$. Now, all that is left is to bound the error from the transition probabilities, and we can combine these two bounds with the triangle inequality.

First, for a fixed s, a, h, t, we aim to bound the error caused by the state transitions:

$$\text{error} = \left| \mathop{\mathbb{E}}_{s' \sim \widehat{P}_h^t(\cdot|s,a)} \left[V_{h+1}^\star(s') \right] - \mathop{\mathbb{E}}_{s' \sim P_h^?(\cdot|s,a)} \left[V_{h+1}^\star(s') \right]. \right| \tag{6.1}$$

Note that expanding out the definition of $\widehat{P}_h^t$ gives

$$\begin{aligned}
\mathop{\mathbb{E}}_{s' \sim \widehat{P}_h^t(\cdot|s,a)} \left[V_{h+1}^\star(s') \right] &= \sum_{s' \in \mathcal{S}} \frac{N_h^t(s,a,s')}{N_h^t(s,a)} V_{h+1}^\star(s') \\
&= \frac{1}{N_h^t(s,a)} \sum_{i=0}^{t-1} \sum_{s' \in \mathcal{S}} \mathbf{1}\left\{ (s_h^i, a_h^i, s_{h+1}^i) = (s,a,s') \right\} V_{h+1}^\star(s') \\
&= \frac{1}{N_h^t(s,a)} \sum_{i=0}^{t-1} \underbrace{\mathbf{1}\left\{ (s_h^i, a_h^i) = (s,a) \right\} V_{h+1}^\star(s_{h+1}^i)}_{X^i}
\end{aligned}$$

since the terms where $s' \neq s_{h+1}^i$ vanish.

Now, in order to apply Hoeffding's inequality, we would like to express the second term in (6.1) as a sum over t random variables as well. We will do this by redundantly averaging over all desired trajectories (i.e. where we visit state s and action a at time h):

$$\begin{aligned}
\mathop{\mathbb{E}}_{s' \sim P_h^?(\cdot|s,a)} \left[V_{h+1}^\star(s') \right] &= \sum_{s' \in \mathcal{S}} P_h^?(s' \mid s,a) V_{h+1}^\star(s') \\
&= \sum_{s' \in \mathcal{S}} \frac{1}{N_h^t(s,a)} \sum_{i=0}^{t-1} \mathbf{1}\left\{ (s_h^i, a_h^i) = (s,a) \right\} P_h^?(s' \mid s,a) V_{h+1}^\star(s') \\
&= \frac{1}{N_h^t(s,a)} \sum_{i=0}^{t-1} \mathop{\mathbb{E}}_{s_{h+1}^i \sim P_h^?(\cdot|s_h^i, a_h^i)} X^i.
\end{aligned}$$

Now we can apply Hoeffding's inequality to obtain that, with probability at least $1-\delta$,

$$\text{error} = \left| \frac{1}{N_h^t(s,a)} \sum_{i=0}^{t-1} \left(X^i - \mathop{\mathbb{E}}_{s_{h+1}^i \sim P_h^?(\cdot|s_h^i, a_h^i)} X^i \right) \right| \leq 2H \sqrt{\frac{\ln(1/\delta)}{N_h^k(s,a)}}.$$

Applying a union bound over all $s \in \mathcal{S}, a \in \mathcal{A}, t \in [T], h \in [H]$ gives the $b_h^t(s,a)$ term above.

6.3.3 Performance of UCB-VI

How exactly does UCB-VI strike a good balance between exploration and exploitation? In UCB, the bonus exploration term was fairly interpretable, but now we must consider the value function of the policy returned by UCB-VI at iteration t, that is, $V_h^{\pi^t}(s)$.

Recall we constructed b_h^t so that, with high probability, $V_h^\star(s) \le \widehat{V}_h^t(s)$ and so

$$V_h^\star(s) - V_h^{\pi^t}(s) \le \widehat{V}_h^t(s) - V_h^{\pi^t}(s).$$

If the r.h.s. is *small*, this implies that the l.h.s. difference is also small, i.e. that π^t is *exploiting* actions that are giving high reward.

If the r.h.s. is *large*, then we have overestimated the value: π^t, the optimal policy of $\widetilde{M}$, does not perform well in the true environment $M^?$. This indicates that either some $b_h^t(s,a)$ must be large, which corresponds to an action taken to *explore* the environment, or some $\widehat{P}_h^t(\cdot \mid s,a)$ is inaccurate; but notice that this correlates with a low visit count of (s,a), and thus $b_h^t(s,a)$ being large.

It turns out that UCB-VI achieves a per-episode regret of

$$\mathbb{E}\left[\sum_{t=0}^{T-1}\left(V_0^\star(s_0) - V_0^{\pi^t}(s_0)\right)\right] = \widetilde{O}(H^2\sqrt{|\mathcal{S}||\mathcal{A}|T})$$

Comparing this to the UCB regret bound $\widetilde{O}(\sqrt{TK})$, we see that we've reduced the number of effective arms from

For some more intuition, consider how many episodes it takes for us to achieve an average regret of 1. Ignoring the H^2 factor (we'll see later why this is allowed), the average regret is

$$\frac{1}{T}\,\mathbb{E}[\text{Regret}_T] = \widetilde{O}\left(\sqrt{\frac{|\mathcal{S}||\mathcal{A}|}{T}}\right),$$

and for this to be constant, it would take $T = \Omega(|\mathcal{S}||\mathcal{A}|)$ steps. Note the number of entries of the $Q^\star$ matrix is of the same order. But note that the transition matrix has $|\mathcal{S}|^2|\mathcal{A}|$ entries. This shows that it's possible to achieve low regret, and achieve a near-optimal policy, while only understanding a $1/|\mathcal{S}|$ fraction of the world.

ssion of H^2

6.4 Linear MDPs

Above, when estimating the transition matrix $\widehat{P}_h^t$, we said that we can do in the case where we don't know anything about the structure of the MDP. But in many cases, for large or continuous $|\mathcal{S}|$ and $|\mathcal{A}|$, even some polynomial factor in $|\mathcal{S}|$ and $|\mathcal{A}|$ will become intractable. Instead, we're going to look at **linear MDPs**: an example of a *parameterized* MDP where the rewards and state transitions depend only on some *low-dimensional* parameter space.

Definition 6.4.1: Linear MDP

We assume that the transition probabilities and rewards are *linear* in some feature vector $\phi(s,a) \in \mathbb{R}^d$:

$$P_h(s' \mid s,a) = \phi(s,a)^\top \mu_h^\star(s')$$
$$r_h(s,a) = \phi(s,a)^\top \theta_h^\star$$

Note that we can also think of $P_h(\cdot \mid s,a) = \mu_h^\star$ as an $|\mathcal{S}| \times d$ matrix, and think of $\mu_h^\star(s')$ as indexing into the s'-th row of this matrix (treating it as a column vector). Thinking of $V^\star$ as an $|\mathcal{S}|$-dimensional vector, this allows us to write

$$\mathop{\mathbb{E}}_{s' \sim P_h(\cdot|s,a)}[V^\star(s)] = (\mu_h^\star \phi(s,a))^\top V^\star.$$

The ϕ feature mapping could be designed to capture interactions between the state s and action a. In this book, we'll assume that the feature map $\phi : \mathcal{S} \times \mathcal{A} \to \mathbb{R}^d$ and the reward $\theta_h^\star$ are known.

Let's see what value iteration looks like in a linear MDP.

6.4.1 UCB-VI in a linear MDP

First, value iteration can be described as follows:

We initialize $V_H^\star(s) = 0 \forall s$. Then we iterate:

$$\begin{aligned}
Q_h^\star(s,a) &= r_h(s,a) + \mathop{\mathbb{E}}_{s' \sim P_h(\cdot|s,a)}[V_{h+1}^\star(s')] \\
&= \phi(s,a)^\top \theta_h^\star + (\mu_h^\star \phi(s,a))^\top V_{h+1}^\star \\
&= \phi(s,a)^\top \underbrace{(\theta_h^\star + (\mu_h^\star)^\top V_{h+1}^\star)}_{w_h} \\
V_h^\star(s) &= \max_a Q_h^\star(s,a) \\
\pi_h^\star(s) &= \arg\max_a Q_h^\star(s,a)
\end{aligned}$$

Now, we can use techniques from **supervised learning** to model the unknown MDP.

Let us write $\delta_s \in \mathcal{S}$ as a one-hot vector in $\mathbb{R}^{|\mathcal{S}|}$, with a 1 in the s-th entry and 0 everywhere else. Note that

$$\mathop{\mathbb{E}}_{s' \sim P_h(\cdot|s,a)}[\delta_{s'}] = P_h(\cdot \mid s,a) = \mu_h^\star \phi(s,a).$$

Recall that **supervised learning** is precisely useful for estimating conditional expectations by minimizing mean squared error. Furthermore, since the expectation here is linear in terms of

(some function of) the observed quantities s, a, we can directly apply least-squares multi-target linear regression (with a regularization term) to construct the estimate

$$\widehat{\mu} = \operatorname*{arg\,min}_{\mu \in \mathbb{R}^{|\mathcal{S}| \times d}} \sum_{t=0}^{T-1} \|\mu\phi(s_h^i, a_h^i) - \delta_{s_{h+1}^i}\|_2^2 + \lambda \|\mu\|_{\mathsf{F}}^2.$$

This has a well-known closed-form solution:

$$\widehat{\mu}^\top = (A_h^t)^{-1} \sum_{i=0}^{t-1} \phi(s_h^i, a_h^i) \delta_{s_{h+1}^i}^\top$$

$$\text{where} \quad A_h^t = \sum_{i=0}^{t-1} \phi(s_h^i, a_h^i) \phi(s_h^i, a_h^i)^\top + \lambda I$$

We can directly plug in this estimate into $\widehat{P}_h^t(\cdot \mid s, a) = \widehat{\mu}_h^t \phi(s, a)$.

Now, to design the reward bonus, we can't apply Hoeffding anymore, since the terms no longer involve sample means of bounded random variables; instead, we're incorporating information across different states and actions. Rather, we can construct an upper bound using *Chebyshev's inequality*.

Derivations

6.5 Natural policy gradient

The TRPO objective is

$$\max_{\theta} \mathop{\mathbb{E}}_{s_0,\ldots,s_{H-1}\sim\rho_{\theta^k}}$$

finish derivat
from 2023-1
lecture

Bibliography

[1] Alekh Agarwal et al. "Reinforcement Learning: Theory and Algorithms". In: (Jan. 31, 2022), p. 205. URL: https://rltheorybook.github.io/rltheorybook_AJKS.pdf.

[2] T. L Lai and Herbert Robbins. "Asymptotically Efficient Adaptive Allocation Rules". In: *Advances in Applied Mathematics* 6.1 (1985), pp. 4–22. ISSN: 0196-8858. DOI: 10.1016/0196-8858(85)90002-8. URL: https://www.sciencedirect.com/science/article/pii/0196885885900028 (visited on 10/23/2023).