ESE 650: Learning in Robotics

Spring 2024

Syllabus

Instructor

Pratik Chaudhari pratikac@seas.upenn.edu

Teaching Assistants

Harold Lian

Yifan Fei

Dhruv Parekh

Zhaojin Sun

Xijie Jiao

Beiming Li

Yiming Huang

Amila Mujkic

Yifei Shao

Dexter Ong

June 4, 2024

Contents

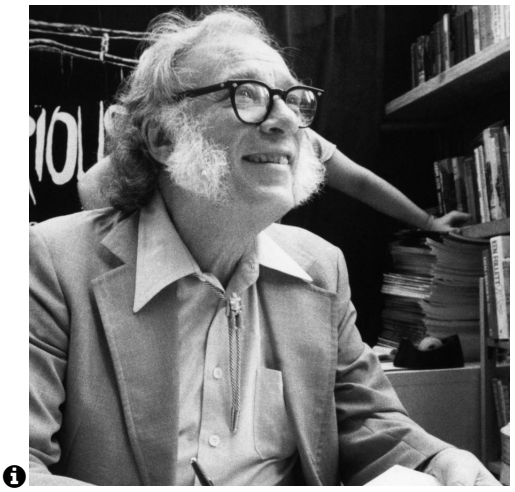
Chapter 1

What is Robotics?

Reading

1. Computing machinery and intelligence, Turing (2009)
2. Thrun Chapter 1
3. Barfoot Chapter 1

The word *robotics* was first used by a Czech writer Karel Capek in a play named "Rossum's Universal Robots" where the owner of this company Mr. Rossum builds robots, i.e., agents who do forced labor, and effectively an artificial man. The word was popularized by Issac Asimov in one of his short stories named Liar!. This is about a *robot* named RB-34 which, through a manufacturing fault, happens to be able to read the minds of humans around it. Around 1942, Isaac Asimov started using the word robot in his writings. This is also when he introduced the Three Laws of Robotics as the theme for how robots would interact with others in his stories/books. These are as follows.



1. A robot may not injure a human being or, through inaction, allow a human being to come to harm.
2. A robot must obey the orders given it by human beings except where such orders would conflict with the First Law.
3. A robot must protect its own existence as long as such protection does not conflict with the First or Second Law.

Asimov would go on to base his stories on the counter-intuitive ways in which robots could apply these laws. In this case, RB-34 adheres to the First Law and in order to not hurt the feelings of humans and make them happy, it deliberately lies to them. It tells the *robopsychologist* Susan Colins that one of her co-workers is infatuated with her. However, when she confronts RB-34 later by pointing out that lying to people can end up hurting them, the robot experiences a logical conflict within its laws and becomes unresponsive.

This is, after all, science fiction but these laws give us insight into what robots are. Let's see what modern *roboticists* have to say.

> "Robotics is the science of perceiving and manipulating the physical world through computer-controlled mechanical devices." — Sebastian Thrun in Probabilistic Robotics

> "EVERYTHING comes together in the field of robotics. The design of an autonomous robots involves: the choice of the mechanical platform, the choice of actuators, the choice of sensors, the choice of the energy source, the choices of algorithms (perception, planning, and control). Each of these subproblems corresponds to a discipline in itself, with its design trade-offs of achievable performance vs limited resources." — Andrea Censi in Censi (2016).

I find the Third Law really insightful to understand intelligence as well. Let us define intelligence as the ability of an organism to survive[1]. We will all agree that trees are less intelligent than animals, an ant is less than intelligent than a dog, which is less intelligent than a human. A program like AlphaGo is not very intelligent because you can disable it by simply switching it off. A key indicator of intelligence is the ability to sense possible harm and take actions to change the outcome.

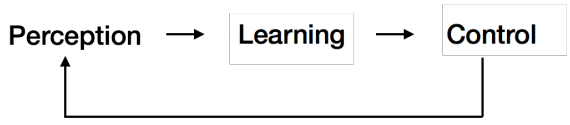
Robotics is Embodied Artificial Intelligence.

> A robot is a machine that senses its environment using sensors, interacts with this environment using actuators to perform a given task and does so efficiently using previous experience of performing similar tasks.

We will cover the fundamentals of these three aspects of robotics: perception, planning and learning.

1.1 Perception-Learning-Control

Perception refers to the sensory mechanisms to gain information about the environment (eyes, ears, tactile input etc.). Action refers to your hands, legs, or motors/engines in machines that help you move on the basis of this information. Learning is kind of the glue in between. It helps crunch information of your sensors quickly, compare it with past data, guesses what future data may look like and computes actions that are likely to succeed. The three facets of intelligence are not sequential and robotics is not merely a feed-forward process. Your sensory inputs depend on the previous action you took.



[1] feel free to come up with another definition

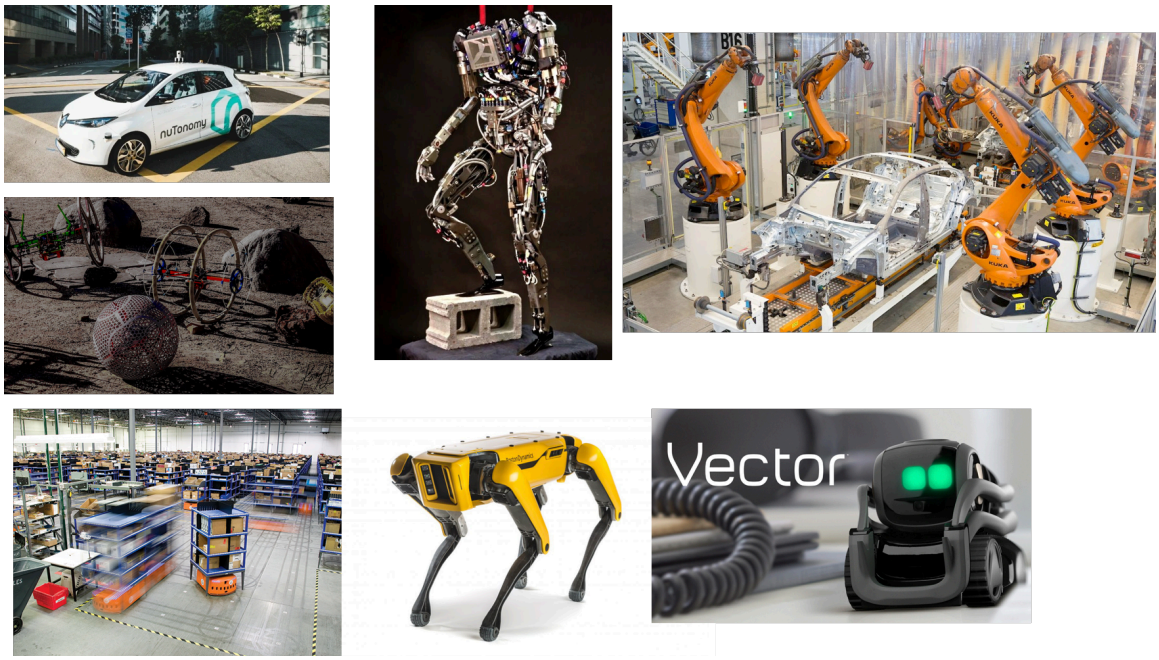
1.2 Goals of this course

The goals of this course is to develop the main ideas in robotic perception, learning and control. Robotics *is* everything, so we will focus on understanding how they are *combined together* to build a typical robot. After this course, we expect you to be able to choose one among the different robotics algorithms to perform a particular task, think critically about these algorithms and build new ones.

Other courses Some other courses at Penn that address various aspects of this picture above are

- Perception: CIS 580, CIS 581, CIS 680
- Learning: CIS 520, CIS 521, CIS 522, CIS 620, CIS 700, ESE 545, ESE 546
- Control: ESE 650, MEAM 520, MEAM 620, ESE 500, ESE 505, ESE 619

1.3 Some of my favorite robots



These videos should give you an idea of how the everyday life of a roboticist looks like: Kiva's robots, Waymo's 360 experience, Boston Dynamics' Spot, JPL-MIT team at the DARPA Sub-T Challenge, Romeo and Juliet at Ferrari's factory, Anki's Vector, and the DARPA Humanoid Challenge.

Chapter 2

Introduction to State Estimation

Reading

1. Barfoot, Chapter 2.1-2.2
2. Thrun, Chapter 2
3. Russell Chapter 15.1-15.3

2.1 A review of probability

Probability is a very useful construct to reason about real systems which we cannot model at all scales. It is a fundamental part of robotics. No matter how sophisticated your camera, it will have noise in how it measures the real world around it. No matter how good your model for a motor is, there will be unmodeled effects which make it move a little differently than how you expect. We begin with a quick review of probability, you can read more at many sources, e.g., MIT's OCW.

An experiment is a procedure which can be repeated infinitely and has a well-defined set of possible outcomes, e.g., the toss of a coin or the roll of dice. The outcome itself need not always be deterministic, e.g., depending upon your experiment, the coin may come up heads or tails. We call the set Ω the *sample space*, it is the set of all possible outcomes of an experiment. For two coins, this set would be

$$\Omega = \{HH, HT, TH, TT\}.$$

We want to pick this set to be right granularity to answer relevant questions, e.g., it is correct but not very useful for Ω to be the position of all the

molecules in the coin. After every experiment, in this case tossing the two coins once each, we obtain an event, it is a subset *event* $A \subset \Omega$ from the sample space.

$$A = \{HH\}.$$

Probability theory is a mathematical framework that allows us to reason about phenomena or experiments whose outcome is uncertain. Probability of an event

$$\mathrm{P}(A)$$

is a function that maps each event A to a number between 0 and 1: closer to 1 this number, stronger our belief that the outcome of the experiment is going to be A.

Axioms Probability is formalized using a set of three basic axioms that are intuitive and yet very powerful. They are known as Kolmogorov's axioms:

- Non-negativity: $\mathrm{P}(A) \geq 0$
- Normalization: $\mathrm{P}(\Omega) = 1$
- Additivity: If two events A, B are such that $A \cap B = \emptyset$, then

$$\mathrm{P}(A \cup B) = \mathrm{P}(A) + \mathrm{P}(B).$$

You can use these axioms to say things like $\mathrm{P}(\emptyset) = 0$, $\mathrm{P}(A^c) = 1 - \mathrm{P}(A)$, or if $A \subseteq B$ then $\mathrm{P}(A) \leq \mathrm{P}(B)$.

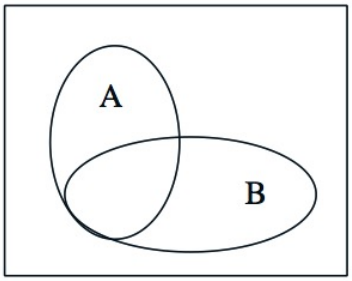
Conditioning on events Conditioning helps us answer questions like

$$\mathrm{P}(A \mid B) := \text{probability of } A \text{ given that } B \text{ occurred.}$$

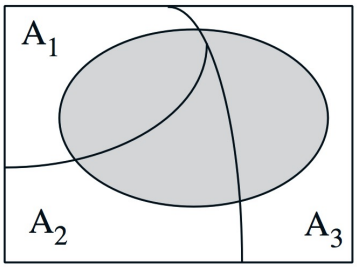
Effectively, the sample space has now shrunk from Ω to the event B. It would be silly to have a null sample-space, so let's say that $\mathrm{P}(B) \neq 0$. We define conditional probability as

$$\mathrm{P}(A \mid B) = \frac{\mathrm{P}(A \cap B)}{\mathrm{P}(B)}; \tag{2.1}$$

the probability is undefined if $\mathrm{P}(B) = 0$. Using this definition, we can compute the probability of events like "what is the probability of rolling a 2 on a die given that an even number was rolled".



Partitioning the sample space



We can use this trick to get the law of total probability: if a finite number of events $\{A_i\}$ form a partition of Ω, i.e.,

$$A_i \cap A_j = \emptyset \; \forall i, j, \text{ and } \bigcup_i A_i = \Omega$$

$$\mathrm{P}(B) = \sum_i \mathrm{P}(B \mid A_i)\ \mathrm{P}(A_i). \tag{2.2}$$

Bayes' rule Imagine that instead of someone telling us that the conditioning event actually happened, we simply had a belief

$$\mathrm{P}(A_i)$$

about the possibility of such events $\{A_i\}$. For each of A_i, we can compute the conditional probability $\mathrm{P}(B \mid A_i)$ using (2.1). Say we run our experiment and observe that B occurred, how would our belief on the events A_i change? In other words, we wish to compute

$$\mathrm{P}(A_i \mid B).$$

This is the subject of Bayes' rule.

$$\begin{aligned} \mathrm{P}(A_i \mid B) &= \frac{\mathrm{P}(A_i \cap B)}{\mathrm{P}(B)} \\ &= \frac{\mathrm{P}(A_i)\,\mathrm{P}(B|A_i)}{\mathrm{P}(B)} \\ &= \frac{\mathrm{P}(A_i)\,\mathrm{P}(B|A_i)}{\sum_i \mathrm{P}(A_j)\ \mathrm{P}(B \mid A_j)}. \end{aligned} \tag{2.3}$$

Bayes' rule naturally leads to the concept of independent events. Two events $A, B \subset \Omega$ are independent if observing one does not give us any information about the other

$$\mathrm{P}(A \cap B) = \mathrm{P}(A)\,\mathrm{P}(B). \tag{2.4}$$

This is different from disjoint events. Disjoint events never co-occur, i.e., observing one tells us that the other one *did not* occur.

Probability for experiments with real-valued outcomes We need some more work in defining probability for events with real-valued outcomes. The sample space is easy enough to understand, e.g., $\Omega = [0, 1]$ for your score at the end of this course. We however run into difficulties if we define the probability of general subsets of Ω in terms of the probabilities of elementary outcomes (*elements* of Ω). For instance, if we wish to model all elements $\omega \in \Omega$ to be equally likely, we are forced to assign each element ω a probability of zero (to be consistent with the second axiom of probability). This is not very helpful in determining the probability of the score being 0.9. If you instead assigned some small non-zero number to $\mathrm{P}(\omega_i)$, then we have undesirable conclusions such as

$$\mathrm{P}(\{1, 1/2, 1/3, \ldots\}) = \infty.$$

The way to fix this is to avoid defining the probability of a set in terms of the probability of elementary outcomes and work with more general sets. While we would ideally like to be able to specify the probability of every subset of Ω, it turns out that we cannot do so in a mathematically consistent way. The trick then is to work with a smaller object known as a σ-algebra, that is the set of "nice" subsets of Ω.

Given a sample space Ω, a σ-algebra $\mathcal{F}$ (also called a σ-field) is a collection of subsets of Ω such that

- $\emptyset \in \mathcal{F}$
- If $A \in \mathcal{F}$, then $A^c \in \mathcal{F}$.
- If $A_i \in \mathcal{F}$ for every $i \in \mathbb{N}$, then $\cup_{i=1}^{\infty} A_i \in \mathcal{F}$.

In short, σ-algebra is a collection of subsets of Ω that is closed under complement and countable unions. The pair $(\Omega, \mathcal{F})$, also called a measurable space, is now used to define probability of events. A set A that belongs to $\mathcal{F}$ is called an event. The probability measure

$$\mathrm{P} : \mathcal{F} \to [0, 1].$$

assigns a probability to events in $\mathcal{F}$. We cannot take $\mathcal{F}$ to be too small, e.g., elements of $\mathcal{F} = \{\emptyset, \Omega\}$ are easy to construct our P but are not very useful. For technical reasons, the σ-algebra cannot be too large; notice that we used this concept to *avoid* considering every subset of the sample space $\mathcal{F} = 2^{\Omega}$. Modern probability is defined using a Borel σ-algebra. Roughly speaking, this is an $\mathcal{F}$ that is just large enough to do interesting things but small enough that mathematical technicalities do not occur.

2.1.1 Random variables

A random variable is an assignment of a value to every possible outcome. Mathematically, in our new language of a measurable space, a random variable is a function

$$X : \Omega \to \mathbb{R}$$

if the set $\{\omega : \ X(\omega) \leq c\}$ is $\mathcal{F}$-measurable for every number $c \in \mathbb{R}$. This is equivalent to saying that every preimage of the Borel σ-algebra on reals $\mathcal{B}(\mathbb{R})$ is in $\mathcal{F}$. A statement $X(\omega) = x = 5$ means that the outcome of our experiment happens to be $\omega \in \Omega$ when the realized value of the random variable is a particular number x equal to 5.

We can now define functions of random variables, e.g., if X is a random variable, the function $Y = X^3(\omega)$ for every $\omega \in \Omega$, or $Y = X^3$ for short, is a new random variable. An indicator random variable is special. If $A \subset \Omega$, let $I_A : \Omega \to \{0, 1\}$ be the indicator function of this set A, i.e., $I_A(\omega) = 1$ if $\omega \in A$ and zero otherwise. If our set $A \in \mathcal{F}$, then I_A is an indicator random variable.

Probability mass functions The probability law, or a probability distribution, of a random variable X is denoted by

$$p_X(x) := \mathrm{P}(X = x) = \mathrm{P}\left(\{\omega \in \Omega : X(\omega) = x\}\right).$$

We denote probability distribution using a lower-case p. It is a function of the realized value x in the range of a random variable, and $p_X(x) \geq 0$ (the probability is non-zero) and $\sum_x p_X(x) = 1$ if X takes on a discrete

ℹ Random variables are typically denoted using capital letters, X, Y, Z although we will be sloppy and not always do so in this course to avoid complicated notation. The distinction between a random variable and the value that it takes will be clear from context.

❓ Let us check that Y satisfies our definition of a random variable. If $\{\omega : X(\omega) \leq c\}$ lies in $\mathcal{F}$ then the set $\left\{\omega : Y(\omega) \leq c^{1/3}\right\}$ also lies in $\mathcal{F}$.

ℹ The function I_A is not a random variable if $A \notin \mathcal{F}$, but this is, as we said in the previous section, a mathematical corner case. Most subsets of Ω belong to $\mathcal{F}$.

number of values. For instance, if X is the number of coin tosses until the first head, if we assume that our tosses are independent $\mathrm{P}(H) = p > 0$, then we have

$$p_X(k) = \mathrm{P}(X = k) = \mathrm{P}(TT \cdots TH) = (1-p)^{k-1} p$$

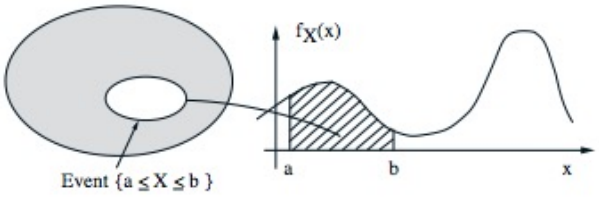
for all $k = 1, 2, \ldots$. This is what is called a geometric probability mass function.

Cumulative distribution function A cumulative distribution function (CDF) is the probability of a random variable X taking a value less than an particular $x \in \mathbb{R}$, i.e.,

$$F_X(x) = \mathrm{P}(X \leq x).$$

The CDF $F_X(x)$ is a non-decreasing function of x. It converges to zero as $x \to -\infty$ and goes to 1 as $x \to \infty$.

Probability density functions A continuous random variable, i.e., one that takes values in $\mathbb{R}$ is described by a probability density function.



If $F_X(x)$ is the CDF of an r.v. X and X takes values in $\mathbb{R}$, the probability density function (PDF) $f_X(x)$ (or sometimes also denoted by $p_X(x)$) is defined to be

$$\mathrm{P}(a \leq X \leq b) = \int_a^b f_X(x)\ \mathrm{d}x\,.$$

We also have the following relationship between the CDF and the PDF, the former is the integral of the latter:

$$\mathrm{P}(-\infty \leq X \leq x) = F_X(x) = \int_{-\infty}^{x} f_X(x)\ \mathrm{d}x\,.$$

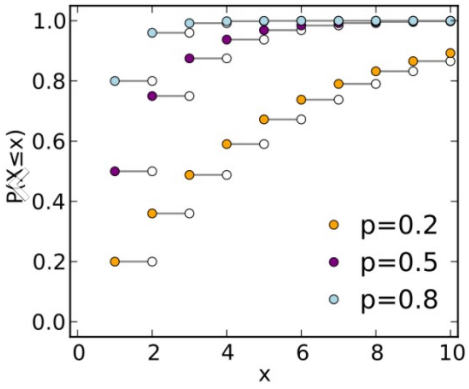
This leads to the following interpretation of the probability density function:

$$\mathrm{P}(x \leq X \leq x + \delta) \approx f_X(x)\,\delta.$$

Expectation and Variance The expected value of a random variable X is

$$\mathrm{E}[X] = \sum_x x\, p_X(x)$$

The CDF of a geometric random variable for different values of p



Note that CDFs need not be continuous: in the case of a geometric random variable, since the values that X takes belong to the set of integers, the CDF is constant between any two integers.

and denotes the center of gravity of the probability mass function. Roughly speaking, it is the average of a large number of repetitions of the same experiment. Expectation is a linear, i.e.,

$$\mathrm{E}[aX + b] = a\,\mathrm{E}[X] + b$$

for any constants a, b. For two independent random variables X, Y we have

$$\mathrm{E}[XY] = \mathrm{E}[X]\,\mathrm{E}[Y].$$

We can also compute the expected value of any function $g(X)$ using the same formula

$$\mathrm{E}[g(X)] = \sum_x g(x)\; p_X(x).$$

In particular, if $g(x) = x^2$ we have the second moment $\mathrm{E}[X^2]$. The variance is defined to be

$$\begin{aligned}\mathrm{Var}(X) &= \mathrm{E}\left[(X - \mathrm{E}[X])^2\right] \\ &= \sum_x (x - \mathrm{E}[X])^2\, p_X(x) \\ &= \mathrm{E}[X^2] - (\mathrm{E}[X])^2\,.\end{aligned}$$

The variance is always non-negative $\mathrm{Var}(X) \geq 0$. For an affine function of X, we have

$$\mathrm{Var}(aX + b) = a^2\,\mathrm{Var}(X).$$

For continuous-valued random variables, the expectation is defined as

$$\mathrm{E}[X] = \int_{-\infty}^{\infty} x p_X(x)\;\mathrm{d}x\,;$$

the definition of variance remains the same.

Joint distributions We often wish to think of the joint probability distribution of multiple random variables, say the location of an autonomous car in all three dimensions. The cumulative distribution function associated with this is therefore

$$F_{X,Y,Z}(x, y, z) = \mathrm{P}(X \leq x, Y \leq y, Z \leq z).$$

Just like we have the probability density of a single random variable, we can also write the joint probability density of multiple random variables $f_{X,Y,Z}(x, y, z)$. In this case we have

$$F_{X,Y,Z}(x, y, z) = \int_{-\infty}^{x}\int_{-\infty}^{y}\int_{-\infty}^{z} f_{X,Y,Z}(x, y, z)\,\mathrm{d}z\,\mathrm{d}y\,\mathrm{d}x\,.$$

The joint probability density factorizes if two random variables are independent:

$$f_{X,Y}(x, y) = f_X(x) f_Y(y) \quad \text{for all } x, y.$$

Two random variables are uncorrelated if and only if

$$\mathrm{E}[XY] = \mathrm{E}[X]\,\mathrm{E}[Y].$$

Note that independence implies uncorrelatedness, they are not equivalent. The covariance is defined as

$$\mathrm{Cov}(X, Y) = \mathrm{E}[XY] - \mathrm{E}[X]\,\mathrm{E}[Y].$$

Conditioning As we saw before, for a single random variable X we have

$$\mathrm{P}(x \le X \le x + \delta) \approx f_X(x)\,\delta.$$

For two random variables, by analogy we would like

$$\mathrm{P}(x \le X \le x + \delta \mid Y \approx y) \approx f_{X|Y}(x \mid y)\delta.$$

The conditional probability density function of X given Y is defined to be

$$f_{X|Y}(x \mid y) = \frac{f_{X,Y}(x, y)}{f_Y(y)} \quad \text{if } f_Y(y) > 0.$$

For any given y, the conditional PDF is a normalized section of the joint PDF, as shown below.

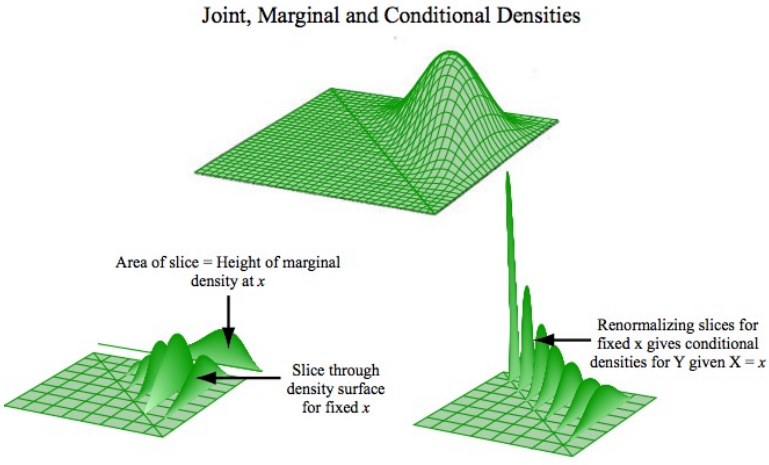


Image by MIT OpenCourseWare, adapted from ***Probability***, by J. Pittman, 1999.

Continuous form of Bayes rule We can show using the definition of conditional probability that

$$f_{Y|X}(y \mid x) = \frac{f_{X|Y}(x \mid y) f_Y(y)}{f_X(x)}. \tag{2.5}$$

Similarly we also have the law of total probability in the continuous form

$$f_X(x) = \int_{-\infty}^{\infty} f_{X|Y}(x \mid y)\ f_Y(y)\ \mathrm{d}y\,.$$

2.2 Using Bayes rule for combining evidence

We now study a prototypical state estimation problem. Let us consider a robot that is trying to check whether the door to a room is open or not.

We will abstract each observation by the sensors of the robot as a random variable Y. This could be the image from its camera after running some algorithm to check the state of the door, the reading from a laser sensor (if the time-of-flight of the laser is very large then the door is open), or any other mechanism. We have two kinds of conditional probabilities in this problem

$$\mathrm{P}(\text{open} \mid Y) \text{ is a diagnostic quantity, while}$$
$$\mathrm{P}(Y \mid \text{open}) \text{ is a causal quantity.}$$

The second one is called a causal quantity because the specific Y we observe depends upon whether the door is open or not. The first one is called a diagnostic quantity because using this observation Y we can infer the state of the environment, i.e., whether the door is open or not. Next imagine how you would calibrate the sensor in a lab: for each value of the state of the door open, not open you would record all the different observations received Y and calculate the conditional probabilities. The causal probability is much easier to calculate in this context, one may even use some knowledge of elementary physics to model the probability $\mathrm{P}(Y \mid \text{open})$, or one may count the number of times the observation is $Y = y$ for a given state of the door.

Bayes rule allows us to transform causal knowledge into diagnostic

knowledge

$$\mathrm{P}(\text{open} \mid Y) = \frac{\mathrm{P}(Y \mid \text{open})\,\mathrm{P}(\text{open})}{\mathrm{P}(Y)}.$$

Remember that the left hand side (diagnostic) is typically something that we desire to calculate. Let us put some numbers in this formula. Let $\mathrm{P}(Y \mid \text{open}) = 0.6$ and $\mathrm{P}(Y \mid \text{not open}) = 0.3$. We will imagine that the door is open or closed with equal probability: $\mathrm{P}(\text{open}) = \mathrm{P}(\text{not open}) = 0.5$. We then have

$$\begin{aligned}\mathrm{P}(\text{open} \mid Y) &= \frac{\mathrm{P}(Y \mid \text{open})\,\mathrm{P}(\text{open})}{\mathrm{P}(Y)} \\ &= \frac{\mathrm{P}(Y \mid \text{open})\,\mathrm{P}(\text{open})}{\mathrm{P}(Y \mid \text{open})\,\mathrm{P}(\text{open}) + \mathrm{P}(Y \mid \text{not open})\,\mathrm{P}(\text{not open})} \\ &= \frac{0.6 \times 0.5}{0.6 \times 0.5 + 0.3 \times 0.5} = \frac{2}{3}.\end{aligned}$$

Notice something very important, the original (prior) probability of the state of the door was 0.5. If we have a sensor that fires with higher likelihood if the door is open, i.e., if

$$\frac{\mathrm{P}(Y \mid \text{open})}{\mathrm{P}(Y \mid \text{not open})} > 1$$

then the probability of the door being open after receiving an observation *increases*. If the likelihood were less than 1, then observing a realization of Y would reduce our estimate of the probability of the door being open.

ℹ The denominator in Bayes rule, i.e., $\mathrm{P}(Y)$ is called the evidence in statistics.

Combining evidence for Markov observations Say we updated the prior probability using our first observation Y_1, let us take another observation Y_2. How can we integrate this new observation? It is again an application of Bayes rule using two observations, or in general multiple observations $Y_1, \ldots, Y_n$. Let us imagine this time that $X = \text{open}$.

$$\mathrm{P}(X \mid Y_1, \ldots, Y_n) = \frac{\mathrm{P}(Y_n \mid X, Y_1, \ldots, Y_{n-1})\,\mathrm{P}(X \mid Y_1, \ldots, Y_{n-1})}{\mathrm{P}(Y_n \mid Y_1, \ldots, Y_{n-1})}.$$

Let us make the very natural assumption that says that our observations from the sensor $Y_1, \ldots, Y_n$ are independent given the state of the door X. This is known as the Markov assumption.

We now have

$$\begin{aligned}\mathrm{P}(X \mid Y_1, \ldots, Y_n) &= \frac{\mathrm{P}(Y_n \mid X)\,\mathrm{P}(X \mid Y_1, \ldots, Y_{n-1})}{\mathrm{P}(Y_n \mid Y_1, \ldots, Y_{n-1})} \\ &= \eta\,\mathrm{P}(Y_n \mid X)\,\mathrm{P}(X \mid Y_1, \ldots, Y_{n-1})\end{aligned}$$

where

$$\eta^{-1} = \mathrm{P}(Y_n \mid Y_1, \ldots, Y_{n-1})$$

is the denominator. We can now expand the diagnostic probability on the

❓ Exchangeable sequences of random variables $Y_1, \ldots, Y_n$ are sequences such that the joint probability of the sequence does not change upon permutations, i.e.,

$$\mathrm{P}(Y_1, \ldots, Y_n) = \mathrm{P}(Y_{\pi(1)}, \ldots, Y_{\pi(n)})$$

for a permutation $(\pi(1), \ldots, \pi(n))$ of $(1, \ldots, n)$. A simple example of an exchangeable sequence is as follows. Take an urn with 1 red ball and 2 blue balls, if Y_i is the color of the ball drawn at the i^{th} draw, then the sequence (Y_1, Y_2, Y_3) is exchangeable. De Finetti's theorem states that for any exchangeable sequence, the random variables are conditionally independent given some latent variable, i.e., if $Y_1, Y_2, \ldots$ is exchangeable, then for any i, j we have

$$\mathrm{P}(Y_i, Y_j \mid X) = \mathrm{P}(Y_i \mid X)\,\mathrm{P}(Y_j \mid X)$$

for the latent variable X. So instead of assuming the existence of a state of the door X in our calculation, we could have assumed that the observations of the sensor are exchangeable. Depending upon your point of view, this is a huge philosophical difference.

right-hand side recursively to get

$$\mathrm{P}(X \mid Y_1, \ldots, Y_n) = \prod_{i=1}^{n} \eta_i \, \mathrm{P}(Y_i \mid X) \;\; \mathrm{P}(X). \tag{2.6}$$

where $\eta_i^{-1} = \mathrm{P}(Y_i \mid Y_1, \ldots, Y_{i-1})$.

> The calculation in (2.6) is very neat and you should always remember it. Given multiple observations $Y_1, \ldots, Y_n$ of the same quantity X, we can compute the conditional probability $\mathrm{P}(X \mid Y_1, \ldots, Y_n)$ if we code up two functions to compute
>
> - the causal probability (also called the likelihood of an observation) $\mathrm{P}(Y_i \mid X)$, and
> - the denominator η_i^{-1}.
>
> Given these two functions, we can use the recursion to update multiple observations. The same basic idea also holds if you have two quantities to estimate, e.g., $X_1 =$ open door and $X_2 =$ color of the door. The recursive application of Bayes rule lies at the heart of all state estimation methods.

Let us again put some numbers into these formulae, imagine that the observation Y_2 was taken using a different sensor which now has

$$\mathrm{P}(Y_2 \mid \text{open}) = 0.5 \text{ and } \mathrm{P}(Y_2 \mid \text{not open}) = 0.6.$$

We have from our previous calculation that $\mathrm{P}(\text{open} \mid Y_1) = 2/3$ and

$$\begin{aligned}\mathrm{P}(\text{open} \mid Y_1, Y_2) &= \frac{\mathrm{P}(Y_2 \mid \text{open}) \, \mathrm{P}(\text{open} \mid Y_1)}{\mathrm{P}(Y_2 \mid \text{open}) \, \mathrm{P}(\text{open} \mid Y_1) + \mathrm{P}(Y_2 \mid \text{not open}) \, \mathrm{P}(\text{not open} \mid Y_1)} \\ &= \frac{0.5 \times 2/3}{0.5 \times 2/3 + 0.6 \times 1/3} = \frac{5}{8} = 0.625.\end{aligned}$$

Notice in this case that the probability that the door is open has reduced from $\mathrm{P}(\text{open} \mid Y_1) = 2/3$.

2.2.1 Coherence of Bayes rule

Would the probability change if we used sensor Y_2 before using Y_1? In this case, the answer to this question is no and you are encouraged to perform this computation for yourselves. Bayes rule is coherent, it will give the same result regardless of the order of observations.

The order of incorporating observation matters if the state of the world changes while we make observations, e.g., if we have a sensor that tracks the location of a car, the car presumably moves in between two observations and we would get the wrong answer if our question was "is there a car at this location".

❓ Can you think of a situation where the order of incorporating observations matters?

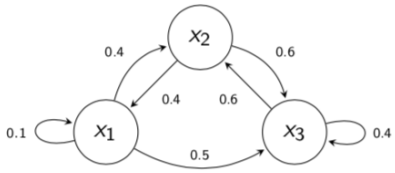
As we motivated in the previous chapter, movement is quite fundamental to robotics and we are typically concerned with estimating the state of a dynamic world around us using our observations. We will next study the concept of a Markov Chain which is a mathematical abstraction for the evolution of the state of the world.

2.3 Markov Chains

Consider the Whack-The-Mole game: a mole has burrowed a network of three holes x_1, x_2, x_3 into the ground. It keeps going in and out of the holes and we are interested in finding which hole it will show up next so that we can give it a nice whack.

- Three holes: $X = \{x_1, x_2, x_3\}$.
- Transition probabilities:

$$T = \begin{bmatrix} 0.1 & 0.4 & 0.5 \\ 0.4 & 0 & 0.6 \\ 0 & 0.6 & 0.4 \end{bmatrix}$$



This is an example of a Markov chain. There is a transition matrix T which determines the probability T_{ij} of the mole resurfacing on a given hole x_j given that it resurfaced at hole x^i the last time. The matrix T^k is the k-step transition matrix

$$T_{ij}^k = \mathrm{P}(X_k = x_j \mid X_0 = x_i).$$

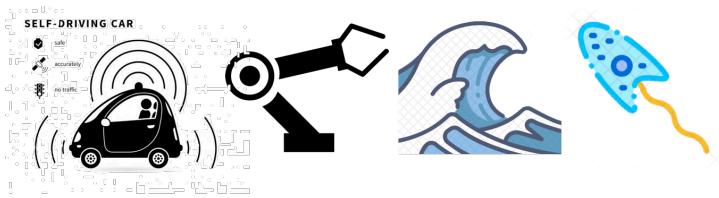
You can see the animations at https://setosa.io/ev/markov-chains to build more intuition.

The key property of a Markov chain is that the next state X_{k+1} is independent of all the past states $X_1, \ldots, X_{k-1}$ given the current state X_k.

$$X_{k+1} \perp\!\!\!\perp X_1, \ldots, X_{k-1} \mid X_k$$

This is known as the Markov property and all systems where we can define a "state" which governs their evolution have this property. Markov chains form a very broad class of systems. For example, all of Newtonian physics fits this assumption.

What is the state of the following systems?



❓ Does a deterministic dynamical system, e.g., a simple pendulum, also satisfy the Markov assumption? What is the transition matrix in this case?

❓ Can you think of a system which does not have the Markov property?

Consider the paramecium above. Its position depends upon a large number of factors: its own motion from the previous time-step but also the viscosity of the material in which it is floating around. One may model the state of the environment around the paramecium as a liquid whose molecules hit thousands of times a second, essentially randomly, and cause disturbances in how the paramecium moves. Let us call this disturbance "noise in the dynamics". If the motion of the molecules of the liquid has some correlations (does it, usually?), this induces correlations in the position of the paramecium. The position of the organism is no longer Markov. This example is important to remember, the Markov property defined above also implies that the noise in the state transition matrix is independent.

Evolution of a Markov chain The probability of being in a state x^i at time $k+1$ can be written as

$$\mathrm{P}(X_{k+1} = x_i) = \sum_{j=1}^{N} \mathrm{P}(X_{k+1} = x_i \mid X_k = x_j)\ \mathrm{P}(X_k = x_j).$$

This equation governs how the probabilities $\mathrm{P}(X_k = x_i)$ change with time k. Let's do the calculations for the Whack-The-Mole example. Say the mole was at hole x_1 at the beginning. So the probability distribution of its presence

$$\pi^{(k)} = \begin{bmatrix} \mathrm{P}(X_k = x_1) \\ \mathrm{P}(X_k = x_2) \\ \mathrm{P}(X_k = x_3) \end{bmatrix}$$

is such that

$$\pi^{(1)} = [1, 0, 0]^\top.$$

We can now write the above formula as

$$\pi^{(k+1)} = T'\pi^{(k)} \tag{2.7}$$

[1] and compute the distribution $\pi^{(t)}$ for all times

$$\begin{aligned}
\pi^{(2)} &= T'\pi^{(1)} = [0.1, 0.4, 0.5]^\top; \\
\pi^{(3)} &= T'\pi^{(2)} = [0.17, 0.34, 0.49]^\top; \\
\pi^{(4)} &= T'\pi^{(3)} = [0.153, 0.362, 0.485]^\top; \\
&\vdots \\
\pi^{(\infty)} &= \lim_{k\to\infty} T'^k\, \pi^{(1)} \\
&= [0.158, 0.355, 0.487]^\top.
\end{aligned}$$

The numbers $\mathrm{P}(X_k = x_i)$ stop changing with time k. Under certain technical conditions, the distribution $\pi^{(\infty)}$ is unique (single communicating

[1] Let us denote the transpose of the matrix T using the Matlab notation T' instead of $T^\top$ for clarity.

class for a Markov chain with a finite number states). We can compute this invariant distribution by writing

$$\pi^{(\infty)} = T'\pi^{(\infty)}.$$

We can also compute the distribution $\pi^{(\infty)}$ directly: the invariant distribution is the right-eigenvector of the matrix T' corresponding to the eigenvalue 1.

Do we always know that the transition matrix has an eigenvalue that is 1?

Example 2.1. Consider a Markov chain on two states where the transition matrix is given by

$$T = \begin{bmatrix} 0.5 & 0.5 \\ 0.4 & 0.6 \end{bmatrix}.$$

The invariant distribution is

$$\pi^{(1)} = 0.5\pi^{(1)} + 0.4\pi^{(2)}$$
$$\pi^{(2)} = 0.5\pi^{(1)} + 0.6\pi^{(2)}.$$

Note that the constraint for π being a probability distribution, i.e., $\pi^{(1)} + \pi^{(2)} = 1$ is automatically satisfied by the two equations. We can solve for $\pi^{(1)}, \pi^{(2)}$ to get

$$\pi^{(1)} = 4/9 \quad \pi^{(2)} = 5/9.$$

2.4 Hidden Markov Models (HMMs)

[2]

Markov chains are a good model for how the state of the world evolves with time. We may not always know the exact state of these systems and only have sensors, e.g., cameras, LiDARs, and radars, to record observations. These sensors are typically noisy. So we model the observations as random variables.

Hidden Markov Models (HMMs) are an abstraction to reason about observations of the state of a Markov chain. An HMM is a sequence of random variables $Y_1, Y_2, \ldots, Y_n$ such that the distribution of Y_k only depends upon the hidden state X_k of the associated Markov chain.

[2] Parts of this section closely follow Emilio Frazzoli's course notes at https://ocw.mit.edu/courses/aeronautics-and-astronautics/16-410-principles-of-autonomy-and-decision-making-fall-2010/lecture-notes/MIT16_410F10_lec20.pdf and https://ocw.mit.edu/courses/aeronautics-and-astronautics/16-410-principles-of-autonomy-and-decision-making-fall-2010/lecture-notes/MIT16_410F10_lec21.pdf

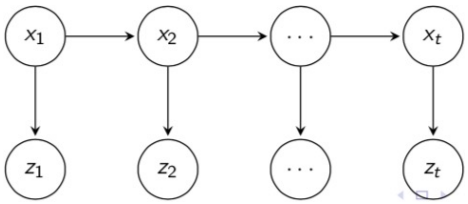


Figure 2.1: A Hidden Markov Model with the underlying Markov chain, the observation at time k only depends upon the hidden state at that time instant. Ignore the notation $Z_1, \ldots, Z_t$ we will denote the observations by Y_k.

Notice that an HMM always has an underlying Markov chain behind it. For example, if we model the position of a car X_k as a Markov chain, our observation of the position at time k would be Y_k. In our example of the robot sensing whether the door is open or closed using multiple observations across time, the Markov chain is trivial, it is simply the transition matrix $\mathrm{P}(\text{not open} \mid \text{not open}) = \mathrm{P}(\text{open} \mid \text{open}) = 1$. Just like Markov chains, HMMs are a very general class of mathematical models that allow us to think about multiple observations across time of a Markov chain.

Let us imagine that the observations of our HMM are also finite in number, e.g., your score in this course $\in [0, 100]$ where the associated state of the Markov chain is your expertise in the subject matter. We will write a matrix of observation probabilities

$$M_{ij} = \mathrm{P}(Y_k = y_j | \; X_k = x_i). \tag{2.8}$$

The matrix M has non-negative entries, after all, each entry is a probability. Since each state has to result in *some* observation, we also have

$$\sum_j M_{ij} = 1.$$

The state transition probabilities of the associated Markov chain are

$$T_{ij} = \mathrm{P}(X_{k+1} = x_j | \; X_k = x_i).$$

Given the abstraction of an HMM, we may be interested in solving a number of problems. We will consider the problem where the state X_k is the position of a car (which could be stationary or moving) and observations Y_k give us some estimate of the position.

1. **Filtering**: Given observations up to time k, compute the distribution of the state at time k

   $$\mathrm{P}(X_k \mid Y_1, \ldots, Y_k).$$

   This is the most natural problem to understand: we want to find the probability of the car being at a location at time k given all previous observations. This is a temporally causal prediction, i.e., we are not using any information from the future to reason about the present.

2. **Smoothing**: Given observations up to time k, compute the distribution of the state at any time $j < k$

$$\mathrm{P}(X_j \mid Y_1, \ldots, Y_k) \quad \text{for } j < k.$$

The observation at a future time Y_{k+1} gives us some indication of where the car might have been at time k. In this case we are interested in using the entire set of observations from the past $Y_1, \ldots, Y_j$, the future $Y_{j+1}, \ldots, Y_k$ to estimate the position of the car. Of course, this problem can only be solved *ex post facto*, i.e., after the time instant j. An important thing to remember is that we are interested in the position of the car for all $j < k$ in smoothing.

3. **Prediction**: Given observations up to time k, compute the distribution of the state at a time $j > k$

$$\mathrm{P}(X_j \mid Y_1, \ldots, Y_k) \quad \text{for } j > k.$$

This is the case when we wish to make predictions about the state of the car $j > k$ given only observations until time k. If we knew the underlying Markov chain for the HMM and its transition matrix T, this would amount to running (2.7) forward using the output of the filtering problem as the initial distribution of the state.

❓ Why is this true?

4. **Decoding**: Find the most likely state trajectory $X_1, \ldots, X_k$ that maximizes the probability

$$\mathrm{P}(X_1, \ldots, X_k \mid Y_1, \ldots, Y_k)$$

given observations $Y_1, \ldots, Y_k$. Observe that the smoothing problem is essentially solved independently for all time-steps $j < k$. It stands to reason that if we knew a certain state (say car made a right turn) was likely given observations at time $k + 1$ and that the traffic light was green at time k (given our observations of the traffic light), then we know that the car did not stop at the intersection at time k. The decoding problem allows us to reason about the joint probability of the states and outputs the most likely trajectory given all observations.

5. **Likelihood of observations**: Given the observation trajectory, $Y_1, \ldots, Y_k$, compute the probability

$$\mathrm{P}(Y_1, \ldots, Y_k).$$

As you may recall, this is the denominator that we need for the recursive application of Bayes rule. It is made difficult by the fact that we do not know the state trajectory $X_1, \ldots, X_k$ corresponding to these observations.

These problems are closely related with each other and we will next dig deeper into them. We will first discuss two building blocks, called the

forward and backward algorithm that together help solve all the above problems.

2.4.1 The forward algorithm

Consider the problem of computing the likelihood of observations. We can certainly write

$$\begin{aligned}
&\mathrm{P}(Y_1, \ldots, Y_k) \\
&= \sum_{\text{all } (x_1,\ldots,x_k)} \mathrm{P}(Y_1, \ldots, Y_k \mid X_1, \ldots, X_k)\, \mathrm{P}(X_1, \ldots, X_k) \\
&= \sum_{\text{all } (x_1,\ldots,x_k)} \prod_{i=1}^{k} \mathrm{P}(Y_i = y_i \mid X_i = x_i)\, \mathrm{P}(X_1 = x_1) \prod_{i=2}^{k} \mathrm{P}(X_k = x_k \mid X_{k-1} = x_{k-1}) \\
&= \sum_{\text{all } (x_1,\ldots,x_k)} M_{x_1 y_1}\, M_{x_2 y_2} \ldots M_{x_k y_k}\, \pi_{x_1}\, T_{x_1 x_2} \ldots T_{x_{k-1} x_k}.
\end{aligned}$$

But this is a very large computation, for each possible trajectory $(x_1, \ldots, x_k)$ the states could have taken, we need to perform $2k$ matrix multiplications.

❓ How many possible state trajectories are there? What is the total cost of computing the likelihood of observations?

Forward algorithm We can simplify the above computation using the Markov property of the HMM as follows. We will define a quantity known as the forward variable

$$\alpha_k(x) = \mathrm{P}(Y_1, \ldots, Y_k, X_k = x) \tag{2.9}$$

where $Y_1, \ldots, Y_k$ is our observation sequence up to time k. Observe now that

1. We can initialize

$$\alpha_1(x) = \pi_x\, M_{x,y_1} \quad \text{for all } x.$$

2. For each time k, for all states x, we can compute

$$\alpha_{k+1}(x) = M_{x y_{k+1}} \sum_{x'} \alpha_k(x')\, T_{x' x}.$$

   using the law of total probability.

3. Finally, we have

$$\mathrm{P}(Y_1, \ldots, Y_k) = \sum_{x} \alpha_k(x)$$

   by marginalizing over the state variables X_k.

This recursion in the forward algorithm is a powerful idea and is much faster than our naive summation above.

❓ What is the computational complexity of the Forward algorithm?

2.4.2 The backward algorithm

Just like the forward algorithm performs the computation recursively in the forward direction, we can also perform a backward recursion to obtain the probability of the observations. Let us imagine that we have an observation trajectory

$$Y_1, \ldots, Y_t$$

up to some time t. We first define the so-called backward variables which are the probability of a future trajectory given the state of the Markov chain at a particular time instant

$$\beta_k(x) = \mathrm{P}(Y_{k+1}, Y_{k+2}, \ldots, Y_t \mid X_k = x). \tag{2.10}$$

Notice that the backward variables β_k with the conditioning on $X_k = x$ are slightly different than the forward variables α_k which are the joint probability of the observation trajectory and $X_k = x$.

Backward algorithm We can compute the variables $\beta_k(x)$ recursively again as follows.

1. Initialize
$$\beta_t(x) = 1 \quad \text{for all } x.$$
This simply indicates that since we are at the end of the trajectory, the future trajectory $Y_{t+1}, \ldots$ does not exist.

2. For all $k = t-1, t-2, \ldots, 1$, for all x, update
$$\beta_k(x) = \sum_{x'} \beta_{k+1}(x')\, T_{xx'} M_{x'y_{k+1}}.$$

3. We can now compute
$$\mathrm{P}(Y_1, \ldots, Y_t) = \sum_{x} \beta_1(x)\, \pi_x\, M_{xy_1}.$$

❓ What is the computational complexity of running the backward algorithm?

Implementing the forward and backward algorithms in practice The update equations for both α_k and β_k can be written using a matrix vector multiplication. We maintain the vectors

$$\alpha_k := [\alpha_k(x_1), \alpha_k(x_2), \ldots, \alpha_k(x_N)]$$
$$\beta_k := [\beta_k(x_1), \beta_k(x_2), \ldots, \beta_k(x_N)]$$

and can write the updates as

$$\alpha_{k+1}^\top = M_{\cdot,y_{k+1}}^\top \odot \left(\alpha_k^\top\, T\right)$$

where $\odot$ denotes the element-wise product and $M_{\cdot,y_{k+1}}$ is the y_{k+1}^{th} column of the matrix M. The update equation for the backward variables is

$$\beta_k = T\left(\beta_{k+1} \odot M_{\cdot,y_{k+1}}\right).$$

You must be careful about directly implement these recursions however, because we are iteratively multiplying by matrices T, M whose entries are all smaller than 1 (they are all probabilities after all), we can quickly run into difficulties where α_k, β_k become too small for some states and we get numerical underflow. You can implement these algorithms in the log-space by writing similar update equations for $\log \alpha_k$ and $\log \beta_k$ to avoid such numerical issues.

2.4.3 Bayes filter

Let us now use the forward and backward algorithms to solve the filtering problem. We want to compute

$$\mathrm{P}(X_k = x \mid Y_1, \ldots, Y_k)$$

for all states x in the Markov chain. We have that

$$\mathrm{P}(X_k = x \mid Y_1, \ldots, Y_k) = \frac{\mathrm{P}(X_k = x, Y_1, \ldots, Y_k)}{\mathrm{P}(Y_1, \ldots, Y_k)} = \eta\, \alpha_k(x) \quad (2.11)$$

where since $\mathrm{P}(X_k = x \mid Y_1, \ldots, Y_k)$ is a legitimate probability distribution on x, we have

$$\eta = \left(\sum_x \alpha_k(x)\right)^{-1}.$$

As simple as that. In order to estimate the state at time k, we run the forward algorithm to update variables $\alpha_i(x)$ from $i = 1, \ldots, k$. We can implement this using the matrix-vector multiplication in the previous section.

This is a commonly used algorithm known as the Bayes filter and is our first insight into state estimation.

An important fact Even if the filtering estimate is computed recursively using each observation as it arrives, the estimate is actually the probability of the current state given *all* past observations.

$$\mathrm{P}(X_k = x \mid Y_1, \ldots, Y_k) \neq \mathrm{P}(X_k = x \mid Y_k)$$

This is an extremely important concept to remember, in state-estimation we are always interested in computing the state given all available observations.

In the same context, is the following statement true?

$$\mathrm{P}(X_k = x \mid Y_1, \ldots, Y_k) = \mathrm{P}(X_k = x \mid Y_k, X_{k-1})$$

2.4.4 Smoothing

Given observations till time t, we would like to compute

$$\mathrm{P}(X_k = x \mid Y_1, \ldots, Y_t)$$

for all time instants $k = 1, \ldots, t$. Observe the filtering

$$\begin{aligned}
\mathrm{P}(X_k = x \mid Y_1, \ldots, Y_t) &= \frac{\mathrm{P}(X_k = x, Y_1, \ldots, Y_t)}{\mathrm{P}(Y_1, \ldots, Y_t)} \\
&= \frac{\mathrm{P}(X_k = x, Y_1, \ldots, Y_k, Y_{k+1}, \ldots, Y_t)}{\mathrm{P}(Y_1, \ldots, Y_t)} \\
&= \frac{\mathrm{P}(Y_{k+1}, \ldots, Y_t \mid X_k = x, Y_1, \ldots, Y_k)\ \mathrm{P}(X_k = x, Y_1, \ldots, Y_k)}{\mathrm{P}(Y_1, \ldots, Y_t)} \\
&= \frac{\mathrm{P}(Y_{k+1}, \ldots, Y_t \mid X_k = x)\ \mathrm{P}(X_k = x, Y_1, \ldots, Y_k)}{\mathrm{P}(Y_1, \ldots, Y_t)} \\
&= \frac{\beta_k(x)\ \alpha_k(x)}{\mathrm{P}(Y_1, \ldots, Y_t)}
\end{aligned} \tag{2.12}$$

Study the first step carefully, the numerator is *not* equal to $\alpha_k(x)$ because observations go all the way till time t. The final step uses both the Markov and the HMM properties: future observations $Y_{k+1}, \ldots, Y_t$ depend only upon future states $X_{k+1}, \ldots, X_t$ (HMM property) which are independent of the past observations and states give the current state $X_k = x$ (Markov property).

Smoothing can therefore be implemented by running the forward algorithm to update α_k from $k = 1, \ldots, t$ and the backward algorithm to update β_k from time $k = t, \ldots, 1$.

To see an example of smoothing in action, see ORB-SLAM 2. What do you think is the state of the Markov chain in this video?

❓ Both the filtering problem and the smoothing problem give us the probability of the state given observations. Discuss which one should we should use in practice and why?

Example for the Whack-the-mole problem Let us assume that we do not see which hole the mole surfaces from (say it is dark outside) but we can hear it. Our hearing is not very precise so we have an observation probabilities

$$M = \begin{bmatrix} 0.6 & 0.2 & 0.2 \\ 0.2 & 0.6 & 0.2 \\ 0.2 & 0.2 & 0.6 \end{bmatrix}.$$

Assume that the mole surfaces three times and we make the measurements

$$Y_1 = 1, Y_2 = 3, Y_3 = 3.$$

We want to compute the distribution of the states the mole could be in at each time. Assume that we know that the mole was in hole 1 at the first step, i.e., $\pi_1 = (1, 0, 0)$ for the Markov chain, like we had in Section 2.3.

Run the forward backward algorithm and see that

$$\alpha_1 = (0.6, 0, 0)\,, \alpha_2 = (0.012, 0.048, 0.18)\,, \alpha_3 = (0.0041, 0.0226, 0.0641)\,,$$

and

$$\beta_3 = (1, 1, 1)\,, \beta_2 = (0.4, 0.44, 0.36)\,, \beta_1 = (0.1512, 0.1616, 0.1392)\,.$$

Using these, we can now compute the filtering and the smoothing state distributions, let us denote them by π^f and π^s respectively.

$$\pi_1^f = (1, 0, 0)\,, \pi_2^f = (0.05, 0.2, 0.75), \pi_3^f = (0.045, 0.2487, 0.7063)$$

and

$$\pi_1^s = (1, 0, 0)\,, \pi_2^s = (0.0529, 0.2328, 0.7143), \pi_3^s = (0.045, 0.2487, 0.7063).$$

❓ Do you notice any pattern in the solution returned by the filtering and the smoothing problem? Explain why that is the case.

2.4.5 Prediction

We would like to compute the future probability of the state give observations up to some time

$$\mathrm{P}(X_k = x \mid Y_1, \ldots, Y_t) \quad \text{for } t < k.$$

Here is a typical scenario when you would need this estimate. Imagine that you are tracking the position of a car using images from your camera. You are using a deep network to detect the car in each image Y_k and since the neural network is quite slow, the car moves multiple time steps forward before you get the next observation. As you can appreciate, it would help us compute a more accurate estimate of the conditional probability of $X_k = x$ if we propagated the position of the car in between successive observations using our Markov chain. This is easy to do.

1. We compute the filtering estimate $\pi_t^f = \mathrm{P}(X_t = x \mid Y_1, \ldots, Y_t)$, using the forward algorithm.
2. Propagate the Markov chain forward for $k - t$ time-steps using π_t^f as the initial condition using

$$\pi_{i+1} = T' \pi_i.$$

2.4.6 Decoding: Viterbi's Algorithm

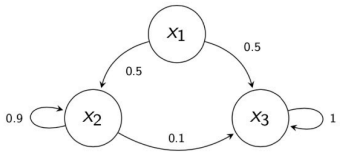
Both filtering and smoothing calculate the probability distribution of the state at time k. For instance, after recording a few observations, we can compute the probability distribution of the position of the car at each time instant. How do we get the most likely trajectory of the car? One option is to choose

$$\hat{X}_k = \operatorname*{argmax}_x \ \mathrm{P}(X_k = x \mid Y_1, \ldots, Y_t)$$

at each instant and output

$$(\hat{X}_1, \ldots, \hat{X}_t)$$

as the answer. This is however only the point-wise best estimate of the state. This sequence may not be the most likely trajectory of the Markov chain underlying our HMM. In the decoding problem, we are interested in computing the most likely state trajectory, not the point-wise most likely sequence of states. Let us take an example of the Whack-the-mole again. We will use a slightly different Markov chain shown below.



$$T = \begin{bmatrix} 0 & 0.5 & 0.5 \\ 0 & 0.9 & 0.1 \\ 0 & 0.1 & 0.9 \end{bmatrix} \qquad M = \begin{bmatrix} 0.5 & 0.5 \\ 0.9 & 0.1 \\ 0.1 & 0.9 \end{bmatrix}$$

There are three states x_1, x_2, x_3 with known initial distribution $\pi = (1, 0, 0)$ and transition probabilities and observations given by matrices T, M respectively. Let us say that we only have two observations $\{y_2, y_3\}$ this time and get the observation sequence

$$(2, 3, 3, 2, 2, 2, 3, 2, 3)$$

from our sensor. The filtering estimates are as follows.

t	x_1	x_2	x_3
1	**1.0000**	0	0
2	0	0.1000	**0.9000**
3	0	0.0109	**0.9891**
4	0	0.0817	**0.9183**
5	0	0.4165	**0.5835**
6	0	**0.8437**	0.1563
7	0	0.2595	**0.7405**
8	0	**0.7328**	0.2672
9	0	0.1771	**0.8229**

The most likely state at each instant is marked in blue. The point-wise most likely sequence of states is

$$(1, 3, 3, 3, 3, 2, 3, 2, 3).$$

Observe that this is not even feasible for the Markov chain. The transition from $x_3 \to x_2$ is not even possible, so this answer is clearly wrong. Let us look at the smoothing estimates.

t	x_1	x_2	x_3
1	**1.0000**	0	0
2	0	**0.6297**	0.3703
3	0	**0.6255**	0.3745
4	0	**0.6251**	0.3749
5	0	**0.6218**	0.3782
6	0	**0.5948**	0.4052
7	0	0.3761	**0.6239**
8	0	0.3543	**0.6457**
9	0	0.1771	**0.8229**

The point-wise most likely states in this case are feasible

$$(1, 2, 2, 2, 2, 2, 3, 3, 3).$$

Because the smoothing estimate at time k also takes into account the observations from the future $t > k$, it effectively eliminates the impossible transition from $x_3 \to x_2$. This is still not however the most likely trajectory.

We will exploit the Markov property again to calculate the most likely state trajectory recursively. Let us define the "decoding variables" as

$$\delta_k(x) = \max_{(x_1,\ldots,x_{k-1})} \mathrm{P}(X_1 = x_1, \ldots, X_{k-1} = x_{k-1}, X_k = x, Y_1, \ldots, Y_k); \tag{2.13}$$

this is the joint probability of the most likely state trajectory that ends at the state x at time k while generating observations $Y_1, \ldots, Y_k$. We can now see that

$$\delta_{k+1}(x) = \max_{x'} \; \delta_k(x')\, T_{x'x}\, M_{x,y_{k+1}}; \tag{2.14}$$

the joint probability that the most likely trajectory ends up at state x at time $k+1$ is the maximum of among the joint probabilities that end up at any state x' at time k multiplied by the one-step state transition $T_{x'x}$ and observation $M_{xy_{k+1}}$ probabilities. We would like to iterate upon this identity to find the most likely path. The key idea is to maintain a pointer to the parent state $\mathrm{parent}_k(x)$ of the most likely trajectory, i.e., the state from which you could have reached $X_k = x$ given observations. Let us see how.

Viterbi's algorithm First initialize

$$\delta_1(x) = \pi_x \; M_{xy_1}$$
$$\text{parent}_1(x) = \text{null}.$$

for all states x. For all times $k = 1, \ldots, t-1$, for all states x, update

$$\delta_{k+1}(x) = \max_{x'} \; \delta_k(x') \; T_{x'x} \; M_{x,y_{k+1}}$$
$$\text{parent}_{k+1}(x) = \operatorname*{argmax}_{x'} \; (\delta_k(x') \; T_{x'x}) \, .$$

The most likely final state is

$$\hat{x}_t = \operatorname*{argmax}_{x'} \; \delta_t(x')$$

and we can now backtrack using our parent pointers to find the most likely trajectory that leads to this state

$$\hat{x}_k = \text{parent}_{k+1}(\hat{x}_{k+1}).$$

The most likely trajectory given observations is

$$\hat{x}_1, \hat{x}_2, \ldots, \hat{x}_t$$

and the joint probability of this trajectory and all observations is

$$\text{P}(X_1 = \hat{x}_1, \ldots, X_t = \hat{x}_t, Y_1 = y_1, \ldots, Y_t = y_t) = \delta_t(\hat{x}_t).$$

This is a very widely used algorithm, both in robotics and in other areas such as speech recognition (given audio, find the most likely sentence spoken by the person), wireless transmission and reception, DNA analysis (e.g., the state of the Markov chain is the sequence ACTG. . . and our observations are functions of these states at periodic intervals). Its name comes from Andrew Viterbi who developed the algorithm in the late 60s, he is one of the founders of Qualcomm Inc.

Here is how Viterbi's algorithm would look like for our whack-the-model example.

$$\delta_1 = (0.6, 0, 0), \delta_2 = (0.012, 0.048, 0.18), \delta_3 = (0.0038, 0.0216, 0.0432)$$
$$\text{parent}_1 = (\text{null}, \text{null}, \text{null}), \text{parent}_2 = (1, 1, 1), \text{parent}_3 = (2, 3, 3).$$

The most likely path is the one that ends in 3 with joint probability 0.0432. This path is (1, 3, 3).

Let us also compute Viterbi's algorithm for a longer observation sequence.

t	x_1	x_2	x_3
1	0.5/0	0	0
2	0/1	0.025/1	0.225/1
3	0/1	0.00225/2	0.2025/3
4	0/1	0.0018225/2	0.02025/3
5	0/1	0.0014762/2	0.002025/3
6	0/1	0.0011957/2	0.0002025/3
7	0/1	0.00010762/2	0.00018225/3
8	0/1	8.717e-05/2	1.8225e-05/3
9	0/1	7.8453e-06/2	1.6403e-05/3

The most likely trajectory is

$$(1, 3, 3, 3, 3, 3, 3, 3, 3).$$

Notice that if we had only 8 observations, the most likely trajectory would be

$$(1, 2, 2, 2, 2, 2, 2, 2, 2).$$

What is the computational complexity of Viterbi's algorithm? It is linear in the time-horizon t and quadratic in the number of states in the Markov chain. We are plucking out the most likely trajectory out of $\text{card}(X)^t$ possible trajectories using the δ_k variables. Does this remind you of some other problem that you may have seen before?

Just like the Bayes filter, Viterbi's algorithm is typically implemented using $\log \delta_k(x)$ to avoid numerical underflows. This is particularly important for Viterbi's algorithm: since $\delta_k(x)$ is the probability of an entire state and observation trajectory it can get small very quickly for unlikely states (as we see in this example).

2.4.7 Shortest path on a Trellis graph

You may have seen Dijkstra's algorithm before that computes the shortest path to reach a node in the graph given costs of traversing every edge.

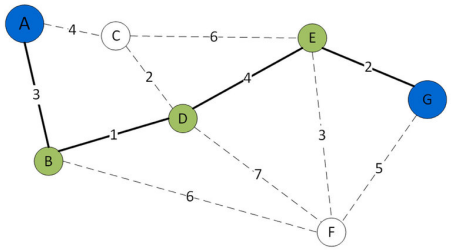


Figure 2.2: A graph with costs assigned to every edge. Dijkstra's algorithm finds the shortest path in this graph between nodes A and B using dynamic programming.

In the case of Viterbi's algorithm, we are also interested in finding the

most likely path. For example we can write our joint probabilities as

$$\mathrm{P}(X_1, X_2, X_3 \mid Y_1, Y_2, Y_3) = \frac{\mathrm{P}(Y_1 \mid X_1)\ \mathrm{P}(Y_2 \mid X_2)\ \mathrm{P}(Y_3 \mid X_3)\ \mathrm{P}(X_1)\,\mathrm{P}(X_2 \mid X_1)\,\mathrm{P}(X_3 \mid X_2)}{\mathrm{P}(Y_1, Y_2, Y_3)}.$$

$$\begin{aligned}\Rightarrow \log \mathrm{P}(X_1, X_2, X_3 \mid Y_1, Y_2, Y_3) &= \log \mathrm{P}(Y_1 \mid X_1) + \log \mathrm{P}(Y_2 \mid X_2) + \log \mathrm{P}(Y_3 \mid X_3) \\ &\quad + \log \mathrm{P}(X_1) + \log \mathrm{P}(X_2 \mid X_1) + \log \mathrm{P}(X_3 \mid X_2) - \log \mathrm{P}(Y_1, Y_2, Y_3).\end{aligned}$$

To find the most likely trajectory, we want to minimize $-\log \mathrm{P}(X_1, X_2, X_3 \mid Y_1, Y_2, Y_3)$. The term $\log \mathrm{P}(Y_1, Y_2, Y_3)$ does not depend on X_1, X_2, X_3 and is a constant as far as the most likely path given observations is concerned. We can now write down the "Trellis" graph as shown below.

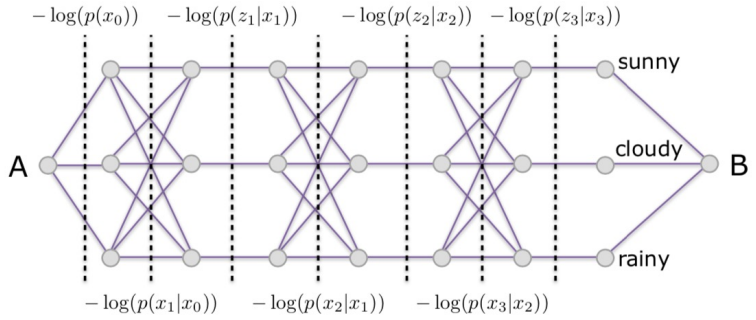


Figure 2.3: A Trellis graph for a 3-state HMM for a sequence of three observations. Disregard the subscript x_0.

Each edge is either the log-probability of the transition of the Markov chain, or it is the log-probability of the receiving the observation given a state. We create a dummy initial node A and a dummy terminal node B. The edge-costs of the final three states, in this case sunny/cloudy/rainy, are zero. The costs from node A to the respective states are the log-probabilities of the initial state distribution. Dijkstra's algorithm, which we will study in Module 2 in more detail, now gives the shortest path on the Trellis graph. This approach is the same as that of the Viterbi's algorithm: our parent pointers $\text{parent}_k(x)$ are the parent nodes in Dijkstra's algorithm and our delta variables $\delta_k(x)$ is the cost of each node in the Trellis graph maintained by the Dijkstra's algorithm.

2.5 Learning an HMM from observations

In the previous sections, given an HMM that had an initial distribution π for the Markov chain, a transition matrix T for the Markov chain and an observation matrix M

$$\lambda = (\pi, T, M)$$

we computed various quantities such as

$$\mathrm{P}(Y_1, \ldots, Y_t;\ \lambda)$$

for an observation sequence $Y_1, \ldots, Y_t$ of the HMM. Given an observation sequence, we can also go back and update our HMM to make this observation sequence more likely. This is the simplest instance of *learning* an HMM. The prototypical problem to imagine that our original HMM λ comes from is our knowledge of the original problem (say a physics model of the dynamics of a robot and its sensors). Given more data, namely the observations, we want to update this model. The most natural way to update the model is to maximize the likelihood of observations given our model, i.e.,

$$\lambda^* = \operatorname*{argmax}_{\lambda} \mathrm{P}(Y_1, \ldots, Y_t;\ \lambda).$$

This is known as maximum-likelihood estimation (MLE). In this section we will look at the Baum-Welch algorithm which solves the MLE problem iteratively. Given λ, it finds a new HMM $\lambda' = (\pi', T', M')$ (the $'$ denotes a new matrix, not the transpose here) such that

$$\mathrm{P}(Y_1, \ldots, Y_t;\ \lambda') > \mathrm{P}(Y_1, \ldots, Y_t;\ \lambda).$$

Let us consider a simple problem. We are going to imagine that the FBI is trying to catch the dangerous criminal Keyser Soze who is known to travel between two cities Los Angeles (LA) which will be state x_1 and New York City (NY) which will be state x_2. The FBI initially have no clue about his whereabouts, so their initial belief on his location is uniform $\pi = [0.5, 0.5]$. His movements are modeled using a Markov chain

$$T = \begin{bmatrix} 0.5 & 0.5 \\ 0.5 & 0.5 \end{bmatrix},$$

e.g., if Soze is in LA, he is likely to stay in LA or go to NY with equal probability. The FBI can make observations about him, they either observe him to be in LA (y_1), NY (y_2) or do not observe anything at all (null, y_3).

$$M = \begin{bmatrix} 0.4 & 0.1 & 0.5 \\ 0.1 & 0.5 & 0.4 \end{bmatrix}.$$

Say that they received an observation sequence of 20 periods

(null, LA, LA, null, NY, null, NY, NY, NY, null, NY, NY, NY, NY, NY, null, null, LA, LA, NY).

Can we say something about the probability of Soze's movements? At each time k we can compute

$$\gamma_k(x) := \mathrm{P}(X_k = x \mid Y_1, \ldots, Y_t)$$

the smoothing probability. We can also compute the most likely state trajectory he could have taken given our observations using decoding. Let us focus on the smoothing probabilities $\gamma_k(x)$ as shown below.

t	LA	NY
1	0.5556	0.4444
2	0.8000	0.2000
3	0.8000	0.2000
...	...	...
18	0.8000	0.2000
19	0.8000	0.2000
20	0.1667	0.8333

The point-wise most likely sequence of states after doing so turns out to be

(LA, LA, LA, LA, NY, LA, NY, NY, NY, LA, NY, NY, NY, NY, NY, LA, LA, LA, LA, NY).

Notice how smoothing fills in the missing observations above.

Expected state visitation counts The next question we should ask is how should we update the model λ given this data. We are going to learn the entries of the state-transition using

$$T'_{x,x'} = \frac{\text{E[number of transitions from } x \text{ to } x']}{\text{E[number of times the Markov chain was in state } x]}.$$

What is the denominator, it is simply the sum of the probabilities that the Markov chain was at state x at time $1, 2, \ldots, t-1$ given our observations, i.e.,

$$\text{E[number of times the Markov chain was in state } x] = \sum_{k=1}^{t-1} \gamma_k(x).$$

The numerator is given in a similar fashion. We will define a quantity

$$\begin{aligned} \xi_k(x, x') &:= \text{P}(X_k = x, X_{k+1} = x' \mid Y_1, \ldots, Y_t) \\ &= \eta\ \alpha_k(x) T_{x,x'} M_{x',y_{k+1}}\ \beta_{k+1}(x'); \end{aligned} \tag{2.15}$$

Derive the expression for $\xi_k(x, x')$ for yourself.

where η is a normalizing constant such that $\sum_{x,x'} \xi_k(x, x') = 1$. Observe that ξ_k is the joint probability of X_k and X_{k+1}

$$\begin{aligned} \xi_k(x, x') &= \text{P}(X_{k+1} = x' \mid X_k = x, Y_1, \ldots, Y_t)\ \gamma_k(x) \\ &\neq T_{x,x'}\ \gamma_k(x) \\ &= \text{P}(X_{k+1} = x' \mid X_k = x)\ \text{P}(X_k = x \mid Y_1, \ldots, Y_t). \end{aligned}$$

The expected value of transitioning between states x and x' is

$$\text{E[number of transitions from } x \text{ to } x'] = \sum_{k=1}^{t-1} \xi_k(x, x').$$

This gives us our new state transition matrix, you will see in the homework that it comes to be

$$T' = \begin{bmatrix} 0.47023 & 0.52976 \\ 0.35260 & 0.64739 \end{bmatrix}.$$

This is a much better informed FBI than the other we had before beginning the problem where the transition matrix was all 0.5s.

The new initial distribution What is the new initial distribution for the HMM? Recall that we are trying to compute the best HMM given the observations, so if the initial distribution was

$$\pi = \mathrm{P}(X_1)$$

before receiving any observations from the HMM, it is now

$$\pi' = \mathrm{P}(X_1 \mid Y_1, \ldots, Y_t) = \gamma_1(x);$$

the smoothing estimate at the first time-step.

Updating the observation matrix We can use a similar logic at the expected state visitation counts to write

$$\begin{aligned} M'_{x,y} &= \frac{\mathrm{E}[\text{number of times in state } x, \text{ when observation was } y]}{\mathrm{E}[\text{number of times the Markov chain was in state } x]} \\ &= \frac{\sum_{k=1}^{t} \gamma_k(x) \mathbf{1}_{\{y_k = y\}}}{\sum_{k=1}^{t} \gamma_k(x)}. \end{aligned}$$

You will see in your homework problem that this matrix comes up to be

$$M' = \begin{bmatrix} 0.39024 & 0.20325 & 0.40650 \\ 0.06779 & 0.706214 & 0.2259 \end{bmatrix}.$$

Notice how the observation probabilities for the unknown state y_3 have gone down because the Markov chain does not have those states.

The ability to start with a rudimentary model of the HMM and update it using observations is quite revolutionary. Baum et al. proved in the paper Baum, Leonard E., et al. "A maximization technique occurring in the statistical analysis of probabilistic functions of Markov chains." The annals of mathematical statistics 41.1 (1970): 164-171. Discuss the following questions:

- When do we stop in our iterated application of the Baum-Welch algorithm?
- Are we always guaranteed to find the same HMM irrespective of our initial HMM?
- If our initial HMM λ is the same, are we guaranteed to find the same HMM λ' across two different iterations of the Baum-Welch algorithm?
- How many observations should we use to update the HMM?

Chapter 3

Kalman Filter and its variants

Reading

1. Barfoot, Chapter 3, 4 for Kalman filter
2. Thrun, Chapter 3 for Kalman filter, Chapter 4 for particle filters
3. Russell Chapter 15.4 for Kalman filter

Hidden Markov Models (HMMs) which we discussed in the previous chapter were a very general class of models. As a consequence algorithms for filtering, smoothing and decoding that we prescribed for the HMM are also very general. In this chapter we will consider the situation when we have a little more information about our system. Instead of writing the state transition and observation matrices as arbitrary matrices, we will use the framework of linear dynamical systems to model them better. Since we know the system a bit better, algorithms that we prescribe for these models for solving filtering, smoothing and decoding will also be more efficient. We will almost exclusively focus on the filtering problem in this chapter. The other two, namely smoothing and decoding, can also be solved easily using these ideas but are less commonly used for these systems.

3.1 Background

Multi-variate random variables and linear algebra For d-dimensional random variables $X, Y \in \mathbb{R}^d$ we have

$$\mathrm{E}[X + Y] = \mathrm{E}[X] + \mathrm{E}[Y];$$

this is actually more surprising than it looks, it is true regardless of whether X, Y are correlated. The covariance matrix of a random variable is defined as

$$\text{Cov}(X) = \text{E}[(X - \text{E}[X])\,(X - \text{E}[X])^\top];$$

we will usually denote this by $\Sigma \in \mathbb{R}^{d\times d}$. Note that the covariance matrix is, by construction, symmetric and positive semi-definite. This means it can be factorized as

$$\Sigma = U\Lambda U^\top$$

where $U \in \mathbb{R}^{d\times d}$ is an orthonormal matrix (i.e., $UU^\top = I$) and Λ is a diagonal matrix with non-negative entries. The trace of a matrix is the sum of its diagonal entries. It is also equal to the sum of its eigenvalues, i.e.,

$$\text{tr}(\Sigma) = \sum_{i=1}^{d} \Sigma_{ii} = \sum_{i=1}^{d} \lambda_i(\Sigma)$$

where $\lambda_i(S) \geq 0$ is the i^{th} eigenvalue of the covariance matrix S. The trace is a measure of the uncertainty in the multi-variate random variable X, if X is a scalar and takes values in the reals then the covariance matrix is also, of course, a scalar $\Sigma = \sigma^2$.

A few more identities about the matrix trace that we will often use in this chapter are as follows.

- For matrices A, B we have

  $$\text{tr}(AB) = \text{tr}(BA);$$

  the two matrices need not be square themselves, only their product does.

- For $A, B \in \mathbb{R}^{m\times n}$

  $$\text{tr}\big(A^\top B\big) = \text{tr}\big(B^\top A\big) = \sum_{i=1}^{m}\sum_{j=1}^{n} B_{ij}A_{ij}.$$

  This operation can be thought of as taking the inner product between two matrices.

Gaussian/Normal distribution We will spend a lot of time working with the Gaussian/Normal distribution. The multi-variate d-dimensional Normal distribution has the probability density

❓ Why is it so ubiquitous?

$$f(x) = \frac{1}{\sqrt{\det(2\pi\Sigma)}}\ \exp\left\{-\frac{1}{2}(x-\mu)^T\Sigma^{-1}(x-\mu)\right\}$$

where $\mu \in \mathbb{R}^d, \Sigma \in \mathbb{R}^{d\times d}$ denote the mean and covariance respectively. You should commit this formula to memory. In particular remember that

$$\int_{x\in\mathbb{R}^d} \exp\left\{-\frac{1}{2}(x-\mu)^T\Sigma^{-1}(x-\mu)\right\}\,\mathrm{d}x = \sqrt{\det(2\pi\Sigma)}$$

which is simply expressing the fact that the probability density function integrates to 1.

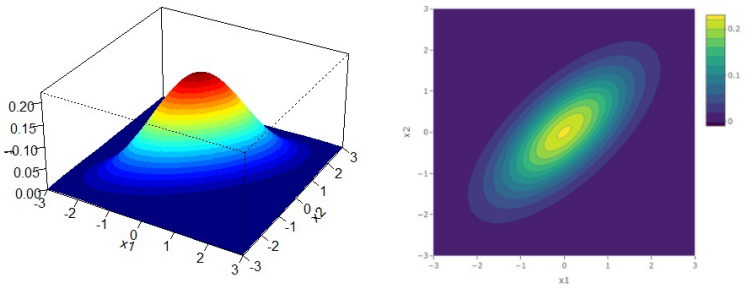


Figure 3.1: Probability density (left) and iso-probability contours (right) of a bi-variate Normal distribution. Warm colors denote regions of high probability.

Given two Gaussian rvs. $X, Y \in \mathbb{R}^d$ and $Z = X + Y$ we have

$$\mathrm{E}[Z] = \mathrm{E}[X + Y] = \mathrm{E}[X] + \mathrm{E}[Y]$$

with covariance

$$\mathrm{Cov}(Z) = \Sigma_Z = \Sigma_X + \Sigma_Y + \Sigma_{XY} + \Sigma_{YX}$$

where

$$\mathbb{R}^{d\times d} \ni \Sigma_{XY} = \mathrm{E}\left[(X - \mathrm{E}[X])\,(Y - \mathrm{E}[Y])^\top\right];$$

the matrix S_{YX} is defined similarly. If X, Y are independent (or uncorrelated) the covariance simplifies to

$$\Sigma_Z = \Sigma_X + \Sigma_Y.$$

If we have a linear function of a Gaussian random variable X given by $Y = AX$ for some *deterministic* matrix A then Y is also Gaussian with mean

$$\mathrm{E}[Y] = \mathrm{E}[AX] = A\,\mathrm{E}[X] = A\mu_X$$

and covariance

$$\begin{aligned}\mathrm{Cov}(Y) &= \mathrm{E}[(AX - A\mu_X)(AX - A\mu_X)^\top] \\ &= \mathrm{E}[A(X - \mu_X)(X - \mu_X)^\top A^\top] \\ &= A\,\mathrm{E}[(X - \mu_X)(X - \mu_X)^\top]A^\top \\ &= A\Sigma_X A^\top.\end{aligned} \tag{3.1}$$

This is an important result that you should remember.

3.2 Linear state estimation

With that background, let us now look at the basic estimation problem. Let $X \in \mathbb{R}^d$ denote the true state of a system. We would like to build an

estimator for this state, this is denote by

$$\hat{X}.$$

An estimator is any quantity that indicates our belief of what X is. The estimator is created on the basis of observations and we will therefore model it as a random variable. We would like the estimator to be unbiased, i.e.,

$$\mathrm{E}[\hat{X}] = X;$$

this expresses the concept that if we were to measure the state of the system many times, say using many sensors or multiple observations from the same sensor, the resultant estimator $\hat{X}$ is correct on average. The error in our belief is

$$\tilde{X} = \hat{X} - X.$$

The error is zero-mean $\mathrm{E}[\tilde{X}] = 0$ and its covariance $\Sigma_{\tilde{X}}$ is called the covariance of the estimator.

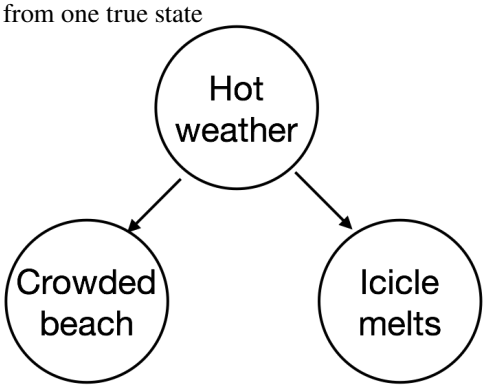
Optimally combining two estimators Let us now imagine that we have two estimators $\hat{X}_1$ and $\hat{X}_2$ for the same true state X. We will assume that the two estimators were created independently (say different sensors) and therefore are conditionally independent random variables given the true state X Say both of them are unbiased but each of them have a certain covariance of the error

$$\Sigma_{\tilde{X}_1} \text{ and } \Sigma_{\tilde{X}_2}.$$

We would like to combine the two to obtain a *better* estimate of what the state could be. *Better* can mean many different quantities depending upon the problem but in general in this course we are interested in improving the error covariance. Our goal is then

> Given two estimators $\hat{X}_1$ and $\hat{X}_2$ of the true state X combine them to obtain a new estimator
>
> $$\hat{X} = \text{some function}(\hat{X}_1, \hat{X}_2)$$
>
> which has the best error covariance $\mathrm{tr}(\Sigma_{\tilde{X}})$.

Conditionally independent observations from one true state



3.2.1 One-dimensional Gaussian random variables

Consider the case when $\hat{X}_1, \hat{X}_2 \in \mathbb{R}$ are Gaussian random variables with means μ_1, μ_2 and variances σ_1^2, σ_2^2 respectively. Assume that both are unbiased estimators of $X \in \mathbb{R}$. Let us combine them linearly to obtain a new estimator

$$\hat{X} = k_1\hat{X}_1 + k_2\hat{X}_2.$$

How should we pick the coefficients k_1, k_2? We would of course like the new estimator to be unbiased, so

$$\begin{aligned} \mathrm{E}[\hat{X}] = \mathrm{E}[k_1\hat{X}_1 + k_2\hat{X}_2] = (k_1 + k_2)X = X \\ \Rightarrow k_1 + k_2 = 1. \end{aligned}$$

The variance of the $\hat{X}$ is

$$\mathrm{Var}(\hat{X}) = k_1^2\sigma_1^2 + k_2^2\sigma_2^2 = k_1^2\sigma_1^2 + (1-k_1)^2\sigma_2^2.$$

The optimal k_1 that leads to the smallest variance is thus given by

$$k_1 = \frac{\sigma_2^2}{\sigma_1^2 + \sigma_2^2}.$$

We set the derivative of $\mathrm{Var}(\hat{X})$ with respect to k_1 to zero to get this. The final estimator is

$$\hat{X} = \frac{\sigma_2^2}{\sigma_1^2 + \sigma_2^2}\hat{X}_1 + \frac{\sigma_1^2}{\sigma_1^2 + \sigma_2^2}\hat{X}_2. \tag{3.2}$$

It is unbiased of course and has variance

$$\sigma_{\hat{X}}^2 = \frac{\sigma_1^2\sigma_2^2}{\sigma_1^2 + \sigma_2^2}.$$

Notice that since $\sigma_2^2/(\sigma_1^2 + \sigma_2^2) < 1$, the variance of the new estimator is smaller than either of the original estimators. This is an important fact to remember, combining two estimators *always* results in a better estimator.

Some comments about the optimal combination.

- It is easy to see that if $\sigma_2 \gg \sigma_1$ then the corresponding estimator, namely $\hat{X}_2$ gets less weight in the combination. This is easy to understand, if one of our estimates is very noisy, we should rely less upon it to obtain the new estimate. In the limit that $\sigma_2 \to \infty$, the second estimator is not considered at all in the combination.
- If $\sigma_1 = \sigma_2$, the two estimators are weighted equally and since $\sigma_{\hat{X}}^2 = \sigma_1^2/2$ the variance reduces by half after combination.
- The minimal variance of the combined estimator is not zero. This is easy to see because if we have two noisy estimates of the state, combining them need not lead to us knowing the true state with certainty.

3.2.2 General case

Let us now perform the same exercise for multi-variate Gaussian random variables. We will again combine the two estimators linearly to get

$$\hat{X} = K_1\hat{X}_1 + K_2\hat{X}_2$$

where $K_1, K_2 \in \mathbb{R}^{d\times d}$ are matrices that we would like to choose. In order for the estimator to be unbiased we again have the condition

$$\begin{aligned}\mathrm{E}[\hat{X}] &= \mathrm{E}[K_1\hat{X}_1 + K_2\hat{X}_2] = (K_1+K_2)X = X\\ &\Rightarrow K_1 + K_2 = I_{d\times d}.\end{aligned}$$

The covariance of $\hat{X}$ is

$$\begin{aligned}\Sigma_{\tilde{X}} &= K_1\Sigma_1 K_1^\top + K_2\Sigma_2 K_2^\top\\ &= K_1\Sigma_1 K_1^\top + (I-K_1)\Sigma_2(I-K_1)^\top.\end{aligned}$$

Just like the minimized the variance in the scalar case, we will minimize the trace of this covariance matrix. We know that the original covariances Σ_1 and Σ_2 are symmetric. We will use the following identity for the partial derivative of a matrix product

$$\frac{\partial}{\partial A}\operatorname{tr}\big(ABA^\top\big) = 2AB \tag{3.3}$$

for a symmetric matrix B. Minimizing $\operatorname{tr}(\Sigma_{\tilde{X}})$ with respect to K_1 amounts to setting

$$\frac{\partial}{\partial K_1}\operatorname{tr}(\Sigma_{\tilde{X}}) = 0$$

which yields

$$\begin{aligned}0 &= K_1\Sigma_1 - (I-K_1)\Sigma_2\\ \Rightarrow K_1 &= \Sigma_2(\Sigma_1+\Sigma_2)^{-1} \text{ and } K_2 \quad = \Sigma_1(\Sigma_1+\Sigma_2)^{-1}.\end{aligned}$$

The optimal way to combine the two estimators is thus

$$\hat{X} = \Sigma_2(\Sigma_1+\Sigma_2)^{-1}\hat{X}_1 + \Sigma_1(\Sigma_1+\Sigma_2)^{-1}\hat{X}_2. \tag{3.4}$$

You should consider the similarities of this expression with the one for the scalar case in (3.2). The same broad comments hold, i.e., if one of the estimators has a very large variance, that estimator is weighted less in the combination.

3.2.3 Incorporating Gaussian observations of a state

Let us now imagine that we have a sensor that can give us observations of the state. The development in this section is analogous to our calculations in Chapter 2 with the recursive application of Bayes rule or the observation matrix of the HMM. We will consider a special type of sensor that gives observations

$$\mathbb{R}^p \ni Y = CX + \nu \tag{3.5}$$

which is a linear function of the true state $X \in \mathbb{R}^d$ with the matrix $C \in \mathbb{R}^{p\times d}$ being something that is unique to the particular sensor. This observation is not precise and we will model the sensor as having zero-mean Gaussian noise

$$\nu \sim N(0, Q)$$

of covariance $Q \in \mathbb{R}^{p \times p}$. Notice something important here, the dimensionality of the observations need not be the same as the dimensionality of the state. This should not be surprising, after all the the number of observations in the HMM need not be the same as the number of the states in the Markov chain.

We will solve the following problem. Given an existing estimator $\hat{X}'$ we want to combine it with the observation Y to update the estimator to $\hat{X}$, in the best way, i.e., in a way that gives the minimal variance. We will again use a linear combination

$$\hat{X} = K'\hat{X}' + KY.$$

Again we want the estimator to be unbiased, so we set

$$\begin{aligned} \mathrm{E}[\hat{X}] &= \mathrm{E}[K'\hat{X}' + KY] \\ &= K'X + K\,\mathrm{E}[Y] \\ &= K'X + K\,\mathrm{E}[CX + \nu] \\ &= K'X + KCX \\ &= X. \end{aligned}$$

to get that

$$\begin{aligned} I &= K' + KC. \\ \Rightarrow \hat{X} &= (I - KC)\hat{X}' + KY \\ &= \hat{X}' + K(Y - C\hat{X}'). \end{aligned} \tag{3.6}$$

This is special form which you will do well to remember. The old estimator $\hat{X}'$ gets an additive term $K(Y - C\hat{X}')$. For reasons that will soon become clear, we call this term

$$\text{innovation} = Y - C\hat{X}'.$$

Let us now optimize K as before to compute the estimator with minimal variance. We will make the following important assumption in this case.

> We will assume that the observation Y is independent of the estimator $\hat{X}'$ given X. This is a natural assumption because presumably our original estimator $\hat{X}'$ was created using past observations and the present observation Y is therefore independent of it given the state X.

The covariance of $\hat{X}$ is

$$\Sigma_{\tilde{X}} = (I - KC)\Sigma_{\tilde{X}'}(I - KC)^\top + KQK^\top.$$

We optimize the trace of $\Sigma_{\tilde{X}}$ with respect to K to get

$$\begin{aligned} 0 &= \frac{\partial}{\partial K} \operatorname{tr}(\Sigma_{\tilde{X}}) \\ 0 &= -2(I - KC)\Sigma_{\tilde{X}'} C^\top + 2KQ \\ \Rightarrow \Sigma_{\tilde{X}'} C^\top &= K(C\Sigma_{\tilde{X}'} C^\top + Q) \\ \Rightarrow K &= \Sigma_{\tilde{X}'} C^\top (C\Sigma_{\tilde{X}'} C^\top + Q)^{-1}. \end{aligned}$$

The matrix $K \in \mathbb{R}^{d \times p}$ is called the "Kalman gain" after Rudoph Kalman who developed this method in the 1960s.

Kalman gain This is an important formula and it helps to have a mnemonic and a slightly simpler notation to remember it by. If Σ' is the covariance of the previous estimator, Q is the covariance of the zero-mean observation and C is the matrix that gives the observation from the state, then the Kalman gain is

$$K = \Sigma_{\tilde{X}'} C^\top (C\Sigma_{\tilde{X}'} C^\top + Q)^{-1}. \tag{3.7}$$

and the new estimator for the state is

$$\hat{X} = \hat{X}' + K(Y - C\hat{X}').$$

The covariance of the updated estimator $\hat{X}$ is given by

$$\begin{aligned} \Sigma_{\tilde{X}} &= (I - KC)\Sigma_{\tilde{X}'} (I - KC)^\top + KQK^\top \\ &= \left(\Sigma_{\tilde{X}'}^{-1} + C^\top Q^{-1} C\right)^{-1}. \end{aligned} \tag{3.8}$$

If $C = I$, the Kalman gain is the same expression as the optimal coefficient in (3.4). This should not be surprising because the observation is an estimator for the state.

The second expression for $\Sigma_{\tilde{X}}$ follows by substituting the value of the Kalman gain K. Yet another way of remembering this equation equation is to notice that

$$\begin{aligned} \Sigma_{\tilde{X}}^{-1} &= \Sigma_{\tilde{X}'}^{-1} + C^\top Q^{-1} C \\ K &= \Sigma_{\tilde{X}}^{-1} C^\top Q^{-1} \\ \hat{X} &= \hat{X}' + \Sigma_{\tilde{X}}^{-1} C^\top Q^{-1} \left(Y - C\hat{X}'\right). \end{aligned} \tag{3.9}$$

ℹ Derive these expressions for the Kalman gain and the covariance yourself.

3.2.4 An example

Consider the scalar case when we have multiple measurements of some scalar quantity $x \in \mathbb{R}$ corrupted by noise.

$$y_i = x + \nu_i$$

where $y_i \in \mathbb{R}$ and the scalar noise $\nu_i \sim N(0,1)$ is zero-mean and standard Gaussian. Find the updated estimate of the state x after k such measurements; this means both the mean and the covariance of the state.

You can solve this in two ways, you can either use the measurement matrix C to be $\mathbb{1}_k = [1, \ldots, 1]$ to be a vector of all ones and apply the formula in (3.7) and (3.8) Show that the estimate $\hat{x}_k$ after k measurements has mean and covariance

$$\mathrm{E}[\hat{x}_k] = \frac{1}{k}\sum_{i=1}^{k} y_i.$$
$$\mathrm{Cov}(\hat{x}_k) = C^\top C^{-1} = \frac{1}{k}.$$

If we take one more measurement $y_{k+1} = x + \nu_{k+1}$ with noise $\nu_{k+1} \sim N(0, \sigma^2)$, show using (3.9) that

$$\mathrm{Cov}(\hat{x}_{k+1})^{-1} = \mathrm{Cov}(\hat{x}_k)^{-1} + \frac{1}{\sigma^2}$$
$$\Rightarrow \mathrm{Cov}(\hat{x}_{k+1}) = \frac{\sigma^2}{\sigma^2 k + 1}.$$

The updated mean using (3.9) again

$$\mathrm{E}[\hat{x}_{k+1}] = \hat{x}_k + \mathrm{Cov}(\hat{x}_{k+1})\frac{1}{\sigma^2}(y_{k+1} - \hat{x}_k)$$
$$= \hat{x}_k + \frac{y_{k+1} - \hat{x}_k}{\sigma^2 k + 1}.$$

You will notice that if the noise on the $k+1^{\text{th}}$ observation is very small, even after k observations, the new estimate fixates on the latest observation

$$\sigma \to 0 \Rightarrow \hat{x}_{k+1} \to y_{k+1}.$$

Similarly, if the latest observation is very noisy, the estimate does not change much

$$\sigma \to \infty \Rightarrow \hat{x}_{k+1} \to \hat{x}_k.$$

3.3 Background on linear and nonlinear dynamical systems

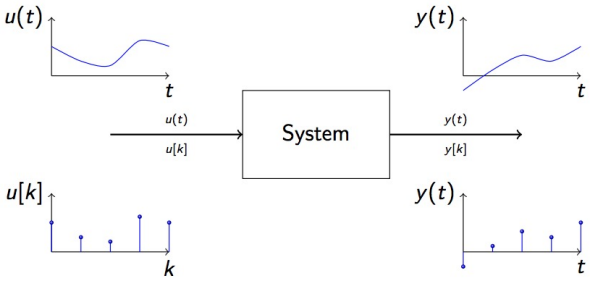
The true state X need not be static. We will next talk about models for how the state of the world evolves using ideas in dynamical systems.

A continuous-time signal is a function that associates to each time $t \in \mathbb{R}$ a real number $y(t)$. We denote signals by

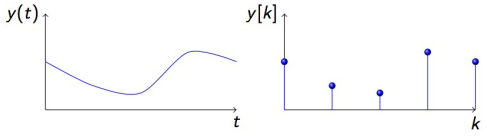
$$y : t \mapsto y(t).$$

Similarly a discrete-time signal is a function that associates to each integer k a real number $y(k)$, we have been denoting quantities like this by y_k.

A dynamical system is an operator (a box) that transforms an input signal $u(t)$ or u_k to an output $y(t)$ or y_k respectively. We call the former a continuous-time system and the latter a discrete-time system.



ℹ A continuous-time signal $y(t)$ and discrete-time signal y_k.



Almost always in robotics, we will be interested in systems that are temporally *causal*, i.e., the output at time t_0 is only a function the input *up to* time t_0. Analogously, the output at time k_0 for a discrete-time system is dependent only on the input up to time k_0. Most systems in the physical world at temporally causal.

❓ Can you give an example of a dynamical system that is non-causal? Think of how a DVD Ripper, or a pre-programmed acrobatic maneuver on a plane works.

State of a system We know that if the system is causal, in order to compute its output at a time t_0, we only need to know all the input from time $t = (-\infty, t_0]$. This is a lot of information. The concept of a state, about which we have been cavalier until now helps with this. The state $x(t_1)$ of a causal system at time t_1 is the information needed, together with the input u between times t_1 and t_2 to *uniquely compute* the output $y(t_2)$ at time t_2, for all times $t_2 \geq t_1$. In other words, the state of a system summarizes the whole history of what happened between $(-\infty, t_1)$.

Typically the state of a system is a d-dimensional vector in $\mathbb{R}^d$. The dimension of a system is the minimum d required to define a state.

❓ Discuss some examples of the state.

Is the state of a system uniquely defined?

3.3.1 Linear systems

A system is called a linear system if for any two input *signals* u_1 and u_2 and any two real numbers a, b

$$u_1 \to y_1$$
$$u_2 \to y_2$$
$$au_1 + bu_2 \to ay_1 + by_2.$$

Linearity is a very powerful property. For instance, it suggests that if we can decompose a complicated input into the sum of simple signals, then the output of the system is also a sum of the outputs of these simple signals. For example, if we can write the input as a Fourier series $u(t) = \sum_{i=0}^{\infty} a_i \cos(it) + b_i \sin(it)$ we can pass each of the terms in this summation to system and get the output of $u(t)$ by summing up the individual outputs.

Finite-dimensional systems can be written using a set of differential equations as follows. Consider the spring-mass system. If $z(t)$ denotes the position of the mass at time t and $u(t)$ is the force that we are applying upon it at time t, the position of the mass satisfies the differential equation

$$m\frac{\mathrm{d}^2 z(t)}{\mathrm{d}t^2} + c\frac{\mathrm{d}z(t)}{\mathrm{d}t} + kx(t) = u(t)$$
$$\text{or } m\ddot{z} + c\dot{z} + kz = u$$

in short. Here m is the mass of the block, c is the damping coefficient of the spring and k is the spring force constant. Let us define

$$z_1(t) := z(t)$$
$$z_2(t) := \frac{\mathrm{d}z(t)}{\mathrm{d}t}.$$

We can now rewrite the dynamics as

$$\begin{bmatrix}\dot{z_1}\\ \dot{z_2}\end{bmatrix} = \begin{bmatrix}0 & 1\\ -k/m & -c/m\end{bmatrix}\begin{bmatrix}z_1\\ z_2\end{bmatrix} + \begin{bmatrix}0\\ 1/m\end{bmatrix} u$$

3.3.2 Linear Time-Invariant (LTI) systems

If we define the state $x(t) = \begin{bmatrix}z_1(t)\\ z_2(t)\end{bmatrix}$, then the above equation can be written as

$$\dot{x}(t) = Ax(t) + Bu(t). \tag{3.10}$$

This is a linear system that takes in the input $u(t)$ and has a state $x(t)$. You can check the conditions for linearity to be sure. It is also a linear time-invariant (LTI) system because the matrices A, B do not change with time t. **The input $u(t)$ is also typically called the control (or action, or the control input) and essentially the second half of the course is about computing good controls.**

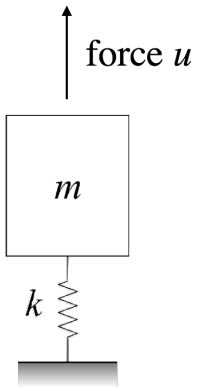
Since the state at time t encapsulates everything that happened to the system due to the inputs $\{u(-\infty), u(t)\}$, we can say that the system computes its output $y(t)$ as a function of the state $x(t)$ and the latest input $u(t)$

$$y(t) = \text{function}(x(t), u(t))$$

If this function is linear we have

$$y(t) = Cx(t) + Du(t). \tag{3.11}$$

A second-order spring mass system



The pair of equations (3.10) and (3.11) together are the so-called state-space model of an LTI system. The development for discrete-time systems is completely analogous, we will have

$$\begin{aligned} x_{k+1} &= A_{\Delta t} x_k + B_{\Delta t} u_k \\ y_k &= C x_k + D u_k. \end{aligned} \tag{3.12}$$

> We have used the subscript Δt to denote that these are discrete-time matrices and are different from the continuous-time ones in (3.10) and (3.11). This is an important point to keep in mind.

If the dynamics matrices A, B, C, D change with time, we have a time-varying system.

3.3.3 Nonlinear systems

Nonlinear systems are defined entirely analogously as linear systems. Imagine if we had a non-linear spring in the spring-mass system whereby the dynamics of the block was given by

$$m\ddot{z} + c\dot{z} + (k_1 z + k_2 z^2) = u.$$

The state of the system is still $x = [z_1, z_2]^\top$. But we cannot write this second-order differential equation as two first-order linear differential equations. We are forced to write

$$\dot{x} = \begin{bmatrix} \dot{z_1} \\ \dot{z_2} \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ -k_1/m - k_2 z_1/m & -c/m \end{bmatrix} \begin{bmatrix} z_1 \\ z_2 \end{bmatrix} + \begin{bmatrix} 0 \\ 1/m \end{bmatrix} u.$$

Such systems are called nonlinear systems. We will write them succinctly as

$$\begin{aligned} \dot{x} &= f(x, u) \\ y &= g(x, u). \end{aligned} \tag{3.13}$$

The function $f : \mathcal{X} \times \mathcal{U} \to \mathcal{X}$ that maps the state-space and the input space to the state-space is called the dynamics of the system. Analogously, for discrete-time nonlinear systems we will have

$$\begin{aligned} x_{k+1} &= f_{\Delta t}(x_k, u_k) \\ y_k &= g(x_k, u_k). \end{aligned}$$

Again the discrete-time nonlinear dynamics has a different equation than the corresponding one in (3.13).

❓ Is the nonlinear spring-mass system time-invariant?

3.4 Markov Decision Processes (MDPs)

Let us now introduce a concept called MDPs which is very close to Markov chains that we saw in the previous chapter. In fact, you are already implementing an MDP in your HW 1 problem on the Bayes filter.

MDPs are a model for the scenario when we do not completely know the dynamics $f(x_k, u_k)$.

This may happen for a number of reasons and it is important to appreciate them in order to understand the widespread usage of MDPs.

1. We did not do a good job of identifying the function $f : \mathcal{X} \times \mathcal{U} \to \mathcal{X}$. This may happen when you are driving a car on an icy road, if you undertake the same control as you do on a clean road, you might reach a different future state x_{k+1}.
2. We did not use the correct state-space $\mathcal{X}$. You could write down the state of the car as given by $(x, y, \theta, \dot{x}, \dot{y}, \dot{\theta})$ where x, y are the Euclidean co-ordinates of the car and θ is its orientation. This is not a good model for studying high-speed turns, which are affect by other quantities like wheel slip, the quality of the suspension etc. We may not even know the full state sometimes. This occurs when you are modeling how users interact with an online website like Amazon.com, you'd like to model the change in state of the user from "perusing stuff" to "looking stuff to buy it" to "buying it" but there are certainly many other variables that affect the user's behavior. As another example, consider the path that an airplane takes to go from Philadelphia to Los Angeles. This path is affected by the weather at all places along the route, it'd be cumbersome to incorporate the weather to find the shortest-time path for the airplane.
3. We did not use the correct control-space U for the controller. This is akin to the second point above. The gas pedal which one may think of as the control input to a car is only one out of the large number of variables that affect the running of the car's engine.

MDPs are a drastic abstraction of all the above situations. We

write

$$x_{k+1} = f(x_k, u_k) + \epsilon_k \tag{3.14}$$

where the "noise" ϵ_k is not under our control. The quantity ϵ_k is not arbitrary however, we will assume that

1. noise ϵ_k is a random variable and we know its distribution. For example, you ran your car lots of times on icy road and measured how the state x_{k+1} deviates from similar runs on a clean road. The difference between the two is modeled as ϵ_k. Note that the distribution of ϵ_k may be a function of time k.
2. noise at different timesteps $\epsilon_1, \epsilon_2, \ldots,$ is independent.

Instead of a deterministic transition for our system from x_k to x_{k+1}, we now have

$$x_{k+1} \sim \mathrm{P}(x_{k+1} \mid x_k, u_k).$$

which is just another way of writing (3.14).

The latter is a probability table of size $|\mathcal{X}| \times |\mathcal{U}| \times |\mathcal{X}|$ akin to the transition matrix of a Markov chain except that there is a different transition matrix for every control $u \in \mathcal{U}$. The former version (3.14) is more amenable to analysis. MDPs can be alternatively called stochastic dynamical systems, we will use either names for them in this course. For completeness, let us note down that linear stochastic systems will be written as

$$x_{k+1} = Ax_k + Bu_k + \epsilon_k.$$

The moral of this section is to remember that as pervasive as noise seems in all problem formulations in this course, it models different situations depending upon the specific problem. Understanding where noise comes from is important for real-world applications.

Noise in continuous-time systems You will notice that we only talked about discrete-time systems with noise in (3.14). We can also certainly talk about continuous-time systems whose dynamics f we do not know precisely

$$\dot{x}(t) = f(x(t), u(t)) + \epsilon(t) \tag{3.15}$$

and model the gap in our knowledge as noise $\epsilon(t)$. While this may seem quite natural, it is mathematically very problematic. The hurdle stems from the fact that if we want $\epsilon(t)$ to be a random variable *at each time instant*, then the signal $\epsilon(t)$ may not actually exist, e.g., it may not even be continuous. Signals like $\epsilon(t)$ exist only in very special cases, one of them is called "Brownian motion" where the increment of the signal after

You should think about the state-space, control-space and the noise in the MDP for the Bayes filter problem in HW 1. Where do we find MDPs in real-life? There are lots of expensive robots in GRASP, e.g., a Kuka manipulator such as this https://www.youtube.com/watch?v=ym64NFCWO costs upwards of $100,000. Would you model it as a stochastic dynamical system?

infinitesimal time Δt is a Gaussian random variable

$$\epsilon(t + \Delta t) - \epsilon(t) = N(0, \Delta t).$$

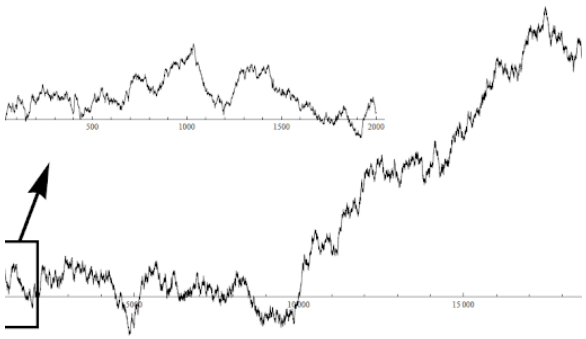


Figure 3.2: A typical Brownian motion signal $\epsilon(t)$. You can also see an animation at https://en.wikipedia.org/wiki/File:Brownian_Motion.ogv

We will not worry about this technicality in this course. We will talk about continuous-time systems with noise but with the implicit understanding that there is some underlying real-world discrete-time system and the continuous-time system is only an abstraction of it.

❓ Do continuous-time systems, stochastic or non-stochastic, exist in the real world? Consider the Kuka manipulator again, do you think the dynamics of this robot is a continuous-time system? Would you model it so?

3.4.1 Back to Hidden Markov Models

Since our sensors measure the state x of the world, it will be useful to think of the output y of a dynamical system as the observations from Chapter 2. This idea neatly ties back our development of dynamical systems to observations. Just like we considered an HMM with observation probability

$$\mathrm{P}(Y_k = y \mid X_k = x)$$

we will consider dynamical systems for which we do not precisely know how the output computation. We will model the gap in our knowledge of the exact observation mechanism as the output being a noisy function of the state. This is denoted as

$$y_k = g(x_k) + \nu_k. \tag{3.16}$$

The noise ν_k is similar to the noise in the dynamics ϵ_k in (3.14). Analogously, we can also have noise in the observations of a linear system

$$y_k = Cx_k + Du_k + \nu_k.$$

Hidden Markov Models with underlying MDPs/Markov chains

ℹ Observation noise and dynamics noise are different in subtle ways. The former may not always be due to our poor modeling. For instance, the process by which a camera acquires its images has some inherent noise. You may have seen a side-by-side comparison of different cameras using their ISOs

An image taken from a camera with low lighting has a lot of "noise". What causes this noise?

and stochastic dynamical systems with noisy observations are two different ways to think of the same concept, namely getting observations across time about the true state of a dynamic world.

In the former we have

$$\begin{aligned} \text{(state transition matrix)} \quad & \mathrm{P}(X_{k+1} = x' \mid X_k = x, u_k = u) \\ \text{(observation matrix)} \quad & \mathrm{P}(Y_k = y \mid X_k = x), \end{aligned}$$

while in the latter we have

$$\begin{aligned} \text{(nonlinear dynamics)} \quad & x_{k+1} = f(x_k, u_k) + \epsilon_k \\ \text{(nonlinear observation model)} \quad & y_k = g(x_k) + \nu_k, \end{aligned}$$

or

$$\begin{aligned} \text{(linear dynamics)} \quad & x_{k+1} = Ax_k + Bu_k + \epsilon_k \\ \text{(linear observation model)} \quad & y_k = Cx_k + Du_k + \nu_k. \end{aligned}$$

HMMs are easy to use for certain kinds of problems, e.g., speech-to-text, or a robot wandering in a grid world (like the Bayes filter problem in HW 1). Dynamical systems are more useful for certain other kinds of problems, e.g., a Kuka manipulator where you can use Newton's laws to simply write down the functions f, g.

ℹ You will agree that creating the state-transition matrix for the Bayes filter problem in HW 1 was really the hardest part of the problem. If the state-space were continuous and not a discrete cell-based world, you could have written the dynamics very easily in one line of code.

3.5 Kalman Filter (KF)

We will now introduce the Kalman Filter. It is the analog of the Bayes filter from the previous chapter. This is by far the most important algorithm in robotics and it is hard to imagine running any robot without the Kalman fiter or some variant of it.

Consider a linear dynamical system with linear observations

$$\begin{aligned} x_{k+1} &= Ax_k + Bu_k + \epsilon_k \\ y_k &= Cx_k + \nu_k. \end{aligned} \tag{3.17}$$

where the noise vectors

$$\begin{aligned} \epsilon_k &\sim N(0, R) \\ \nu_k &\sim N(0, Q) \end{aligned}$$

are both zero-mean and Gaussian with covariances R and Q respectively. We have also assumed that $D = 0$ because typically the observations do not depend on the control.

ℹ We will assume that the distribution of noise ϵ_k, ν_k does not change with time k. If it does change in your problem, you will see that following equations are quite easy to modify.

Our goal is to compute the best estimate of the state after multiple

observations

$$\mathrm{P}(x_k \mid y_1, \ldots, y_k).$$

This is the same as the filtering problem that we solved for Hidden Markov Models. Just like we used the forward algorithm to compute the filtering estimate recursively, we are going to use our development of the Kalman gain to incorporate a new observation recursively.

3.5.1 Step 0: Observing that the state estimate at any timestep should be a Gaussian

Maintaining the entire probability distribution $\mathrm{P}(x_k \mid y_1, \ldots, y_k)$ is difficult now, as opposed to the HMM with a finite number of states. We will exploit the following important fact. If we assume that the initial distribution of x_0 was a Gaussian, since all operations in (3.17) are linear, our new estimate of the state $\hat{x}_k$ at time k is also a Gaussian

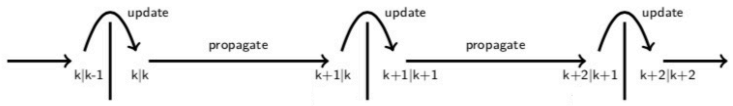
$$\hat{x}_{k|k} \sim \mathrm{P}(x_k \mid y_1, \ldots, y_k) \equiv N(\mu_{k|k}, \Sigma_{k|k}).$$

The subscript

$$\hat{x}_{k+1|k}$$

denotes that the quantity being talked about, i.e., $\hat{x}_{k+1|k}$, or others like $\mu_{k+1|k}$, is of the $(k+1)^{\text{th}}$ timestep and was calculated on the basis of observations up to (and including) the k^{th} timestep. We will therefore devise recursive updates to obtain $\mu_{k+1|k+1}, \Sigma_{k+1|k+1}$ using their old values $\mu_{k|k}, \Sigma_{k|k}$. We will imagine that our initial estimate for the state $\hat{x}_{0|0}$ has a known distribution

$$\hat{x}_{0|0} \sim N(\mu_{0|0}, \Sigma_{0|0}).$$



3.5.2 Step 1: Propagating the dynamics by one timestep

Suppose we had an estimate $\hat{x}_{k|k}$ after k observations/time-steps. Since the dynamics is linear, we can use the prediction problem to compute the estimate of the state at time $k+1$ before the next observation arrives

$$\mathrm{P}(x_{k+1} \mid y_1, \ldots, y_k).$$

From the first equation of (3.17), this is given by

$$\hat{x}_{k+1|k} = A\hat{x}_{k|k} + Bu_k + \epsilon_k$$

Notice that the subscript on the left-hand side is $k+1 \mid k$ because we did not take into account the observation at timestep $k+1$ yet. The mean and covariance of this estimate are given by

$$\begin{aligned} \mu_{k+1|k} = \mathrm{E}[\hat{x}_{k+1|k}] &= \mathrm{E}[A\hat{x}_{k|k} + Bu_k + \epsilon_k] \\ &= A\mu_{k|k} + Bu_k. \end{aligned} \tag{3.18}$$

We can also calculate the covariance of the estimate $\hat{x}_{k+1|k}$ to see that

$$\begin{aligned} \Sigma_{k+1|k} = \mathrm{Cov}(\hat{x}_{k+1|k}) &= \mathrm{Cov}(A\hat{x}_{k|k} + Bu_k + \epsilon_k) \\ &= A\Sigma_{k|k}A^\top + R, \end{aligned} \tag{3.19}$$

using our calculation in (3.1).

ℹ Observe that even if we knew the state dynamics precisely, i.e., if $R = 0$, we still have a non-trivial propagation equation for $\Sigma_{k+1|k}$.

3.5.3 Step 2: Incorporating the observation

After the dynamics propagation step, our estimate of the state is $\hat{x}_{k+1|k}$, this is the state of the system that we believe is true after k observations. We should now incorporate the latest observation y_{k+1} to update this estimate to get

$$\mathrm{P}(x_{k+1} \mid y_1, \ldots, y_k, y_{k+1}).$$

This is exactly the same problem that we saw in Section 3.2.3. Given the measurement

$$y_{k+1} = Cx_{k+1} + \nu_{k+1}$$

we first compute the Kalman gain K_{k+1} and the updated mean of the estimate as

$$\begin{aligned} K_{k+1} &= \Sigma_{k+1|k}C^\top \left(C\Sigma_{k+1|k}C^\top + Q\right)^{-1} \\ \mu_{k+1|k+1} &= \mu_{k+1|k} + K_{k+1}\left(y_{k+1} - C\mu_{k+1|k}\right). \end{aligned} \tag{3.20}$$

The covariance is given by our same calculation again

$$\begin{aligned} \Sigma_{k+1|k+1} &= \left(I - K_{k+1}C\right)\Sigma_{k+1|k}, \text{ or} \\ &= \left(I - K_{k+1}C\right)\Sigma_{k+1|k}\left(I - K_{k+1}C\right)^\top + K_{k+1}QK_{k+1}^\top, \text{ or} \\ &= \left(\Sigma_{k+1|k}^{-1} + C^\top Q^{-1}C\right)^{-1}. \end{aligned} \tag{3.21}$$

The second expression is known as Joeseph's form and is numerically more stable than the other expressions.

The new estimate of the state is

$$\hat{x}_{k+1|k+1} \sim \mathrm{P}(x_{k+1} \mid y_1, \ldots, y_{k+1}) \equiv N(\mu_{k+1|k+1}, \Sigma_{k+1|k+1}).$$

and we can again proceed to Step 1 for the next timestep.

3.5.4 Discussion

There are several important observations to make and remember about the Kalman Filter (KF).

- **Recursive updates to compute the best estimate given all past observations.** The KF is a recursive filter (just like the forward algorithm for HMMs) and incorporates observations one by one. The estimate that it maintains, namely $\hat{x}_{k+1|k+1}$, depends upon all past observations

$$\hat{x}_{k+1|k+1} \sim \mathrm{P}(x_{k+1} \mid y_1, \ldots, y_{k+1}).$$

  We have simply *computed* the estimate recursively.

- **Optimality of the KF for linear systems with Gaussian noise.** The KF is optimal in the following sense. Imagine if we had access to all the observations $y_1, \ldots, y_k$ beforehand and computed some other estimate

$$\hat{x}_{k|k}^{\text{fancy filter}} = \text{some function}(\hat{x}_{0|0}, y_1, \ldots, y_k).$$

  We use some other fancy method to design this estimator, e.g., nonlinear combination of the observations or incorporating observations across multiple timesteps together etc. to obtain something that has the smallest error with respect to the true state x_k

$$\mathrm{tr}\left(\mathop{\mathrm{E}}_{\epsilon_1,\ldots,\epsilon_k,\nu_1,\ldots,\nu_k} \left[(\hat{x}_{k|k}^{\text{fancy filter}} - x_k)(\hat{x}_{k|k}^{\text{fancy filter}} - x_k)^\top \right] \right). \tag{3.22}$$

  Then this estimate would be exactly the same as that of the KF

$$\hat{x}_{k|k}^{\text{fancy filter}} = \hat{x}_{k|k}^{\text{KF}}.$$

  This is a deep fact. First, the KF estimate was created recursively and yet we can do no better than it with our fancy estimator. This is analogous to the fact that the forward algorithm computes the correct filtering estimate even if it incorporates observations one by one recursively. Second, the KF combines the new observation and the old estimate linearly in (3.20). You could imagine that there is some other way to incorporate new observations, but it turns out that for linear dynamical systems with Gaussian noise, the KF is the best solution, we can do no better.

- **The KF is the best linear filter.** If we had a nonlinear dynamical system or a non-Gaussian noise with a linear dynamics/observations, there are other filters that can give a smaller error (3.22) than the KF. In the next section, we will take a look at one such example. However, even in these cases, the KF is the *best linear filter*.

- **Assumptions that are implicit in the KF**. We assumed that both the dynamics noise ϵ_κ and the observation noise ν_{k+1} are uncorrelated

with the estimate $\hat{x}_{k+1|k}$ computed prior to them (where did we use these assumptions?). This implicitly assumes that dynamics and observation noise are "white", i.e., uncorrelated in time

$$\mathrm{E}[\epsilon_k \, \epsilon_{k'}^\top] = 0 \quad \text{for all } k, k'$$
$$\mathrm{E}[\nu_k \, \nu_{k'}^\top] = 0 \quad \text{for all } k, k'.$$

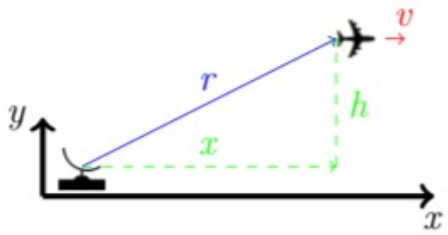
The Wikipedia webpage at https://en.wikipedia.org/wiki/Kalman_filter#Example_application,_technical gives a simple example of a Kalman Filter.

❓ How should one modify the KF equations if we have multiple sensors in a robot, each coming in at different frequencies?

3.6 Extended-Kalman Filter (EKF)

The KF heavily exploits the fact that our dynamics/measurements are linear. For most robots, both of these are nonlinear. The Extended-Kalman Filter (EKF) is a modification of the KF to handle such situations.

Example of a nonlinear dynamical system The state of most real problems evolves as a nonlinear function of their current state and control. This is a the same for sensors such as cameras measure a nonlinear function of the state. We will first see how to linearize a given nonlinear system shown below.



We have a radar sensor that measures the distance of the plane r from the radar trans-receiver up to noise ν. We would like to measure its distance x and height h. If the plane travels with a constant velocity, we have

$$\dot{x} = v, \text{ and } \dot{v} = 0,$$

and

$$r = \sqrt{x^2 + h^2}.$$

Since we do not really know how the plane might change its altitude, let's assume that it maintains a constant altitude

$$\dot{h} = 0.$$

The above equations are our model for how the state of the airplane evolves and could of course be wrong. As we discussed, we will model the

discrepancy as noise.

$$\begin{bmatrix} \dot{x_1} \\ \dot{x_2} \\ \dot{x_3} \end{bmatrix} = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} + \epsilon;$$
$$r = \sqrt{x_1^2 + x_3^2} + \nu;$$

here $x_1 \equiv x$, $x_2 \equiv v$ and $x_3 = h$, and $\epsilon \in \mathbb{R}^3, \nu \in \mathbb{R}$ are zero-mean Gaussian noise. The dynamics in this case is linear but the observations are a nonlinear function of the state.

One way to use the Kalman Filter for this problem is to linearize the observation equation around some state, say $x_1 = x_2 = x_3 = 0$ using the Taylor series

$$\begin{aligned} r_{\text{linearized}} &= r(0,0,0) + \left.\frac{\partial r}{\partial x_1}\right|_{x_1=0,x_3=0} (x_1 - 0) + \left.\frac{\partial r}{\partial x_3}\right|_{x_1=0,x_3=0} (x_3 - 0) \\ &= 0 + \left.\frac{2x_1}{2\sqrt{x_1^2 + x_3^2}}\right|_{x_1=0,x_3=0} x_1 + \left.\frac{2x_3}{2\sqrt{x_1^2 + x_3^2}}\right|_{x_1=0,x_3=0} x_3 \\ &= x_1 + x_3. \end{aligned}$$

In other words, upto first order in x_1, x_3, the observations are linear and we can therefore run the KF for computing the state estimate after k observations.

3.6.1 Propagation of statistics through a nonlinear transformation

Given a Gaussian random variable $\mathbb{R}^d \ni x \sim N(\mu_x, \Sigma_x)$, we saw how to compute the mean and covariance after an *affine* transformation $y = Ax$

$$\mathrm{E}[y] = A\,\mathrm{E}[x], \text{ and } \Sigma_y = A\Sigma_x A^\top.$$

If we had a nonlinear function of x

$$\mathbb{R}^p \ni y = f(x)$$

we can use the Taylor series by linearizing around the mean of x to approximate the first and second moments of y as follows.

$$\begin{aligned} y = f(x) &\approx f(\mu_x) + \left.\frac{\mathrm{d}f}{\mathrm{d}x}\right|_{x=\mu_x} (x - \mu_x) \\ &= Jx + (f(\mu_x) - J\mu_x). \end{aligned}$$

where we have defined the Jacobian matrix

$$\mathbb{R}^{p\times d} \ni J = \left.\frac{\mathrm{d}f}{\mathrm{d}x}\right|_{x=\mu_x}. \tag{3.23}$$

❓ You can try to perform a similar linearization for a simple model of a car

$$\begin{aligned} \dot{x} &= \cos\theta \\ \dot{y} &= \sin\theta \\ \dot{\theta} &= u. \end{aligned}$$

where x, y, θ are the XY-coordinates and the angle of the steering wheel respectively. This model is known as a Dubins car.

This gives

$$\begin{aligned} \mathrm{E}[y] &\approx \mathrm{E}[Jx + (f(\mu_x) - J\mu_x)] = f(\mu_x) \\ \Sigma_y &= \mathrm{E}[(y - \mathrm{E}[y])(y - \mathrm{E}[y])^\top] \approx J\Sigma_x J^\top. \end{aligned} \tag{3.24}$$

Observe how, up to first order, the mean μ_x is directly transformed by the nonlinear function f while the covariance Σ_x is transformed as if there were a linear operation $y \approx Jx$.

A simple example

$$y = \begin{bmatrix} y_1 \\ y_2 \end{bmatrix} = f\left(\begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}\right) = \begin{bmatrix} x_1^2 + x_2 x_3 \\ \sin x_2 + \cos x_3 \end{bmatrix}.$$

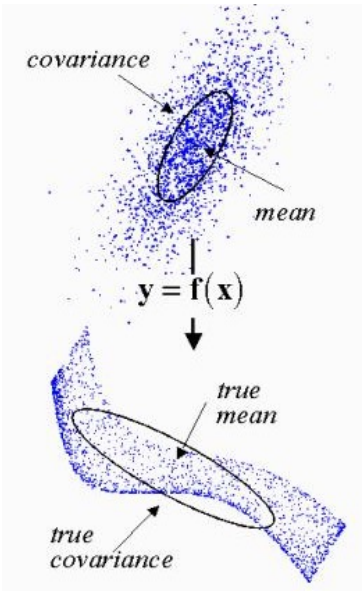
We have

$$\frac{\mathrm{d}f}{\mathrm{d}x} = \nabla f(x) = \begin{bmatrix} 2x_1 & x_3 & x_2 \\ 0 \cos x_2 & -\sin x_3 & \end{bmatrix}.$$

The Jacobian at $\mu_x = [\mu_{x_1}, \mu_{x_2}, \mu_{x_3}]^\top$ is

$$J = \nabla f(x)\Big|_{x=\mu_x} = \begin{bmatrix} 2\mu_{x_1} & \mu_{x_3} & \mu_{x_2} \\ 0 \cos \mu_{x_2} & -\sin \mu_{x_3} & \end{bmatrix}.$$

It is very important to remember that we are approximating the distribution of $\mathrm{P}(f(x))$ as a Gaussian. Even if x is a Gaussian random variable, the distribution of $y = f(x)$ need not be Gaussian. Indeed y is only Gaussian if f is an affine function of x.



3.6.2 Extended Kalman Filter

The above approach of linearizing the observations of the plane around the origin may lead to a lot of errors. This is because the point about which we linearize the system is fixed. We can do better by linearizing the system at each timestep. Let us say that we are given a nonlinear system

$$\begin{aligned} x_{k+1} &= f(x_k, u_k) + \epsilon \\ y_k &= g(x_k) + \nu. \end{aligned}$$

Can you say where will our linearized observation equation incur most error?

> The central idea of the Extended Kalman Filter (EKF) is to linearize a nonlinear system at each timestep k around the latest state estimate given by the Kalman Filter and use the resultant linearized dynamical system in the KF equations for the next timestep.

Step 1: Propagating the dynamics by one timestep

We will linearize the dynamics equation around the mean of the previous state estimate $\mu_{k|k}$

$$\begin{aligned} x_{k+1} &= f(x_k, u_k) + \epsilon \\ &\approx f(\mu_{k|k}, u_k) + \left.\frac{\partial f}{\partial x}\right|_{x=\mu_{k|k}} \left(x_k - \mu_{k|k}\right) + \epsilon_k. \end{aligned}$$

Let the Jacobian be

$$A(\mu_{k|k}) = \left.\frac{\partial f}{\partial x}\right|_{x=\mu_{k|k}}. \tag{3.25}$$

The mean and covariance of the EKF after the dynamics propagation step is therefore given by

$$\begin{aligned} \mu_{k+1|k} &= f(\mu_{k|k}, u_k) \\ \Sigma_{k+1|k} &= A\Sigma_{k|k}A^\top + R. \end{aligned} \tag{3.26}$$

It is worthwhile to notice the similarities of the above set of equations with (3.18) and (3.19). The mean $\mu_{k|k}$ is propagated using a nonlinear function f to get $\mu_{k+1|k}$, the covariance is propagated using the Jacobian $A(\mu_{k|k})$ which is recomputed using (3.25) at each timestep.

Step 2: Incorporating the observation

We have access to $\mu_{k+1|k}$ after Step 1, so we can linearize the nonlinear observations at this state.

$$\begin{aligned} y_{k+1} &= g(x_{k+1}) + \nu \\ &\approx g(\mu_{k+1|k}) + \left.\frac{\mathrm{d}g}{\mathrm{d}x}\right|_{x=\mu_{k+1|k}} (x_{k+1} - \mu_{k+1|k}) + \nu \end{aligned}$$

Again define the Jacobian

$$C(\mu_{k+1|k}) = \frac{\partial g}{\partial x}\bigg|_{x=\mu_{k+1|k}}. \tag{3.27}$$

Consider the fake observation which is a transformed version of the actual observation y_{k+1} (think of this as a new sensor or a post-processed version of the original sensor)

$$y'_{k+1} = y_{k+1} - g(\mu_{k+1|k}) + C\mu_{k+1|k} \approx Cx_{k+1}.$$

Our fake observation is a nice linear function of the state x_{k+1} and we can therefore use the Kalman Filter equations to incorporate this fake observation

$$\begin{aligned}\mu_{k+1|k+1} &= \mu_{k+1|k} + K(y'_{k+1} - C\,\mu_{k+1|k}) \\ \text{where } K &= \Sigma_{k+1|k}C^\top \left(C\Sigma_{k+1|k}C^\top + Q\right)^{-1}.\end{aligned}$$

Let us resubstitute our fake observation in terms of the actual observation y_{k+1}.

$$y'_{k+1} - C\,\mu_{k+1|k} = y_{k+1} - g(\mu_{k+1|k}),$$

to get the EKF equations for incorporating one observation

$$\begin{aligned}\mu_{k+1|k+1} &= \mu_{k+1|k} + K(y_{k+1} - g(\mu_{k+1|k})) \\ \Sigma_{k+1|k+1} &= (I - KC)\,\Sigma_{k+1|k}.\end{aligned} \tag{3.28}$$

The Extended Kalman Filter estimates the state of a nonlinear system by linearizing the dynamics and observation equations at each timestep.

1. Say we have the current estimate $\mu_{k|k}$ and $\Sigma_{k|k}$.
2. After a control input u_k the new estimate is

$$\begin{aligned}\mu_{k+1|k} &= f(\mu_{k|k}, u_k) \\ \Sigma_{k+1|k} &= A\Sigma_{k|k}A^\top + R.\end{aligned}$$

   where A depends on $\mu_{k|k}$.
3. We next incorporate an observation by linearizing the observation equations around $\mu_{k+1|k}$

$$\begin{aligned}K &= \Sigma_{k+1|k}C^\top \left(C\Sigma_{k+1|k}C^\top + Q\right)^{-1} \\ \mu_{k+1|k+1} &= \mu_{k+1|k} + K(y_{k+1} - g(\mu_{k+1|k})) \\ \Sigma_{k+1|k+1} &= (I - KC)\,\Sigma_{k+1|k}\end{aligned}$$

   where again C depends on $\mu_{k+1|k}$

Discussion

1. The EKF dramatically expands the applicability of the Kalman Filter. It can be used for most real systems, even with very complex models f, h. It is very commonly used in robotics and can handle nonlinear observations from complex sensors such as a LiDAR and camera easily. For instance, sophisticated augmented/virtual reality systems like Google ARCore/Snapchat/iPhone etc. (https://www.youtube.com/watch?v=cape_Af9j7w) run EKF to track the motion of the phone or of the objects in the image.

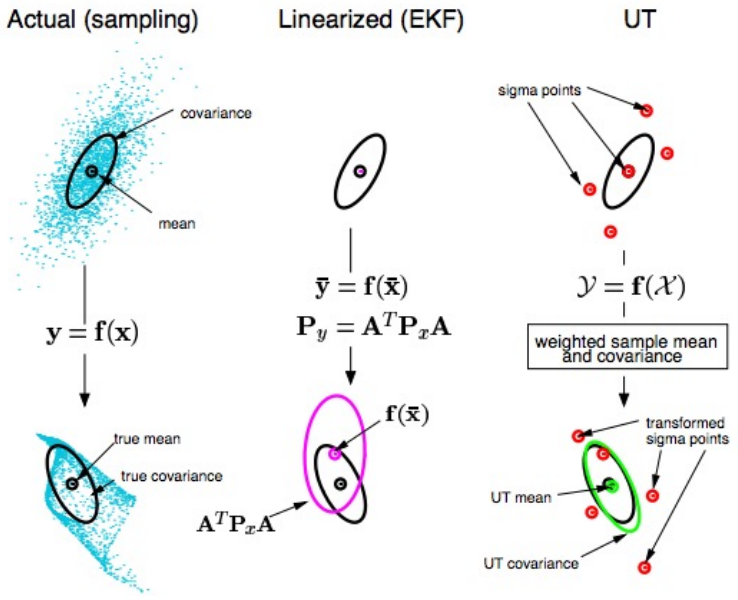
2. The KF was special because it is the optimal linear filter, i.e., KF estimates have the smallest mean squared error with respect to the true state for linear dynamical systems with Gaussian. The EKF is a clever application of KF to nonlinear systems but it no longer has this property. There do exist filters for nonlinear systems that will have a smaller mean-squared error than the EKF. We will look at some of them in the next section.

3. Linearization is the critical step in the implementation of the EKF and EKF state estimate can be quite inaccurate if the system is at a state where the linearized matrix A and the nonlinear dynamics $f(x_k, u_k)$ differ significantly. A common trick for handling this is to perform multiple steps of dynamics propagation using a continuous-time model of the system between successive observations. Say we have a system
$$\dot{x} = f(x(t), u(t)) + \epsilon(t)$$
where $\epsilon(t + \delta t) - \epsilon(t)$ is a Gaussian random variable $N(0, R\delta t)$ as $\delta \to 0$; see the section on Brownian motion for how to interpret noise in continuous-time systems. We can construct a discrete-time system from this as
$$\begin{aligned} x_{t+\Delta t} &= x(t) + f(x(t), u(t))\ \Delta t + \epsilon \\ &\equiv f^{\text{discrete-time}}(x(t), u(t)) + \epsilon. \end{aligned}$$
where $\epsilon \sim N(0, R\Delta t)$ is noise. This is now a discrete-time dynamics and we can perform Step 1 of the EKF multiple times to obtain a more accurate estimate of $\mu_{k+1|k}$ and $\Sigma_{k+1|k}$.

3.7 Unscented Kalman Filter (UKF)

Linearization of the dynamics in the EKF is a neat trick to use the KF equations. But as we said, this can cause severe issues in problems where the dynamics is very nonlinear. In this section, we will take a look at a powerful method to handle nonlinear dynamics that is better than linearization.

Let us focus on Step 1 which propagates the dynamics in the EKF.



We know that even if x is Gaussian (faint blue points in top left picture), the transformed variable $y = f(x)$ need not be Gaussian (faint blue points in bottom left). The EKF is really approximating the probability distribution $\mathrm{P}(x_{k+1} \mid y_1, \ldots, y_k)$ as a Gaussian; this distribution could be very different from a Gaussian. This is really the crux of the issue in filtering for nonlinear systems. This approximation, which happens because we are linearizing about the mean $\mu_{k|k}$.

Let us instead do the following:

1. Sample a few points from the Gaussian $N(\mu_{k|k}, \Sigma_{k|k})$ (red points in top right).
2. Transform *each of the points* using the nonlinear dynamics f (red points in bottom right).
3. Compute their mean and covariance to get $\mu_{k+1|k}$ and $\Sigma_{k+1|k}$. Notice how the green ellipse is slightly different than the black ellipse (which is the true mean and covariance). Both of these would be different from the mean and covariance obtained by linearization of f (middle column) but the green one is more accurate.

In general, we would need a large number of sample points (red) to

accurately get the mean and covariance of $y = f(x)$. The Unscented Transform (UT) uses a special set of points known as "sigma points" (these are the ones actually shown in red above) and transforms those points. Sigma points have the special property that the empirical mean of the transformed distribution (UT mean in the above picture) is close to the true mean up to third order; linearization is only accurate up to first order. The covariance (UT covariance) and true covariance also match up to third order.

3.7.1 Unscented Transform

Given a random variable $x \sim N(\mu_x, \Sigma_x)$, the Unscented Transform (UT) uses sigma points to compute an approximation of the probability distribution of the random variable $y = f(x)$.

Preliminaries: matrix square root. Given a symmetric matrix $\Sigma \in \mathbb{R}^{n \times n}$, the matrix square root of Σ is a matrix $S \in \mathbb{R}^{n \times n}$ such that

$$\Sigma = SS.$$

We can compute this via diagonalization as follows.

$$\begin{aligned}
\Sigma &= VDV^{-1} \\
&= V \begin{bmatrix} d_{11} & \cdots & 0 \\ 0 & & \\ \cdots & 0 & \\ 0 & \cdots & d_{nn} \end{bmatrix} V^{-1} \\
&= V \begin{bmatrix} \sqrt{d_{11}} & \cdots & 0 \\ 0 & & \\ \cdots & 0 & \\ 0 & \cdots & \sqrt{d_{nn}} \end{bmatrix}^2 V^{-1}.
\end{aligned}$$

We can therefore define

$$S = V \begin{bmatrix} \sqrt{d_{11}} & \cdots & 0 \\ 0 & & \\ \cdots & 0 & \\ 0 & \cdots & \sqrt{d_{nn}} \end{bmatrix} V^{-1}.$$

Notice that

$$SS = \left(VD^{1/2}V^{-1}\right)\left(VD^{1/2}V^{-1}\right) = VDV^{-1} = \Sigma.$$

We can also define the matrix square root using the Cholesky decomposition $\Sigma = LL^\top$ which is numerically more stable than computing the square root using the above expression. Recall that matrices L and Σ have the same eigenvectors. Typical applications of the Unscented Transform will use this method.

Given a random variable $\mathbb{R}^n \ni x \sim N(\mu, \Sigma)$, we will use the matrix square root to compute the sigma points as

$$\begin{aligned} x^{(i)} &= \mu + \sqrt{n\Sigma}_i^\top \\ x^{(n+i)} &= \mu - \sqrt{n\Sigma}_i^\top \\ &\text{for } i = 1, \dots, n, \end{aligned} \tag{3.29}$$

where $\sqrt{n\Sigma}_i$ is the i^{th} row of the matrix $\sqrt{n\Sigma}$. There are $2n$ sigma points

$$\left\{ x^{(1)}, \dots, x^{(2n)} \right\}$$

for an n-dimensional Gaussian. Each sigma point is assigned a weight

$$w^{(i)} = \frac{1}{2n}. \tag{3.30}$$

We then transform each sigma point to get the transformed sigma points

$$y^{(i)} = f(x^{(i)}).$$

The mean and covariance of the transformed random variable y can now be computed as

$$\begin{aligned} \mu_y &= \sum_{i=1}^{2n} w^{(i)} y^{(i)} \\ \Sigma_y &= \sum_{i=1}^{2n} w^{(i)} \left(y^{(i)} - \mu_y \right) \left(y^{(i)} - \mu_y \right)^\top. \end{aligned} \tag{3.31}$$

Example Say we have $x = \begin{bmatrix} r \\ \theta \end{bmatrix}$ with $\mu_x = [1, \pi/2]$ and $\Sigma_x = \begin{bmatrix} \sigma_r^2 & 0 \\ 0 & \sigma_\theta^2 \end{bmatrix}$. We would like to compute the probability distribution of $y = f(x) = \begin{bmatrix} r\cos\theta \\ r\sin\theta \end{bmatrix}$ which is a polar transformation. Since x is two-dimensional, we will have 4 sigma points with equal weights $w^{(i)} = 0.25$. The square root in the sigma point expression is

$$\sqrt{n\Sigma} = \begin{bmatrix} \sqrt{2}\sigma_r & 0 \\ 0 & \sqrt{2}\sigma_\theta \end{bmatrix}$$

and the sigma points are

$$\begin{aligned} x^{(1)} &= \begin{bmatrix} 1 \\ \pi/2 \end{bmatrix} + \begin{bmatrix} \sqrt{2}\sigma_r \\ 0 \end{bmatrix}, \quad x^{(3)} = \begin{bmatrix} 1 \\ \pi/2 \end{bmatrix} - \begin{bmatrix} \sqrt{2}\sigma_r \\ 0 \end{bmatrix} \\ x^{(2)} &= \begin{bmatrix} 1 \\ \pi/2 \end{bmatrix} + \begin{bmatrix} 0 \\ \sqrt{2}\sigma_\theta \end{bmatrix}, \quad x^{(4)} = \begin{bmatrix} 1 \\ \pi/2 \end{bmatrix} - \begin{bmatrix} 0 \\ \sqrt{2}\sigma_\theta \end{bmatrix}. \end{aligned}$$

❓ Compute the mean and covariance of y by linearizing the function $f(x)$.

The transformed sigma points are

$$
\begin{aligned}
y^{(1)} &= \begin{bmatrix} r^{(1)}\cos\theta^{(1)} \\ r^{(1)}\sin\theta^{(1)} \end{bmatrix} = \begin{bmatrix} 0 \\ 1+\sqrt{2}\sigma_r \end{bmatrix} \\
y^{(2)} &= \begin{bmatrix} r^{(2)}\cos\theta^{(2)} \\ r^{(2)}\sin\theta^{(2)} \end{bmatrix} = \begin{bmatrix} \cos\big(\pi/2+\sqrt{2}\sigma_\theta\big) \\ \sin\big(\pi/2+\sqrt{2}\sigma_\theta\big) \end{bmatrix} \\
y^{(3)} &= \begin{bmatrix} 0 \\ 1-\sqrt{2}\sigma_r \end{bmatrix} \\
y^{(4)} &= \begin{bmatrix} \cos\big(\pi/2-\sqrt{2}\sigma_\theta\big) \\ \sin\big(\pi/2-\sqrt{2}\sigma_\theta\big) \end{bmatrix}.
\end{aligned}
$$

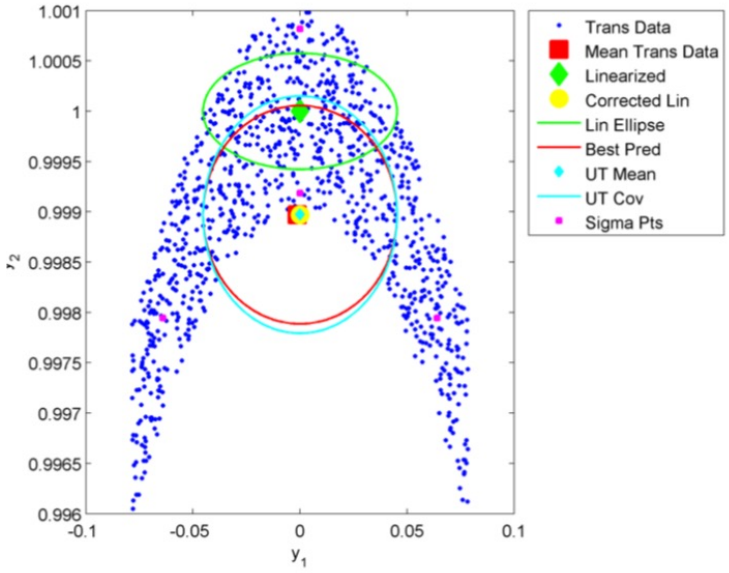


Figure 3.3: Note that the true mean is being predicted very well by the UT and is clearly a better estimate than the linearized mean.

3.7.2 The UT with tuning parameters

The UT is a basic template for a large suite of techniques that capture the covariance Σ_x as a set of points and transform those points through the nonlinearity. You will see many alternative implementations of the UT that allow for user-tunable parameters. For instance, sometimes the UT is implemented with an additional sigma point $x^{(0)} = \mu$ with weight $w^{(0)} = \frac{\lambda}{n+\lambda}$ and the weights of the other points are adjusted to be $w^{(i)} = \frac{1}{2(n+\lambda)}$ for a user-chosen parameter λ. You may also see people using one set of weights $w^{(i)}$ for computing the mean μ_y and and another set of weights for computing the covariance Σ_y.

❓ Are the transformed sigma points $y^{(i)}$ the sigma points of $\mathrm{P}(y) = N(\mu_y, \Sigma_y)$?

❓ We are left with a big lingering question. Why do you think this method is called the "unscented transform"?

3.7.3 Unscented Kalman Filter (UKF)

The Unscented Transform gives us a way to accurately estimate the mean and covariance of the transformed distribution through a nonlinearity. We can use the UT to modify the EKF to make it a more accurate state estimator. The resultant algorithm is called the Unscented Kalman Filter (UKF).

Step 1: Propagating the dynamics by one timestep Given our current state estimate $\mu_{k|k}$ and $\Sigma_{k|k}$, we use the UT to obtain the updated estimates $\mu_{k+1|k}$ and $\Sigma_{k+1|k}$. If $x^{(i)}$ are the sigma points with corresponding weights $w^{(i)}$ for the Gaussian $N(\mu_{k|k}, \Sigma_{k|k})$, we set

$$\begin{aligned}\mu_{k+1|k} &:= \sum_{i=1}^{2n} w^{(i)} f(x^{(i)}, u_k) \\ \Sigma_{k+1|k} &:= R + \sum_{i=1}^{2n} w^{(i)} \left(f(x^{(i)}) - \mu_{k+1|k}\right) \left(f(x^{(i)}) - \mu_{k+1|k}\right)^\top\end{aligned} \tag{3.32}$$

Step 2.1: Incorporating one observation The observation step is also modified using the UT. The key issue in this case is that we need a way to compute the Kalman gain in terms of the sigma points in the UT. We proceed as follows.

Using *new* sigma points $x^{(i)}$ for the updated state distribution $N(\mu_{k+1|k}, \Sigma_{k+1|k})$ with equal weights $w^{(i)} = 1/2n$, we first compute their mean after the transformation

$$\hat{y} = \sum_{i=1}^{2n} w^{(i)} g(x^{(i)}) \tag{3.33}$$

and covariances

$$\begin{aligned}\Sigma_{yy} &:= Q + \sum_{i=1}^{2n} w^{(i)} \left(g(x^{(i)}) - \hat{y}\right) \left(g(x^{(i)}) - \hat{y}\right)^\top \\ \Sigma_{xy} &:= \sum_{i=1}^{2n} w^{(i)} \left(x^{(i)} - \mu_{k+1|k}\right) \left(g(x^{(i)}) - \hat{y}\right)^\top.\end{aligned} \tag{3.34}$$

Step 2.2: Computing the Kalman gain Until now we have written the Kalman gain using the measurement matrix C. We will now discuss a more abstract formulation that gives the same expression.

Say we have a random variable x with known μ_x, Σ_x and get a new observation y. We saw how to incorporate this new observation to obtain a better estimator for x in Section 3.2.3. We will go through a similar analysis as before but in a slightly different fashion, one that does not involve the matrix C. Let

$$z = \begin{bmatrix} x \\ y \end{bmatrix}$$

and $\mu_z = \begin{bmatrix}\mu_x & \mu_y\end{bmatrix}$ and

$$\Sigma_z = \begin{bmatrix}\Sigma_{xx} & \Sigma_{xy} \\ \Sigma_{yx} & \Sigma_{yy}\end{bmatrix}.$$

Finding the best (minimum mean squared error estimator) $\hat{x} = \mu_x + K(y - \mu_y)$ amounts to minimizing

$$\min_K \mathrm{E}\left[\mathrm{tr}\left\{(\hat{x} - x)(\hat{x} - x)^\top\right\}\right].$$

This is called the least squares problem, which you have seen before perhaps in slightly different notation. You can solve this problem to see that the best gain K is given by

$$K^* = \Sigma_{xy}\Sigma_{yy}^{-1}. \tag{3.35}$$

and this gain leads to the new covariance

$$(\hat{x} - x)(\hat{x} - x)^\top = \Sigma_{xx} - \Sigma_{xy}\Sigma_{yy}^{-1}\Sigma_{yx} = \Sigma_{xx} - K^*\Sigma_{yy}{K^*}^\top.$$

The nice thing about the Kalman gain in (3.35) is that we can compute it now using expressions of Σ_{xy} and Σ_{yy} in terms of the sigma points. This goes as as follows:

$$\begin{aligned} K^* &= \Sigma_{xy}\Sigma_{yy}^{-1} \\ \mu_{k+1|k+1} &= \mu_{k+1|k} + K\left(y_{k+1} - \hat{y}\right) \\ \Sigma_{k+1|k+1} &= \Sigma_{k+1|k} - \Sigma_{xy}\Sigma_{yy}^{-1}\Sigma_{yx} \\ &= \Sigma_{k+1|k} - K^*\Sigma_{yy}{K^*}^\top. \end{aligned} \tag{3.36}$$

Summary of UKF

1. The Unscented Transform (UT) is an alternative to linearization. It gives a better approximation of the mean and covariance of the random variable after being transformed using a nonlinear function than taking the Taylor series approximation.
2. The UKF uses the UT and its sigma points for propagation of uncertainty through the dynamics (3.32) and observation nonlinearities (3.36).

3.7.4 UKF vs. EKF

As compared to the Extended Kalman Filter, the UKF is a better approximation for nonlinear systems. Of course, if the system is linear, both EKF and UKF are equivalent to a Kalman Filter.

In practice, we typically use the UKF with some tuning parameters in

the Unscented Transform as discussed in Section 3.7.2. In practice, the EKF also has tuning parameters where we may wish to perform multiple updates of the dynamics equations with a smaller time-discretization before the next observation comes in to alleviate the effect of linearizing the dynamics. A well-tuned EKF is often only marginally worse than an UKF: the former requires us to compute Jacobians at each step which the latter does not, but the latter is often a more involved implementation.

UKF/EKF approximate filtering distribution as a Gaussian An important point to remember about both the UKF and EKF is that even if they can handle nonlinear systems, they still approximate the filtering distribution

$$\mathrm{P}(x_k \mid y_1, \ldots, y_k)$$

as a Gaussian.

3.8 Particle Filters (PFs)

We next look at particle filters (PFs) which are a generalization of the UKF and can handle non-Gaussian filtering distributions. Just like the UT forms the building block of the UKF, the building block of a particle filter is the idea of importance sampling.

3.8.1 Importance sampling

Consider the following problem, given a probability distribution $p(x)$, we want to approximate it as a sum of Dirac-delta distributions at points $x^{(i)}$, also called "particles", each with weight $w^{(i)}$

$$p(x) \approx \sum_{i=1}^{n} w^{(i)} \delta_{x^{(i)}}(x).$$

Say all weights are equal $1/n$. Depending upon how we pick the samples $x^{(i)}$, we can get very different approximations

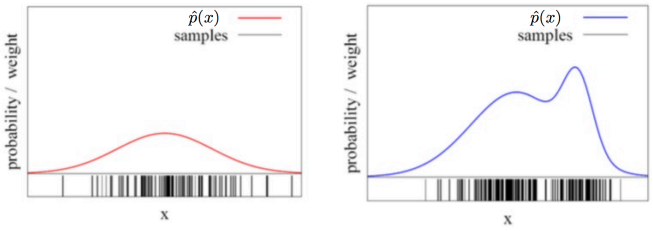
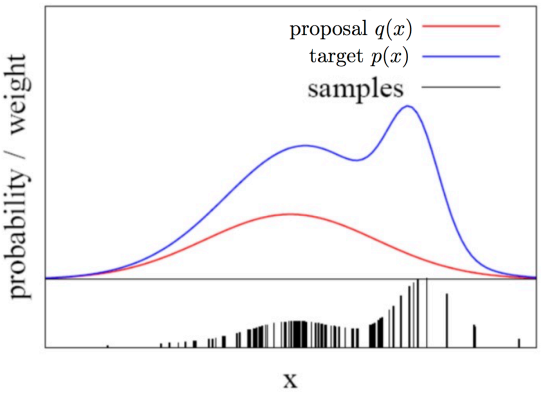


Figure 3.4: Black lines denote particles $x^{(i)}$, while red and blue curves denote the approximations obtained using them. If there are a large number of particles in a given region, the approximated probability density of that region is higher.

We see in Figure 3.4 that depending upon the samples, the approximated probability distributions $\hat{p}(x)$ can be quite different. Importance sampling is a technique to sample the particles to approximate a given probability distribution $p(x)$. The main idea is to use another *known* probability distribution, let us call it $q(x)$ to *generate particles* $x^{(i)}$ and account for the differences between the two by assigning weights to each particle

$$\begin{aligned} &\text{For } i = 1, \ldots, n, \\ &x^{(i)} \sim q \\ &w^{(i)} = \frac{p(x^{(i)})}{q(x^{(i)})}. \end{aligned}$$

The original distribution $p(x)$ is called the "target" and our chosen distribution $q(x)$ is called the "proposal". If the number of particles n is large, we can expect a better approximation of the target density $p(x)$.



1

3.8.2 Resampling particles to make the weights equal

A particle filter modifies the weights of each particle as it goes through the dynamics and observation update steps. This often causes some particles to have very low weights and some others to have very high weights.

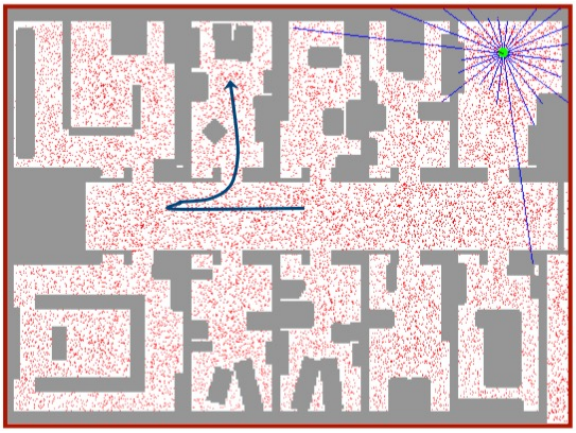


Figure 3.5: An example run of a particle filter. The robot is shown by the green dot in the top right. Observations from a laser sensor (blue rays) attached to the robot measure its distance in a 360-degree field of view around it. Red dots are particles, i.e., possible locations of the robot that we need in order to compute the filtering density $\mathrm{P}(x_k \mid y_1, \ldots, y_k)$. You should think of this picture as being similar to Problem 1 in Homework 1 where the robot was traveling on a grid. Just like the the filtering density in Problem 1 was essentially zero in some parts of the domain, the particles, say in the bottom left, will have essentially zero weights in a particle filter once we incorporate multiple observations from the robot in top right. Instead of having to carry around these null particles with small weights, the resampling step is used to remove them and sample more particles, say in the top right, where we can benefit from a more accurate approximation of the filtering density.

The resampling step takes particles $\left\{w^{(i)}, x^{(i)}\right\}_{i=1}^{n}$ which ap-

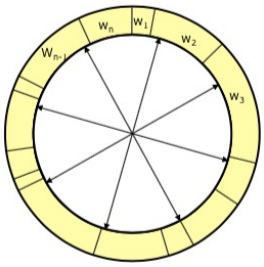
proximate a probability density $p(x)$

$$p(x) = \sum_{i=1}^{n} w^{(i)} \delta_{x^{(i)}}(x)$$

and returns a new set of particles $x'^{(i)}$ with equal weights $w'^{(i)} = 1/n$ that approximate the same probability density

$$p(x) = \frac{1}{n} \sum_{i=1}^{n} \delta_{x'^{(i)}}(x).$$

The goal of the resampling step is to avoid particle degeneracy, i.e., remove unlikely particles with very low weights and effectively split the particles with very large weights into multiple particles.



Consider the weights of particles $\{w^{(i)}\}$ arranged in a roulette wheel as shown above. We perform the following procedure: we start at some location, say $\theta = 0$, and move along the wheel in random increments of the angle. After each random increment, we add the corresponding particle into our set $\{x'^{(i)}\}$. Since particles with higher weights take up a larger angle in the circle, this procedure will often pick those particles and quickly move across particles with small weights without picking them too often. We perform this procedure n times for n particles. As an algorithm

1. Let r be a uniform random variable in interval $[0, 1/n]$. Pick $c = w^{(1)}$ and initialize $i = 1$.

2. For each $m = 1, \ldots, n$, let $u = r + (m-1)/n$. Increment $i \leftarrow i + 1$ and $c \leftarrow c + w^{(i)}$ while $u > c$ and set new particle location $x'^{(m)} = x^{(i)}$.

It is important to notice that the resampling procedure does not actually change the locations of particles. Particles with weights much lower than $1/n$ will be eliminated while particles with weights much higher than $1/n$ will be "cloned" into multiple particles each of weight $1/n$.

There are many other methods of resampling. We have discussed here, something known as "low variance resampling", which is easy to remember and code up. Fancier resampling methods also change the locations of the particles. The goal remains the same, namely to eliminate particles with low weights.

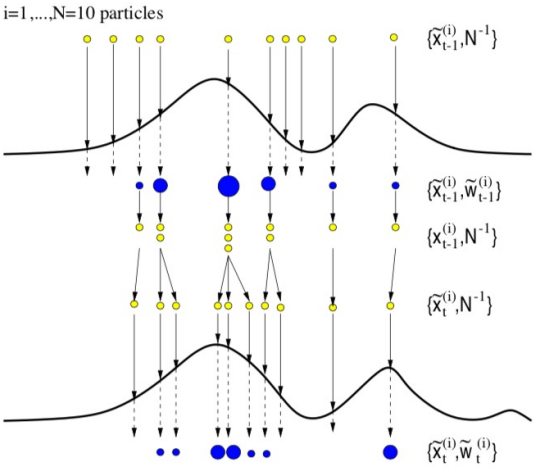


Figure 3.6: A cartoon depicting resampling. Disregard the different notation in this cartoon. Resampling does not change the probability distribution that we wish to approximate; it simply changes the particles and their weights.

3.8.3 Particle filtering: the algorithm

The basic template of a PF is similar to that of the UKF and involves two steps, the first where we propagate particles using the dynamics to estimate $\mathrm{P}(x_{k+1} \mid y_1, \ldots, y_k)$ and a second step where we incorporate the observation to compute the updated distribution $\mathrm{P}(x_{k+1} \mid y_1, \ldots, y_{k+1})$.

Before we look at the theoretical derivation of a particle filter, it will help to go through the algorithm as you would implement on a computer. We assume that we have access to particles $x_{k|k}^{(i)}$

$$\mathrm{P}(x_k \mid y_1, \ldots, y_k) = \frac{1}{n} \sum_{i=1}^{n} \delta_{x_{k|k}^{(i)}}(x),$$

all with equal weights $w_{k|k}^{(i)} = 1/n$.

1. **Step 1: Propagating the dynamics**. Each particle $i =$

$1, \ldots, n$ is updated by one timestep

$$x^{(i)}_{k+1|k} = f(x^{(i)}_{k|k}, u_k) + \epsilon_k$$

where f is the system dynamics using Gaussian noise $\epsilon_k \sim N(0, R)$. Weights of particles are unchanged $w^{(i)}_{k+1|k} = w^{(i)}_{k|k} = 1/n$.

2. **Step 2: Incorporating the observation**. Given a new observation y_{k+1}, we update the weight of each particle using the likelihood of receiving that observation

$$w^{(i)}_{k+1|k+1} \propto \mathrm{P}(y_{k+1} \mid x^{(i)}_{k+1|k})\, w^{(i)}_{k+1|k}.$$

Note that $\mathrm{P}(y_{k+1} \mid x^{(i)}_{k+1|k})$ is a Gaussian and depends upon the Gaussian observation noise ν_k. The mean of this Gaussian is $g(x^{(i)}_{k+1|k})$ and its variance is equal to Q, i.e.,

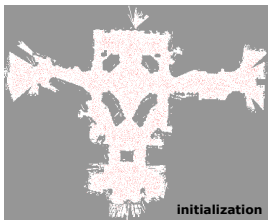
$$\begin{aligned}\mathrm{P}(y_{k+1} \mid x^{(i)}_{k+1|k}) &= \mathrm{P}(\nu_{k+1} \equiv y_{k+1} - g(x^{(i)}_{k+1|k})) \\ &= \frac{1}{\sqrt{(2\pi)^p \det(Q)}} \exp\left(-\frac{\nu^\top_{k+1} Q^{-1} \nu_{k+1}}{2}\right).\end{aligned}$$

Normalize the weights $w^{(i)}_{k+1|k+1}$ to sum up to 1.

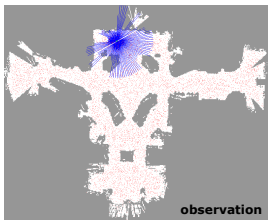
3. **Step 3: Resampling step** Perform the resampling step to obtain new particle locations $x^{(i)}_{k+1|k+1}$ with uniform weights $w^{(i)}_{k+1|k+1} = 1/n$.

1

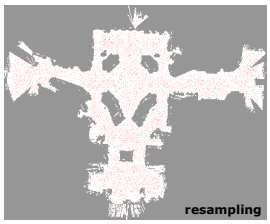
3.8.4 Example: Localization using particle filter



18

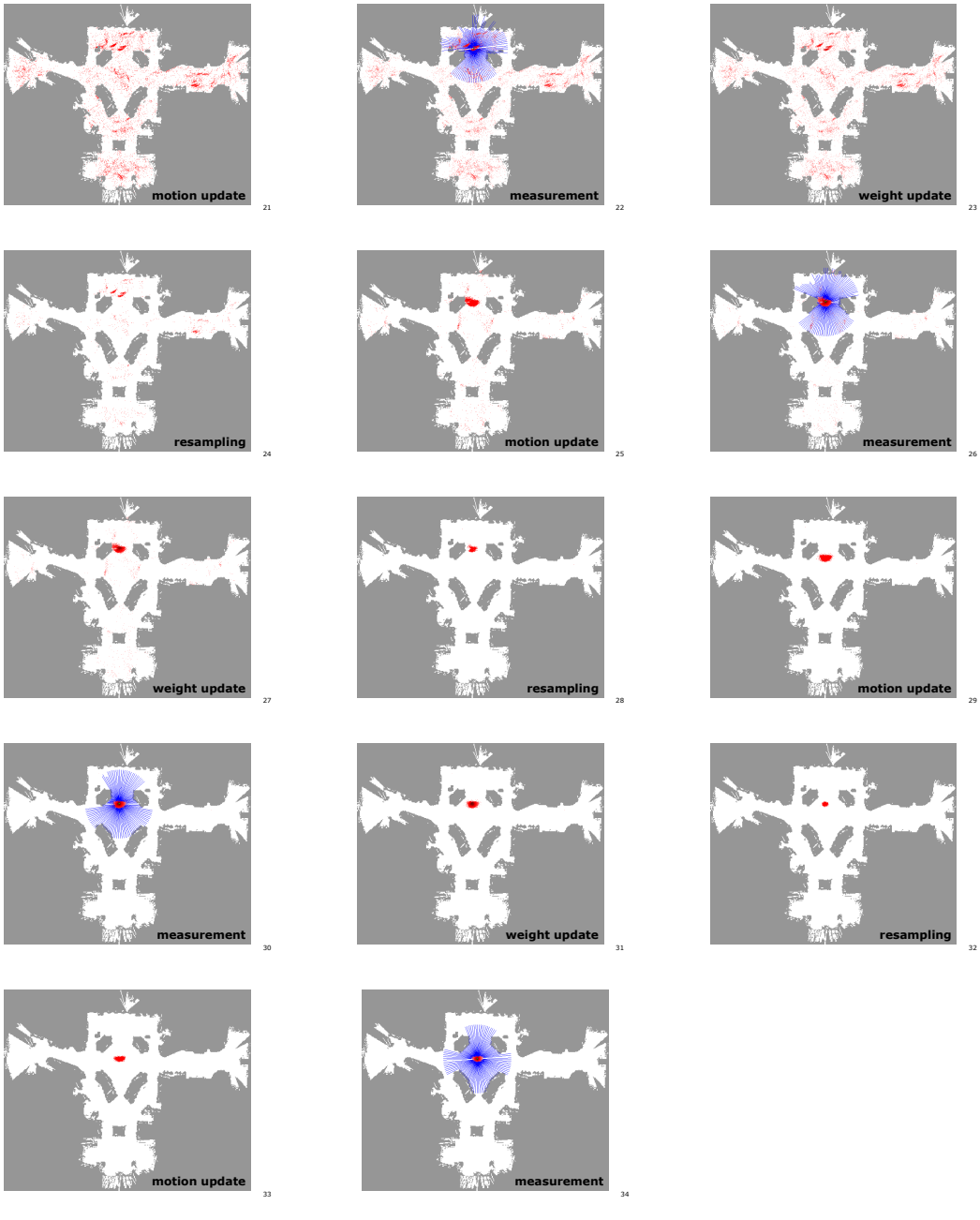


19



20

2



3.8.5 Theoretical insight into particle filtering

Step 1: Propagating the dynamics As we introduced in the section on Markov Decision Processes (MDPs), a stochastic dynamical system

$$x_{k+1} = f(x_k, u_k) + \epsilon_k$$

is equivalent to a probability transition matrix $x_{k+1} \sim \mathrm{P}(x_{k+1} \mid x_k, u_k)$. Our goal is to approximate the distribution of $x_{k+1|k}$ using particles.

What proposal distribution show we choose? The "closest" probability distribution to $x_{k+1|k}$ that we have available is $x_{k|k}$. So we set

$$\begin{aligned}\text{target}: &\mathrm{P}(x_{k+1} \mid y_1, \ldots, y_k)\\ \text{proposal}: &\mathrm{P}(x_k \mid y_1, \ldots, y_k)\end{aligned}$$

In this sense, picking a proposal distribution to draw particles from is like linearization. Better the match between the proposal and the target, fewer samples we need to approximate the target.

Suppose we had performed resampling on our particle set from the distribution $x_{k|k}$ and have a set of n particles $\left\{x_{k|k}^{(i)}\right\}$ with equal weights $1/n$

$$\mathrm{P}(x_k \mid y_1, \ldots, y_k) \approx \frac{1}{n}\sum_{i=1}^{n} \delta_{x_{k|k}^{(i)}}(x).$$

Propagating the dynamics in a PF involves computing importance sampling weights. If we had a particle at location x that was supposed to approximate the distribution of $x_{k+1|k}$, as we saw for importance sampling, its importance weight is the ratio of the target and proposal densities at that location

$$w_{k+1|k}(x) = \frac{\mathrm{P}(x_{k+1} = x \mid y_1, \ldots, y_k)}{\mathrm{P}(x_k = x \mid y_1, \ldots, y_k)}.$$

Let us focus on the numerator. We have

$$\begin{aligned}\mathrm{P}(x_{k+1} = x \mid y_1, \ldots, y_k) &= \int \mathrm{P}(x_{k+1} = x, x_k = x' \mid y_1, \ldots, y_k)\, \mathrm{d}x_k\\ &= \int \mathrm{P}(x_{k+1} = x \mid x_k = x', y_1, \ldots, y_k)\ \mathrm{P}(x_k = x' \mid y_1, \ldots, y_k)\mathrm{d}x_k\\ &= \int \mathrm{P}(x_{k+1} = x \mid x_k = x')\ \mathrm{P}(x_k = x' \mid y_1, \ldots, y_k)\, \mathrm{d}x'\\ &\approx \frac{1}{n}\int \mathrm{P}(x_{k+1} = x \mid x_k = x')\ \sum_{i=1}^{n} \delta_{x_{k|k}^{(i)}}(x')\, \mathrm{d}x'\\ &= \frac{1}{n}\sum_{i=1}^{n} \mathrm{P}(x_{k+1} = x \mid x_k = x_{k|k}^{(i)}, u = u_k),\end{aligned}$$

where the system dynamics is $f(x_k, u_k) + \epsilon_k$ and u_k is the control at time k. The denominator $\mathrm{P}(x_k = x_{k|k}^{(i)} \mid y_1, \ldots, y_k)$ when evaluated at particles $x_{k|k}^{(i)}$ is simply $1/n$. This gives us weights

$$w_{k+1|k}(x) = \sum_{i=1}^{n} \mathrm{P}(x_{k+1} = x \mid x_k = x_{k|k}^{(i)}, u = u_k). \tag{3.37}$$

Let us now think about what particles we should pick for $x_{k+1|k}$. We have from (3.37) a function that lets us compute the correct weight for any particle we may choose to approximate $x_{k+1|k}$.

Say we keep the particle locations unchanged, i.e., $x_{k+1|k}^{(i)} = x_{k|k}^{(i)}$.

We then have

$$\mathrm{P}(x_{k+1} = x \mid y_1, \ldots, y_k) \approx \sum_{i=1}^{n} w_{k+1|k}(x_{k|k}^{(i)})\, \delta_{x_{k|k}^{(i)}}(x). \tag{3.38}$$

Draw a picture of how this approximation looks.

You will notice that keeping the particle locations unchanged may be a very poor approximation. After all, the probability density $\mathrm{P}(x_{k+1} \mid y_1, \ldots, y_k)$ is large, not at the particles $x_{k|k}^{(i)}$ (that were a good approximation of $x_{k|k}$), but rather at the transformed locations of these particles

$$f(x_{k|k}^{(i)}, u_k).$$

We will therefore update the locations of the particles to be

$$x_{k+1|k}^{(i)} = f(x_{k|k}^{(i)}, u_k) \tag{3.39}$$

with weight of the i^{th} particle given by

$$\begin{aligned} w_{k+1|k}^{(i)} := w_{k+1|k}(x_{k+1|k}^{(i)}) &= \sum_{j=1}^{n} \mathrm{P}(x_{k+1} = x_{k+1|k}^{(i)} \mid x_k = x_{k|k}^{(j)}, u = u_k) \\ &\approx \mathrm{P}(x_{k+1} = x_{k+1|k}^{(i)} \mid x_k = x_{k|k}^{(i)}, u = u_k). \end{aligned} \tag{3.40}$$

The approximation in the above equation is very crude: we are assuming that each particle $x_{k|k}^{(i)}$ is transformed independently of the other particles to a new location $x_{k+1|k}^{(i)} = f(x_{k|k}^{(i)}, u_k)$, and no two particles arrive at the same location. This completes the first step of a particle filter and we have

$$\mathrm{P}(x_{k+1} = x \mid y_1, \ldots, y_k) \approx \sum_{i=1}^{n} w_{k+1|k}^{(i)} \delta_{x_{k+1|k}^{(i)}}(x).$$

Step 2: Incorporating the observation The target and proposal distributions in this case are

$$\begin{aligned} \text{target} &: \mathrm{P}(x_{k+1} \mid y_1, \ldots, y_k, y_{k+1}) \\ \text{proposal} &: \mathrm{P}(x_{k+1} \mid y_1, \ldots, y_k). \end{aligned}$$

Since we have particles $x_{k+1|k}^{(i)}$ with weights $w_{k+1|k}^{(i)}$ for the proposal distribution obtained from the propagation step, we now like to update them to incorporate the latest observation y_{k+1}. Let us imagine for a moment that the weights $w_{k+1|k}^{(i)}$ are uniform. We would then set weights

$$\begin{aligned} w(x) &= \frac{\mathrm{P}(x_{k+1} = x \mid y_1, \ldots, y_k, y_{k+1})}{\mathrm{P}(x_{k+1} = x \mid y_1, \ldots, y_k)} \\ &\propto \frac{\mathrm{P}(y_{k+1} \mid x_{k+1} = x)\, \mathrm{P}(x_{k+1} = x \mid y_1, \ldots, y_k)}{\mathrm{P}(x_{k+1} = x \mid y_1, \ldots, y_k)} \quad \text{(by Bayes rule)} \\ &= \mathrm{P}(y_{k+1} \mid x_{k+1} = x). \end{aligned}$$

for each particle $x = x^{(i)}_{k+1|k}$ to get the approximated distribution as

$$\mathrm{P}(x_{k+1} = x \mid y_1, \ldots, y_{k+1}) \approx \sum_{i=1}^{n} \mathrm{P}(y_{k+1} \mid x^{(i)}_{k+1|k})\, w^{(i)}_{k+1|k} \delta_{x^{(i)}_{k+1|k}}(x) \tag{3.41}$$

You will notice that the right hand side is not normalized and the distribution does not integrate to 1 (why? because we did not write the proportionality constant in the Bayes rule above). This is easily fixed by normalizing the coefficients $\mathrm{P}(y_{k+1} \mid x^{(i)}_{k+1|k})\, w^{(i)}_{k+1|k}$ to sum to 1 as follows

$$w^{(i)}_{k+1|k+1} := \frac{\mathrm{P}(y_{k+1} \mid x^{(i)}_{k+1|k})\, w^{(i)}_{k+1|k}}{\sum_j \mathrm{P}(y_{k+1} \mid x^{(j)}_{k+1|k})\, w^{(j)}_{k+1|k}}.$$

Step 3: Resampling step As we discussed in the previous section, after incorporating the observation, some particles may have very small weights. The resampling procedure resamples particles so that all of them have equal weights $1/n$.

$$\left\{x^{(i)}_{k+1|k+1}, 1/n\right\}_{i=1}^{n} = \text{resample}\left(\left\{x^{(i)}_{k+1|k+1}, w^{(i)}_{k+1|k+1}\right\}_{i=1}^{n}\right).$$

3.9 Discussion

This brings our study of filtering to a close. We have looked at some of the most important algorithms for a variety of dynamical systems, both linear and nonlinear. Although, we focused on filtering in this chapter, all these algorithms have their corresponding "smoothing" variants, e.g., you can read about how a typical Kalman smoother is implemented at https://en.wikipedia.org/wiki/Kalman_filter#Fixed-lag_smoother. Filtering, and state estimation, is a very wide area of research even today and you will find variants of these algorithms in almost every device which senses the environment.

Chapter 4

Rigid-body transforms and Mapping

Reading

1. LaValle Chapter 3.2 for rotation matrices, Chapter 4.1-4.2 for quaternions
2. Thrun Chapter 9.1-9.2 for occupancy grids
3. OctoMap: An Efficient Probabilistic 3D Mapping Framework Based on Octrees http://www.arminhornung.de/Research/pub/hornung13auro.pdf, also see https://octomap.github.io.
4. Robot Operating System http://www.willowgarage.com/sites/default/files/icraoss09-ROS.pdf, Optional: Lightweight Communications and Marshalling (LCM) system https://people.csail.mit.edu/albert/pubs/2010-huang-olson-moore-lcm-iros.pdf
5. A Perception-Driven Autonomous Urban Vehicle https://april.eecs.umich.edu/media/pdfs/mitduc2009.pdf
6. Optional reading: Thrun Chapter 10 for simultaneous localization and mapping

In the previous chapter, we looked at ways to estimate the state of the robot in the physical world. We kept our formulation abstract, e.g., the way the robot moves was captured by an abstract expression like $x_{k+1} = f(x_k, u_k) + \epsilon$ and observations $y_k = g(x_k) + \nu$ were similarly opaque. In order to actually implement state estimation algorithms on real robots, we need to put concrete functions in place of f, g.

This is easy to do for some robots, e.g., the robot in Problem 1 in Homework 1 moved across cells. Of course real robots are a bit more complicated, e.g., a car cannot move sideways (which is a huge headache when you parallel park). In the first half of this chapter, we will look at how to model the dynamics f using rigid-body transforms.

The story of measurement models and sensors is similar. Although we need to write explicit formulae in place of the abstract function g. In the second half, we will study occupancy grids and dig deeper into a typical state-estimation problem in robotics, namely that of mapping the location of objects in the world around the robot.

4.1 Rigid-Body Transformations

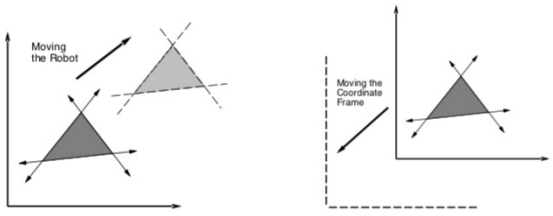
Let us imagine that the robot has a rigid body, we think of this as a subset $A \subset \mathbb{R}^2$. Say the robot is a disc

$$A = \left\{(x, y) \in \mathbb{R}^2 : x^2 + y^2 \leq 1\right\}.$$

This set A changes as the robot moves around, e.g., if the center of mass of the robot is translated by $x_t, y_t \in \mathbb{R}$ the set A changes to

$$A' = \{(x + x_t, y + y_t) : (x, y) \in A\}.$$

The concept of "degrees of freedom" denotes the maximum number of independent parameters needed to completely characterize the transformation applied to a robot. Since the set of allowed values (x_t, y_t) is a two-dimensional subset of $\mathbb{R}^2$, then the degrees of freedom available to a translating robot is two.



(a) Translation of the robot (b) Translation of the frame

As the above figure shows, there are two ways of thinking about this transformation. We can either think of the robot transforming while the co-ordinate frame of the world is fixed, or we can think of it as the robot remaining stationary and the co-ordinate frame undergoing a translation. The second style is useful if you want to imagine things from the robot's perspective. But the first one feels much more natural and we will therefore exclusively use the first notion.

If the same robot if it where rotated counterclockwise by some angle $\theta \in [0, 2\pi]$, we would map

$$(x, y) \mapsto (x \cos\theta - y \sin\theta, x \sin\theta + y \cos\theta).$$

Such a map can be written as multiplication by a 2×2 rotation matrix

$$R(\theta) = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix}. \tag{4.1}$$

to get

$$\begin{bmatrix} x\cos\theta - y\sin\theta \\ x\sin\theta + y\cos\theta \end{bmatrix} = R(\theta) \begin{bmatrix} x \\ y \end{bmatrix}.$$

The transformed robot is thus given by

$$A' = \left\{ R \begin{bmatrix} x \\ y \end{bmatrix} : (x, y) \in A \right\}.$$

If we perform both rotation and translation, we can the transformation using a single matrix

$$T = \begin{bmatrix} \cos\theta & -\sin\theta & x_t \\ \sin\theta & \cos\theta & y_t \\ 0 & 0 & 1 \end{bmatrix} \tag{4.2}$$

and this transformation looks like

$$\begin{bmatrix} x\cos\theta - y\sin\theta + x_t \\ x\sin\theta + y\cos\theta + y_t \\ 1 \end{bmatrix} = T \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}.$$

The point $(x, y, 1) \in \mathbb{R}^3$ is called homogeneous coordinate space corresponding to $(x, y) \in \mathbb{R}^3$ and the matrix T is called a *homogeneous transformation matrix*. The peculiar names comes from the fact that even if the matrix T maps rotations and translations of rigid bodies $A \subset \mathbb{R}^2$, it is just a linear transformation of the point $(x, y, 1)$ if viewed in the larger space $\mathbb{R}^3$.

Rigid-body transformations The transformations $R \in \mathbb{R}^{2\times2}$ or $T \in \mathbb{R}^{3\times3}$ are called rigid-body transformations. Mathematically, it means that they do not cause the distance between any two points inside the set A to change. Rigid-body transformations are what are called an orthogonal group in mathematics.

A group is a mathematical object which imposes certain conditions upon how two operations, e.g., rotations, can be composed together. For instance, if G is the group of rotations, then (i) the composition of two rotations is a rotation, we say that it satisfies closure $R(\theta_1)R(\theta_2) \in G$, (ii) rotations are associative

$$R(\theta_1)\left\{R(\theta_2)R(\theta_3)\right\} = \left\{R(\theta_1)R(\theta_2)\right\}R(\theta_3),$$

and, (iii) there exists an identity and inverse rotation

$$R(0), R(-\theta) \in G.$$

i It is important to remember that T represents rotation *followed* by a translation, not the other way around.

An orthogonal group is a group whose operations preserve distances in Euclidean space, i.e., $g \in G$ is an element of the group that acts on two points $x, y \in \mathbb{R}^d$ then

$$\|g(x) - g(y)\| = \|x - y\|.$$

If we identify the basis in Euclidean space to be the set of orthonormal vectors $\{e_1, \ldots, e_d\}$, then equivalently, the orthogonal group $O(d)$ is the set of orthogonal matrices

$$O(d) := \left\{O \in \mathbb{R}^{d\times d} : OO^\top = O^\top O = I\right\}.$$

This implies that the square of the determinant of any element $a \in O(d)$ is 1, i.e., $\det(a) = \pm 1$.

Check that any rotation matrix R belongs to an orthogonal group.

The Special Orthogonal Group is a sub-group of the orthogonal group where the determinant of each element is +1. You can see that rotations are a special orthogonal group. We denote rotations of objects in $\mathbb{R}^2$ as

$$\mathrm{SO}(2) := \left\{R \in \mathbb{R}^{2\times 2} : R^\top R = RR^\top = I, \det(R) = 1\right\}. \tag{4.3}$$

Each group element $g \in \mathrm{SO}(2)$ denotes a rotation of the XY-plane about the Z-axis. The group of 3D rotations is called the Special Orthogonal Group $\mathrm{SO}(3)$ and is defined similarly

$$\mathrm{SO}(3) := \left\{R \in \mathbb{R}^{3\times 3} : R^\top R = RR^\top = I, \det(R) = 1\right\}. \tag{4.4}$$

The Special Euclidean Group SE(2) is simply a composition of a 2D rotation $R \in \mathrm{SO}(2)$ and a 2D translation $\mathbb{R}^2 \ni v \equiv (x_t, y_t)$

$$\mathrm{SE}(2) = \left\{\begin{bmatrix} R & v \\ 0 & 1 \end{bmatrix} : R \in \mathrm{SO}(2), v \in \mathbb{R}^2\right\} \subset \mathbb{R}^{3\times 3}. \tag{4.5}$$

The Special Euclidean Group $\mathrm{SE}(3)$ is defined similarly as

$$\mathrm{SE}(3) = \left\{\begin{bmatrix} R & v \\ 0 & 1 \end{bmatrix} : R \in \mathrm{SO}(3), v \in \mathbb{R}^3\right\} \subset \mathbb{R}^{4\times 4}; \tag{4.6}$$

again, remember that it is rotation *followed* by a translation.

4.1.1 3D transformations

Translations and rotations in 3D are conceptually similar to the two-dimensional case; however the details appear a bit more difficult because rotations in 3D are more complicated.

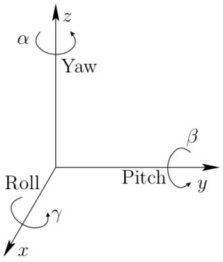


Figure 4.1: Any three-dimensional rotation can be described as a sequence of rotations about each of the cardinal axes. We usually give these specific names: rotation about the Z-axis is called yaw, rotation about the X-axis is called roll and rotation about Y-axis is called pitch. You should commit this picture and these names to memory because it will be of enormous to think about these rotations intuitively.

Euler angles We know that a pure counter-clockwise rotation about one of the axes is written in terms of a matrix, say yaw of α-radians about the Z-axis

$$R_z(\alpha) = \begin{bmatrix} \cos\alpha & -\sin\alpha & 0 \\ \sin\alpha & \cos\alpha & 0 \\ 0 & 0 & 1 \end{bmatrix}.$$

Notice that this is a 3×3 matrix that keeps the Z-coordinate unchanged and only affects the other two coordinates. Similarly we have for pitch (β about the Y-axis) and roll (γ about the X-axis)

$$R_y(\beta) = \begin{bmatrix} \cos\beta & 0 & \sin\beta \\ 0 & 1 & 0 \\ -\sin\beta & 0 & \cos\beta \end{bmatrix}, R_x(\gamma) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\gamma & -\sin\gamma \\ 0 & \sin\gamma & \cos\gamma \end{bmatrix}.$$

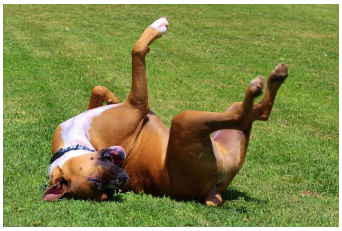
A rotation matrix in three dimensions is a sequential application of rotations, e.g., first yaw, then pitch, and then roll,

$$\mathbb{R}^{3\times3} = R(\gamma,\beta,\alpha) = R_x(\gamma)R_y(\beta)R_z(\alpha). \tag{4.7}$$

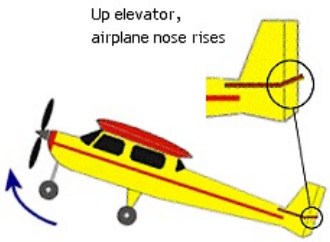
The angles (γ,β,α) (in order: roll, pitch, yaw) are called Euler angles. Imagine how the body frame of the robot changes as successive rotations are applied. If you were sitting in a car, a pure yaw would be similar to the car turning left; the Z-axis corresponding to this yaw would however only be pointing straight up perpendicular to the ground if you had not performed a roll/pitch before. If you had done so, the Z-axis of the body frame with respect to the world will be tiled.

Another important thing to note is that one single parameter determines all possible rotations about one axis, i.e., $\mathrm{SO}(2)$. But three Euler angles are used to parameterize general rotations in three-dimensions. You can watch https://www.youtube.com/watch?v=3Zjf95Jw2UE to get more intuition about Euler angles.

ℹ Here is how I remember these names. Say you are driving a car, usually in robotics we take the X-axis to be longitudinally forward, the Y-axis is your left hand if you are in the driver's seat and the Z-axis points up by the right-hand thumb rule. Roll



is what a dog does when it rolls, it rotates about the X-axis. Pitch is what a plane



does when it takes off, its nose lifts up and it rotates about the Y-axis. Yaw is the one which is not these two.

Rotation matrices to Euler angles We can back-calculate the Euler angles from a rotation matrix as follows. Given an arbitrary matrix

$$R = \begin{bmatrix} r_{11} & r_{12} & r_{13} \\ r_{21} & r_{22} & r_{23} \\ r_{31} & r_{32} & r_{33} \end{bmatrix},$$

we set

$$\begin{aligned} \alpha &= \tan^{-1}(r_{21}/r_{11}) \\ \beta &= \tan^{-1}\left(-r_{31}/\sqrt{r_{32}^2 + r_{33}^2}\right) \\ \gamma &= \tan^{-1}(r_{32}/r_{33}). \end{aligned} \tag{4.8}$$

For each angle, the corresponding quadrant for the Euler angle is determined using the signs of the numerator and the denominator. So you should use the function atan2 in Python/C++ to implement these expressions correctly. Notice that some of the expressions have r_{11} and r_{33} in the denominator, this means that we need $r_{11} = \cos\alpha\cos\beta \neq 0$ and $r_{33} = \cos\beta\cos\gamma \neq 0$. A particular physical rotation can be parameterized in many different ways using Euler angles (depending upon the order in which roll, pitch and yaw are applied), so the map from rotation matrices to Euler angles is not unique.

Homogeneous coordinates in three dimensions Just like the 2D case, we can define a 4×4 matrix that transforms points $(x, y, z) \in \mathbb{R}^3$ to their new locations after a rotation by Euler angles (γ, β, α) and a translation by a vector $v = (x_t, y_t, z_t) \in \mathbb{R}^3$

$$T = \begin{bmatrix} R(\gamma, \beta, \alpha) & v \\ 0 & 1 \end{bmatrix}.$$

4.1.2 Rodrigues' formula: an alternate view of rotations

Consider a point $r(t) \in \mathbb{R}^3$ that is being rotated about an axis denoted by a unit vector $\omega \in \mathbb{R}^3$ with an angular velocity of 1 radian/sec. The instantaneous linear velocity of the head of the vector is

$$\dot{r}(t) = \omega \times r(t) \equiv \hat{\omega} r(t) \tag{4.9}$$

where the $\times$ denotes the cross-product of the two vectors $a, b \in \mathbb{R}^3$

$$a \times b = \begin{bmatrix} a_2 b_3 - a_3 b_2 \\ a_3 b_1 - a_1 b_3 \\ a_1 b_2 - a_2 b_1 \end{bmatrix}$$

In practice, e.g., if we run a Kalman filter to estimate the Euler angles, we need be careful in cases when α , β or $\gamma \approx \pi/2$.

Consider when β crosses $\pi/2$, i.e., it goes from $\pi/2 - \epsilon$ to $\pi/2 + \epsilon$ for some small value of ϵ. In this case, $\alpha = \tan^{-1}(r_{21}/r_{11})$, assuming $r_{21} > 0$, will jump from $\tan^{-1}(\infty) = \pi/2$ to $\tan^{-1}(-\infty) = -\pi/2$—a jump of 180 degrees.

Another classic problem when using Euler angles occurs in what is called "Gimbal lock". This refers to the situation when one of the angles, say $\beta = \pi/2$ (pitch up by 90 degrees). In this case, the SO(3) rotation matrix is

$$R = \begin{bmatrix} 0 & 0 & 1 \\ \sin(\alpha+\gamma) & \cos(\alpha+\gamma) & 0 \\ -\cos(\alpha+\gamma) & \sin(\alpha+\gamma) & 0 \end{bmatrix}.$$

Notice here that changing α (yaw) and γ (roll) have the same effect on the rotation. We cannot distinguish the effect of yaw from roll if the pitch is 90 degrees. And such a "lock" persists until $\beta = \pi/2$. Such a gimbal lock happened on Apollo 11; the mechanism that the engineers had designed to flip the orientation by 180 degrees and escape this degeneracy did not work.

These kind of things make it very cumbersome to work with Euler angles in computer code. They are best used for visualization.

which we can equivalently denote as a matrix vector multiplication $a \times b = \hat{a}b$ where

$$\hat{a} = \begin{bmatrix} 0 & -a_3 & a_2 \\ a_3 & 0 & -a_1 \\ -a_2 & a_1 & 0 \end{bmatrix}. \tag{4.10}$$

is a skew-symmetric matrix. The solution of the differential equation (4.9) at time $t = \theta$ is

$$r(\theta) = \exp(\hat{\omega}\theta)\, r(0)$$

where the matrix exponential of a matrix A is defined as

$$\exp(A) = I + A + \frac{A^2}{2!} + \frac{A^3}{3!} + \ldots = \sum_{k=0}^{\infty} \frac{A^k}{k!}.$$

This is an interesting observation: a rotation about a fixed axis ω by an angle θ can represented by the matrix

$$R = \exp(\hat{\omega}\, \theta). \tag{4.11}$$

You can check that this matrix is indeed a rotation by showing that $R^\top R = I$ and that $\det(R) = +1$. We can expand the matrix exponential and collect odd and even powers of $\hat{\omega}$ to get

$$R = I + \sin\theta\, \hat{\omega} + (1 - \cos\theta)\hat{\omega}^2. \tag{4.12}$$

which is the Rodrigues' formula that relates the angle θ and the axis ω to the rotation matrix. We can also go in the opposite direction, i.e., given a matrix R calculate what angle θ and axis ω it corresponds to using

$$\begin{aligned} \cos\theta &= \frac{\mathrm{tr}(R) - 1}{2} \\ \hat{\omega} &= \frac{R - R^\top}{2\sin\theta}. \end{aligned} \tag{4.13}$$

Note that both the above formulae make sense only for $\theta \neq 0$.

4.1.3 Lie groups, exponential and logarithm maps

Groups such as $\mathrm{SO}(2)$ and $\mathrm{SO}(3)$ are topological spaces (viewed as subsets of $\mathbb{R}^{n^2}$) and operations such as multiplication and inverses are continuous functions on these groups. These groups are also smooth manifolds (a manifold is a generalization of a curved surface). Such groups are called *Lie groups* (after Sophus Lie). Associated to each Lie group is a Lie algebra which is the tangent space of the manifold at identity. The Lie algebra of $\mathrm{SO}(3)$ is denoted by $\mathrm{so}(3)$ and likewise we have $\mathrm{so}(2)$. In a sense, the Lie algebra achieves a "linearization" of the Lie group. It is, locally, a coordinate system for a Lie group, i.e., locally the changes in the transformation can be expressed in terms of the basis of the Lie algebra. The exponential map "unlinearizes" the manifold, i.e., it takes objects in

the Lie algebra to objects in the Lie group

$$\exp : \mathrm{so}(3) \mapsto \mathrm{SO}(3).$$

What we have written in (4.11) is really just this map:

$$\begin{aligned}\mathrm{so}(n) &\ni \hat{\omega}\theta \\ \mathrm{SO}(n) &\ni R = \exp(\hat{\omega}\theta).\end{aligned}$$

We can also find the Lie algebra element that corresponds to an element of the Lie group using the logarithm map:

$$\log : \mathrm{SO}(3) \mapsto \mathrm{so}(3).$$

As an example, for $\mathrm{SO}(3)$, we can calculate

$$\log(R) = \sum_{k=1}^{\infty} \frac{(-1)^{k-1}}{k}(R-I)^k.$$

Many Lie algebra elements can produce the same Lie group element.

If an object whose frame had a rotation matrix R with respect to the origin were rotating with an angular velocity ω in the world frame (remember that angular velocity is a vector whose magnitude is the rate of rotation and direction is axis about which the object is rotation), then the rate of change of R would be given by

$$\dot{R} = \hat{\omega}R. \tag{4.14}$$

We can derive this by observing that

$$\begin{aligned} RR^\top &= I \\ \Rightarrow \dot{R}R^\top + R\dot{R}^\top &= 0 \\ \equiv S + S^\top &= 0 \end{aligned}$$

where we wrote $\dot{R}R^\top = S$ for the skew-symmetric matrix S. This gives

$$\dot{R} = SR.$$

Now consider a point in body frame $r \in \mathbb{R}^3$, this same point in the world frame is Rr and its time-derivative of its coordinates r' in the world frame is therefore

$$\dot{r}' = \dot{R}r = SRr = Sr'$$

If ω denotes the angular velocity of the body in the world frame, then we know from (4.9) that $\dot{r} = \hat{\omega}r$, i.e.,

$$S = \hat{\omega}.$$

This gives a physical interpretation to our skew-symmetric matrix S. If ω' is the angular velocity of the body in the body frame (which is what

ℹ The exponential and logarithm map exist for all Lie groups. In particular, $\mathrm{SE}(3)$ where we can also simplify the infinite series. For $\mathrm{se}(3) \ni \xi \equiv (\omega, \delta)$,

$$\mathrm{SE}(3) \ni \exp\hat{\xi} = \begin{bmatrix} \exp\hat{\omega} & J_l(\omega)\delta \\ 0 & 1 \end{bmatrix}$$

where
$J_l(\omega) = I + \frac{1-\cos(\|\omega\|)}{\|\omega\|^2}\hat{\omega} + \frac{\|\omega\|-\sin(\|\omega\|)}{\|\omega\|^3}\hat{\omega}^2$
is the left Jacobian of $\mathrm{SO}(3)$. The logarithm map that calculates $\mathrm{se}(3) \ni \xi = (\omega, \delta)$ from $(R, t) \in \mathrm{SE}(3)$

$$\begin{aligned}\hat{\omega} &= R \\ \delta &= J_l^{-1}(\omega)t\end{aligned}$$

where
$J_l^{-1}(\omega) = I - \frac{\hat{\omega}}{2} + \left(\frac{1}{\|\omega\|^2} - \frac{1+\cos\|\omega\|}{2\|\omega\|\sin\|\omega\|}\right)\hat{\omega}^2.$

the gyroscope measures), then (4.14) is written as

$$\dot{R} = R\hat{\omega}'; \tag{4.15}$$

we of course have the relation $\omega = R\omega'$ that tells us how the vector ω' transforms from the body frame to the world frame to become ω. If we were to implement a Kalman filter whose state is the rotation matrix R, then this would be the dynamics equation and one would typically have an observation for the velocity ω using a gyroscope.

4.2 Quaternions

We know two ways to think about rotations: we can either think in terms of the three Euler angles (γ, β, α), or we can consider a rotation matrix $R \in \mathbb{R}^{3\times 3}$. We also know ways to go back and forth between these two forms with the caveat that solving for Euler angles using (4.8) may be degenerate in some cases. While rotation matrices are the most general representation of rotations, using them in computer code is cumbersome (it is, after all, a matrix of 9 elements). So while we can build an EKF where the state is a rotation matrix, it would be a bit more expensive to run. We can also implement the same filter using Euler angles but doing so will require special care due to the degeneracies.

Quaternions are a neat way to avoid the problems with both the rotation matrix and Euler angles, they parametrize the space of rotations using 4 numbers. The central idea behind quaternions is Euler's theorem which says that any 3D rotation can be considered as a pure rotation by an angle $\theta \in \mathbb{R}$ about an axis given by the unit vector ω. This is the result that we also exploited in Rodrigues' formula.

Quaternions were invented by British mathematician William Rowan Hamilton while walking along a bridge with his wife. He was quite excited by this discovery and promptly graffitied the expression into the stone of the bridge

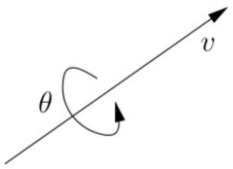


Figure 4.2: Any rotation in 3D can be represented using a unit vector ω and an angle $\theta \in \mathbb{R}$. Notice that there are two ways to encode the same rotation, the unit vector $-\omega$ and angle $2\pi - \theta$ would give the same rotation. Mathematicians say this as quaternions being a double-cover of SO(3).

As you see in the adjoining figure, quaternions also have degeneracies but they are rather easy ones.

A quaternion q as a four-dimensional vector $q \equiv (u_0, u_1, u_2, u_3)$ and we write it as

$$\begin{aligned} q &\equiv (u_0, u), \text{ or} \\ q &= u_0 + u_1 i + u_2 j + u_3 k, \end{aligned} \tag{4.16}$$

with i, j, k being three "imaginary" components of the quaternion with "complex-numbers like" relationships

$$i^2 = j^2 = k^2 = ijk = -1. \tag{4.17}$$

It follows from these relationships that

$$ij = -ji = k, ki = -ik = j, \text{ and } jk = -jk = i.$$

Although you may be tempted to think about this, these imaginary components i, j, k have no relationship with the square roots of negative unity used to define standard complex numbers. You should simply think of the quaternion as a four-dimensional vector. A unit quaternion, i.e., one with

$$u_0^2 + u_1^2 + u_2^2 + u_3^2 = 1,$$

is special: unit quaternions can be used to represent rotations in 3D.

Quaternion to axis-angle representation The quaternion $q = (u_0, u)$ corresponds to to a counterclockwise rotation of angle θ about a unit vector ω where θ and ω are such that

$$u_0 = \cos(\theta/2), \text{ and } u = \omega \sin(\theta/2)\,. \tag{4.18}$$

So given an axis-angle representation of rotation like in Rodrigues' formula (θ, ω) we can write the quaternion as

$$q = (\cos(\theta/2),\ \omega \sin(\theta/2))\,.$$

Using this, we can also compute the inverse of a quaternion (rotation of angle θ about the opposite axis $-\omega$) as

$$q^{-1} := (\cos(\theta/2),\ -\omega \sin(\theta/2))\,.$$

The inverse quaternion is therefore the quaternion where all entries except the first have their signs flipped.

Multiplication of quaternions Just like two rotation matrices multiply together to give a new rotation, quaternions are also a representation for the group of rotations and we can also multiply two quaternions $q_1 = (u_0, u), q_2 = (v_0, v)$ together using the quaternion identities for i, j, k in (4.17) to get a new quaternion

$$q_1 q_2 \equiv (u_0, u) \cdot (v_0, v) = (u_0 v_0 - u^\top v, u_0 v + v_0 u + u \times v).$$

ℹ Quaternions belong to a larger group than rotations called the Symplectic Group $Sp(1)$.

Pure quaternions A pure quaternion is a quaternion with a zero scalar value $u_0 = 0$. This is very useful to simply store a standard 3D vector $u \in \mathbb{R}^3$ as a quaternion $(0, u)$. We can then rotate points easily between

different frames as follows. Given a vector $x \in \mathbb{R}^3$ we can form a quaternion $(0, x)$. It turns out that

$$q \cdot (0, x) \cdot q^* = (0, R(q)x). \tag{4.19}$$

where $q^* = (u_0, -u)$ is the conjugate quaternion of $q = (u_0, u)$; the conjugate is the same as the inverse for unit quaternions. Notice how the right-hand side is the vector $R(q)x$ corresponding to the vector x rotation by a matrix $R(q)$. This is a very useful trick to transform points across coordinate frames instead of multiplying each point $x \in \mathbb{R}^3$ by the corresponding SE(3) matrix element.

Rotational velocity using quaternions We can take the time-derivative of (4.19) to understand how quaternions can be used to describe the rotational velocity. Consider again that $x' \in \mathbb{R}^3$ is a fixed point in body frame. In the world frame, we can represent it by $q \cdot (0, x') \cdot q^*$. Similarly, we also have $x' = q^* \cdot (0, x) \cdot q$. Therefore,

$$\dot{x} = \dot{q} \cdot (0, x') \cdot q^* + q \cdot (0, x') \cdot \dot{q}^*.$$

It is a little bit of derivation using the expression for the product of two quaternions but we can write the analog of (4.14) using quaternions as

$$\dot{q} = \frac{1}{2}(0, \omega) \cdot q \tag{4.20}$$

where ω is the angular velocity of the body in the world frame. Similarly, we can write

$$\dot{q} = \frac{1}{2} q \cdot (0, \omega') \tag{4.21}$$

where ω' is the angular velocity of the body in the body frame (which a gyroscope measures).

Quaternions to rotation matrix The rotation matrix corresponding to a quaternion is

$$\begin{aligned} &R(q) \\ &= (u_0^2 - u^\top u) I_{3\times 3} + 2\frac{u_0 u}{\|u\|} + 2uu^\top \\ &= \begin{bmatrix} 2(u_0^2 + u_1^2) - 1 & 2(u_1u_2 - u_0u_3) & 2(u_1u_3 + u_0u_2) \\ 2(u_1u_2 + u_0u_3) & 2(u_0^2 + u_2^2) - 1 & 2(u_2u_3 - u_0u_1) \\ 2(u_1u_3 - u_0u_2) & 2(u_2u_3 - u_0u_1) & 2(u_0^2 + u_3^2) - 1 \end{bmatrix}. \end{aligned} \tag{4.22}$$

Using this you can show the identity that rotation matrix corresponding to the product of two quaternions is the product of the individual rotation matrices

$$R(q_1 q_2) = R(q_1) R(q_2).$$

Rotation matrix to quaternion We can also go in the reverse direction. Given a rotation matrix R, the quaternion is

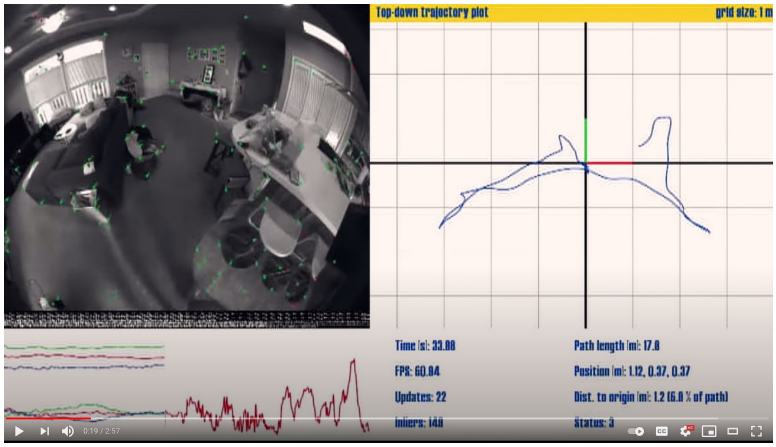
$$
\begin{aligned}
u_0 &= \frac{1}{2}\sqrt{r_{11}+r_{22}+r_{33}+1} \\
\text{if } u_0 \neq 0,\ u_1 &= \frac{r_{32}-r_{23}}{4u_0} \\
u_2 &= \frac{r_{13}-r_{31}}{4u_0} \\
u_3 &= \frac{r_{21}-r_{12}}{4u_0} \\
\text{if } u_0 = 0, u_1 &= \frac{r_{13}r_{12}}{\sqrt{r_{12}^2 r_{13}^2 + r_{12}^2 r_{23}^2 + r_{13}^2 r_{23}^3}} \\
u_2 &= \frac{r_{12}r_{23}}{\sqrt{r_{12}^2 r_{13}^2 + r_{12}^2 r_{23}^2 + r_{13}^2 r_{23}^3}} \\
u_3 &= \frac{r_{13}r_{23}}{\sqrt{r_{12}^2 r_{13}^2 + r_{12}^2 r_{23}^2 + r_{13}^2 r_{23}^3}}.
\end{aligned}
\tag{4.23}
$$

There is little need to memorize these expressions or trying to understand patterns between them. While building a new code base for your robot, you will usually code up these formulae once and all your code will use them again and again.

4.3 Occupancy Grids

Rotation matrices and quaternions let us capture the dynamics of a rigid robot body. We will next look at how to better understand observations.

What is location and what is mapping? Imagine a robot that is moving around in a house. A natural representation of the state of this robot is the 3D location of all the interesting objects in the room, e.g., https://www.youtube.com/watch?v=Qe10ExwzCqk. At each time-instant, we record an observation from our sensor (in this case, a camera) that indicates how far an object is from the robot. This helps us discover the location of the objects in the room. After gathering enough observations, we would have created a *map* of the entire house. This map is the set of positions of all interesting objects in the room. Such a map is called a "feature map", these are all the green points in the image below



The main point to understand about feature map is that we can hand over this map to another robot that comes to the same house. The robot compares images from its camera and if it finds one of the objects inside the map, it can get an estimate of its location/orientation in the room with respect to the known location of the object in the map. The map is just a set of "features" that help identify salient objects in the room (objects which can be easily detected in images and relatively uniquely determine the location inside the room). The second robot using this map to estimate its position/orientation in the room is called the *localization problem*. We already know how to solve the localization problem using filtering.

The first robot was solving a harder problem called *Simultaneous Localization And Mapping (SLAM)*: namely that of discovering the location of both itself and the objects in the house. This is a very important and challenging problem in robots but we will not discuss it further. MEAM 620 digs deeper into it.

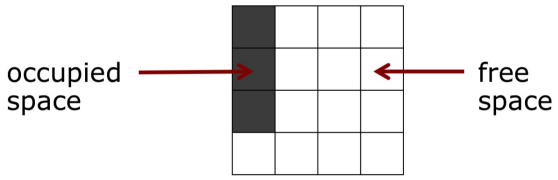
> In this section, we will solve a part of the SLAM problem, namely the mapping problem. We will assume that we know the position/orientation of the robot in the 3D world, and want to build a map of the objects in the world. We will discuss grid maps, which are a more crude way of representing maps than feature maps but can be used easily even if there are lots of objects.

Grid maps We will first discuss two-dimensional grid maps, they look as follows.

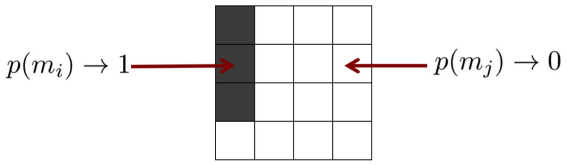
Figure 4.3: A grid map (also called an occupancy grid) is a large gray-scale image, each pixel represents a cell in the physical world. In this picture, cells that are occupied are colored black and empty cells represent free space. A grid map is a useful representation for a robot to localize in this house using observations from its sensors and comparing those to the map.

To get a quick idea of what we want to do, you can watch the mapping being performed in https://www.youtube.com/watch?v=JJhEkIA1xSE. We are interested in learning such maps from the observations that a robot collects as it moves around the physical space. Let us make three simplifying assumptions.

Assumption 1: each cell is either free or occupied



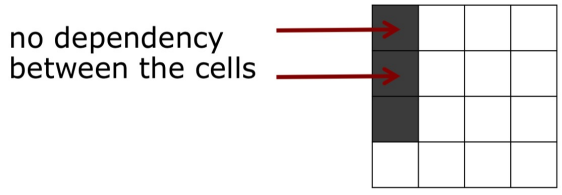
This is neat: we can now model each cell as a binary random variable that indicates occupancy. Let the probability that the cell m_i be occupied be $p(m_i)$



If we have $p(m_i) = 0$, then the cell is not occupied and if we have $p(m_i) = 1$, then the cell is occupied. A priori, we do not know the state of the cell so we will set the prior probability to be $p(m_i) = 0.5$.

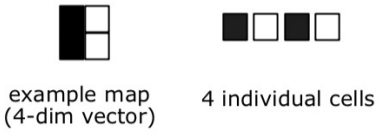
Assumption 2: the world is static Objects in the world do not move. This is reasonable if we are interested in estimating in building a map of the walls inside the room. Note that it is not a reasonable assumption if there are moving people inside the room. We will see a clever hack where the Bayes rule helps automatically disregard such moving objects in this section.

Assumption 3: cells are independent of each other This is another drastic simplification. The state of our system is the occupancy of each cell in the grid map. We assume that before receiving any observations, the occupancy of each individual cell is independent; it is a Bernoulli variable with probability 1/2 since we have assumed the prior to be uniform in Assumption 1.



This means that if cells in the map are denoted by a vector $m = (m_1, \dots,)$, then the probability of the cells being occupied/not-occupied can be written as

$$p(m) = \prod_i p(m_i). \tag{4.24}$$



4.3.1 Estimating the map from the data

Say that the robot pose (position and orientation) is given by the sequence $x_1, \ldots, x_k$. While proceeding along this sequence, the robot receives observations $y_1, \ldots, y_k$. Our goal is to estimate the state of each cell $m_i \in \{0, 1\}$ (aka "the map" $m = (m_1, m_2, \ldots,)$)

$$\mathrm{P}(m \mid x_1, \ldots, x_k, y_1 \ldots, y_k) = \prod_i \mathrm{P}(m_i \mid x_1, \ldots, x_k, y_1 \ldots, y_k). \tag{4.25}$$

This is called the "static state" Bayes filter and is conceptually exactly the same as the recursive application of Bayes rule in Chapter 2 for detecting whether the door was open or closed.

We will use a short form to keep the notation clear

$$y_{1:k} = (y_1, y_2, \ldots, y_k);$$

the quantity $x_{1:k}$ is defined similarly. As usual we will use a recursive Bayes filter to compute this probability as follows.

$$\begin{aligned}
\mathrm{P}(m_i \mid x_{1:k}, y_{1:k}) &\overset{\text{Bayes rule}}{=} \frac{\mathrm{P}(y_k \mid m_i, y_{1:k-1}, x_{1:k})\,\mathrm{P}(m_i \mid y_{1:k-1}, x_{1:k})}{\mathrm{P}(y_k \mid y_{1:k-1}, x_{1:k})} \\
&\overset{\text{Markov}}{=} \frac{\mathrm{P}(y_k \mid m_i, x_k)\,\mathrm{P}(m_i \mid y_{1:k-1}, x_{1:k-1})}{\mathrm{P}(y_k \mid y_{1:k-1}, x_{1:k})} \\
&\overset{\text{Bayes rule}}{=} \frac{\mathrm{P}(m_i \mid y_k, x_k)\,\mathrm{P}(y_k \mid x_k)\,\mathrm{P}(m_i \mid y_{1:k-1}, x_{1:k-1})}{\mathrm{P}(m_i \mid x_k)\,\mathrm{P}(y_k \mid y_{1:k-1}, x_{1:k})} \\
&\overset{\text{Independence}}{=} \frac{\mathrm{P}(m_i \mid y_k, x_k)\,\mathrm{P}(y_k \mid x_k)\,\mathrm{P}(m_i \mid y_{1:k-1}, x_{1:k-1})}{\mathrm{P}(m_i)\,\mathrm{P}(y_k \mid y_{1:k-1}, x_{1:k})}.
\end{aligned} \tag{4.26}$$

We have a similar expression for the opposite probability

$$\mathrm{P}(\neg m_i \mid x_{1:k}, y_{1:k}) = \frac{\mathrm{P}(\neg m_i \mid y_k, x_k)\,\mathrm{P}(y_k \mid x_k)\,\mathrm{P}(\neg m_i \mid y_{1:k-1}, x_{1:k-1})}{\mathrm{P}(\neg m_i)\,\mathrm{P}(y_k \mid y_{1:k-1}, x_{1:k})}.$$

Let us take the ratio of the two to get

$$\begin{aligned}
\frac{\mathrm{P}(m_i \mid x_{1:k}, y_{1:k})}{\mathrm{P}(\neg m_i \mid x_{1:k}, y_{1:k})} &= \frac{\mathrm{P}(m_i \mid y_k, x_k)}{\mathrm{P}(\neg m_i \mid y_k, x_k)} \frac{\mathrm{P}(m_i \mid y_{1:k-1}, x_{1:k-1})}{\mathrm{P}(\neg m_i \mid y_{1:k-1}, x_{1:k-1})} \frac{\mathrm{P}(\neg m_i)}{\mathrm{P}(m_i)} \\
&= \underbrace{\frac{\mathrm{P}(m_i \mid y_k, x_k)}{1 - \mathrm{P}(m_i \mid y_k, x_k)}}_{\text{uses observation } y_k} \underbrace{\frac{\mathrm{P}(m_i \mid y_{1:k-1}, x_{1:k-1})}{1 - \mathrm{P}(m_i \mid y_{1:k-1}, x_{1:k-1})}}_{\text{recursive term}} \underbrace{\frac{1 - \mathrm{P}(m_i)}{\mathrm{P}(m_i)}}_{\text{prior}}.
\end{aligned} \tag{4.27}$$

This is called the odds ratio. Notice that the first term uses the latest observation y_k, the second term can be updated recursively because it is a similar expression as the left-hand side and the third term is a prior

probability of the cell being occupied/non-occupied. Let us rewrite this formula using the log-odds-ratio that makes implementing it particularly easy. The log-odds-ratio of the probability $p(x)$ of a binary variable x is defined as

$$l(x) = \log \frac{p(x)}{1 - p(x)}, \text{ and } p(x) = 1 - \frac{1}{1 + e^{l(x)}}.$$

The product in (4.27) now turns into a sum as

$$l(m_i \mid y_{1:k}, x_{1:k}) = l(m_i \mid y_k, x_k) + l(m_i \mid y_{1:k-1}, x_{1:k-1}) - l(m_i). \tag{4.28}$$

This expression is used to update the occupancy of each cell. The term

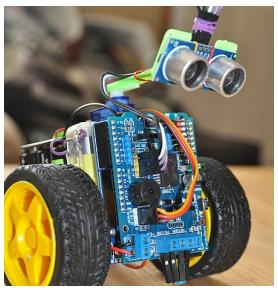
$$\text{sensor model} = l(m_i \mid y_k, x_k)$$

is different for different sensors and we will investigate it next.

❓ We assumed that the map was static. Can you think of why (4.28) automatically lets us handle some moving objects? Think of what the prior odds $l(m_i)$ does to the log-odds-ratio $l(m_i \mid y_{1:k}, x_{1:k})$.

4.3.2 Sensor models

Sonar This works by sending out an ultrasonic chirp and measuring the time between emission and reception of the signal. The time gives an estimate of the distance of an object to the robot.



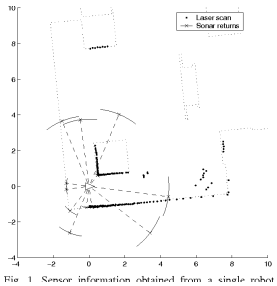
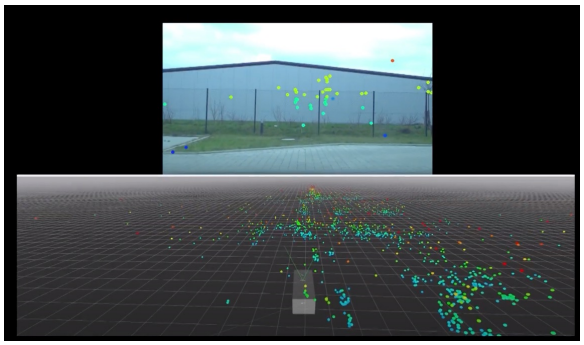


Fig. 1 Sensor information obtained from a single robot

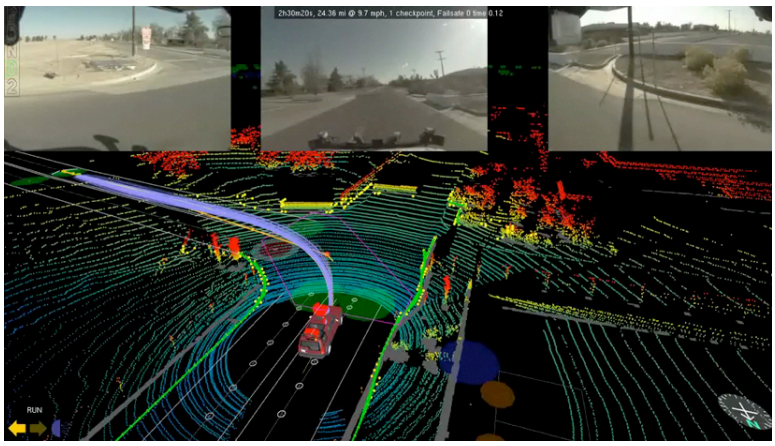
The figure above shows a typical sonar sensor (the two "eyes") on a low-cost robot. Data from the sensor is shown on the right, a sonar is a very low resolution sensor and has a wide field of view, say 15 degrees, i.e., it cannot differentiate between objects that are within 15 degrees of each other and registers them as the same point. Sophisticated sonar technology is used today in marine environments (submarines, fish finders, detecting mines etc.).

Radar works in much the same way as a sonar except that it uses pulses of radio waves and measures the phase difference between the transmitted and the received signal. This is a very versatile sensor (it was invented by the US army to track planes and missiles during World War II) but is typically noisy and requires sophisticated processing to be used for mainstream robotics. Autonomous cars, collision warning systems on human-driven cars, weather sensing, and certainly the military use the radar today. The following picture and the video

https://www.youtube.com/watch?v=hwKUcu_7F9E will give you an appreciation of the kind of data that a radar records. Radar is a very long range sensor (typically 150 m) and works primarily to detect metallic objects.



LiDAR LiDAR, which is short for Light Detection and Ranging, (https://en.wikipedia.org/wiki/Lidar) is a portmanteau of light and radar. It is a sensor that uses a pulsed laser as the source of illumination and records the time it takes (nanoseconds typically) for the signal to return to the source. See https://www.youtube.com/watch?v=NZKvf1cXe8s for how the data from a typical LiDAR (Velodyne) looks like. While a Velodyne contains an intricate system of rotating mirrors and circuitry to measure time elapsed, there are new solid state LiDARs that are rapidly evolving to match the needs of the autonomous driving industry. Most LiDARs have a usable range of about 100 m.



A typical autonomous car This is a picture of MIT's entry named Talos to the DARPA Urban Challenge (https://en.wikipedia.org/wiki/DARPA_Grand_Challenge_(2007)) which was a competition where teams had to traverse a 60 mile urban route within 6 hours, while obeying traffic laws, understanding incoming vehicles etc. Successful demonstrations by multiple teams led by (CMU, Stanford, Odin College, MIT, Penn and Cornell) in this competition jump-started the wave of autonomous driving. While the number of sensors necessary to drive well has come down (Tesla famously does not like

to use LiDARs and rely exclusively on cameras and radars), the type of sensors and challenges associated with them remain essentially the same.

Waymo's autonomous car

4.3.3 Back to sensor modeling

Let us go back to understanding our sensor model $l(m_i \mid y_k, x_k)$ where m_i is a particular cell of the occupancy grid, y_k and x_k are the observations and robot position/orientation at time k.

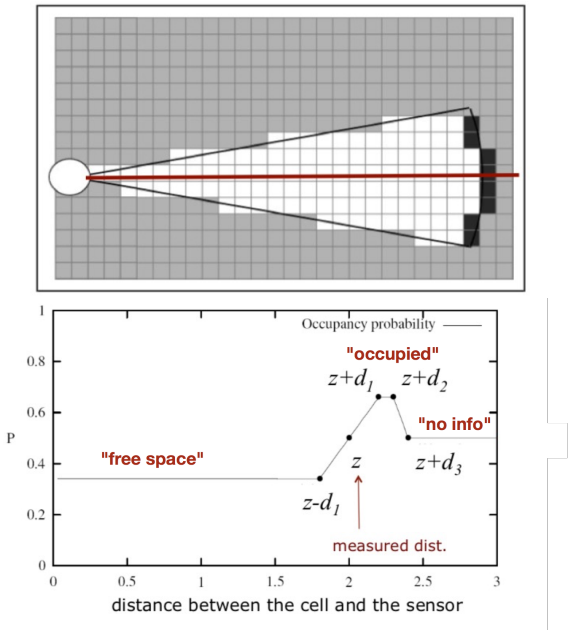
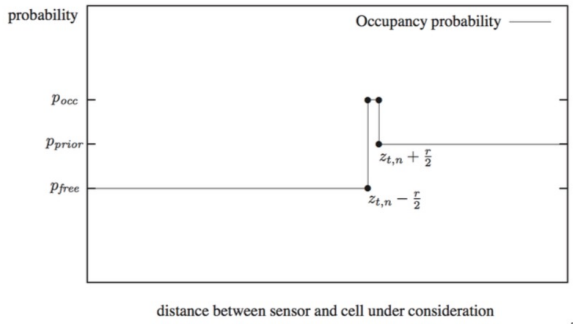


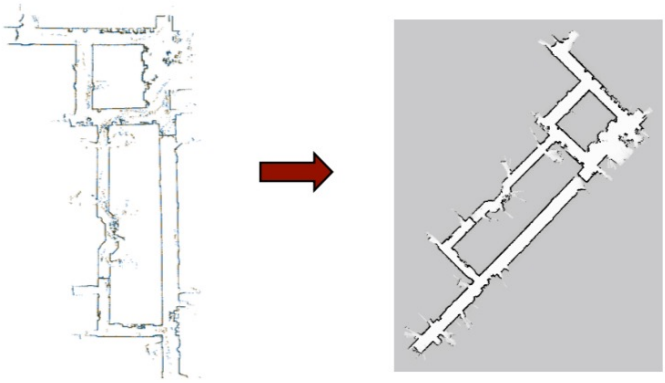
Figure 4.4: **Model for sonar data.** (Top) A sonar gives one real-valued reading corresponding to the distance measured along the red axis. (Bottom) if we travel along the optical axis, the occupancy probability $\mathrm{P}(m_i \mid y_k = z, x_k)$ can be modeled as a spike around the measured value z. It is very important to remember that range sensors such as sonar gives us three kinds of information about this ray: (i) all parts of the environment up to $\approx z$ are *unoccupied* (otherwise we would not have recorded z), (ii) there is some object at z which resulted in the return, (iii) but we do not know anything about what is behind z. So incorporating a measurement y_k from a sonar/radar/lidar involves not just updating the cell which corresponds to the return, but also updating the occupancy probabilities of every grid call along the axis.

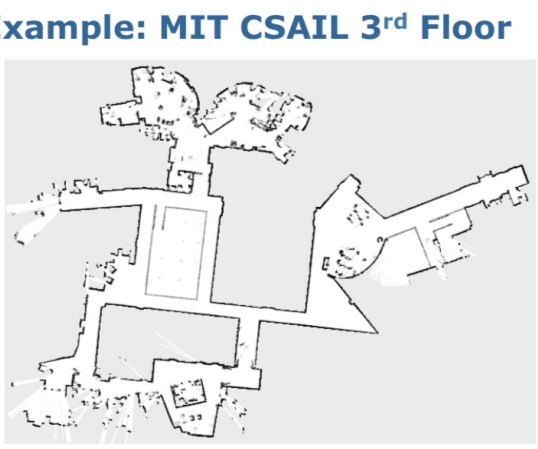
Figure 4.5: (Left) A typical occupancy grid created using a sonar sensor by updating the log-odds-ratio $l(m_i \mid x_{1:k}, y_{1:k})$ for all cells i for multiple time-steps k. At the end of the map building process, if $l(m_i \mid x_{1:k}, y_{1:k}) > 0$ for a particular cell, we set its occupancy to 1 and to zero otherwise, to get the maximum-likelihood estimate of the occupancy grid on the right.

LiDAR model When we say that a LiDAR is a more accurate sensor than the sonar, what we really mean is that the sensor model $\mathrm{P}(m_i \mid y_k, x_k)$ looks as follows.



As a result, we can create high-resolution occupancy grids using a LiDAR.





How will you solve the localization problem given the map? In other words, if we know the occupancy grid of a building as estimated in a prior run, and we now want to find the position/orientation of the robot traveling in this building, how should we use these sensors?

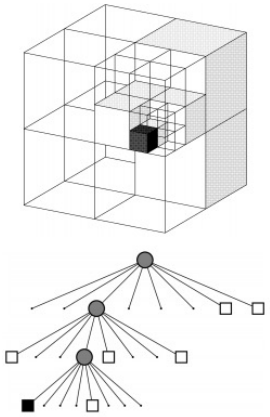
4.4 3D occupancy grids

Two-dimensional occupancy grids are a fine representation for toy problems but they run into some obvious issues. Since the occupancy grid is a "top view" of the world, we cannot represent non-trivial objects in it correctly (a large tree with a thin trunk eats up all the free space). We often desire a fundamentally three-dimensional representation of the physical world.

We could simply create cells in 3D space and our method for occupancy grid would work but this is no longer computationally cheap. For instance, if we want to build a map of Levine Hall (say 100 m $\times$ 100 m area and height of 25 m), a 3D grid map with a resolution of 5 cm $\times$ 5 cm $\times$ cm would have about 2 billion cells (if we store a float in each cell this map will require about 8 GB memory). It would be cumbersome to carry around so many cells and update their probabilities after each sensor reading (a Velodyne gives data at about 30 Hz). More importantly, observe that most of the volume inside Levine is free space (inside of offices, inner courtyard etc.) so we do not really need fine resolution in those regions.

Octrees We would ideally have an occupancy grid whose resolution adapts with the kind of objects that are detected by the sensors. If nearby cells are empty we want to collapse them together to save on memory and computation, on the other hand, if nearby cells are all occupied, we want

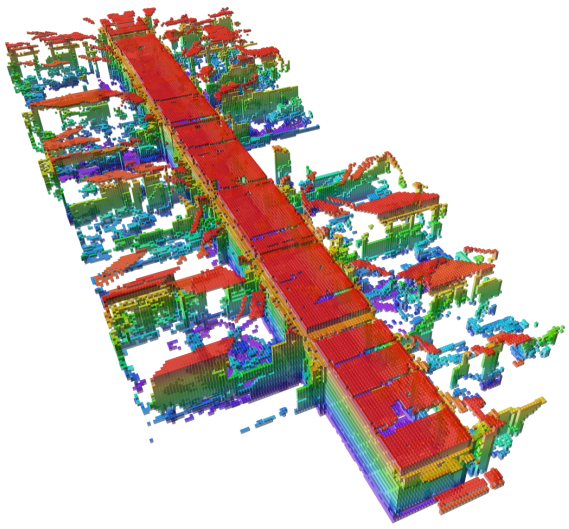
to *refine* the resolution in that area so has to more accurately discern the shape of the underlying objects. Octrees are an efficient representation for 3D volumes.



An octree is a hierarchical data structure that recursively sub-divides the 3D space into octants and allocates volumes as needed for a particular data point observed by a range sensor. It is analogous to a kd-tree. Imagine if the entire space in the above picture were empty (the tree only has a root node), and we receive a reading corresponding to the dark shaded region. An octree would sub-divide the space starting from the root (each node in the tree populates is the parent of its eight child octants) recursively until some pre-determined minimum resolution is reached. This leaf node is grid cell; notice how different cells in the octree have different resolutions. Occupancy probabilities of each leaf node are updated using the same formula as that of (4.28). A key point here is that octrees are designed for accurate sensors (LiDARs) where there is not much noise in the observations returned by the sensor (and thereby we do not refine unnecessary parts of the space)

Octrees are very efficient at storing large map, I expect you can store the entire campus of Penn in about a gigabyte. Ray tracing (following all the cells m_i in tree along the axis of the sensor in Figure 4.4) is harder in this case but there are efficient algorithms devised for this purpose. An example OctoMap (an occupancy map created using an Octree) of a building on the campus of the University of Freiburg is shown below.

ℹ You can find LiDAR maps of the entire United States (taken from a plane) at https://www.usgs.gov/core-science-systems/ngp/3dep



Signed-distance functions A voxel grid or an octree is an explicit representation of the geometry of the scene. There are various alternatives for explicit representations, e.g., meshes that are either generic (made up of triangles for 2D occupancy grids or tetrahedrons for voxel grids) or object specific (e.g., meshes of chairs, tables, people etc. attached to points in the scene).

The size of an explicit representation is large if we want to model the fine-grained structure of an object which makes them ill-suited for regions with sharp corners. Voxel grids/octrees also forget the topology of the scene, e.g., which points belong to which object. Signed distance functions are an implicit representation of the scene, for example a sphere in the Euclidean space $\mathbb{R}^3$ would be stored using a function

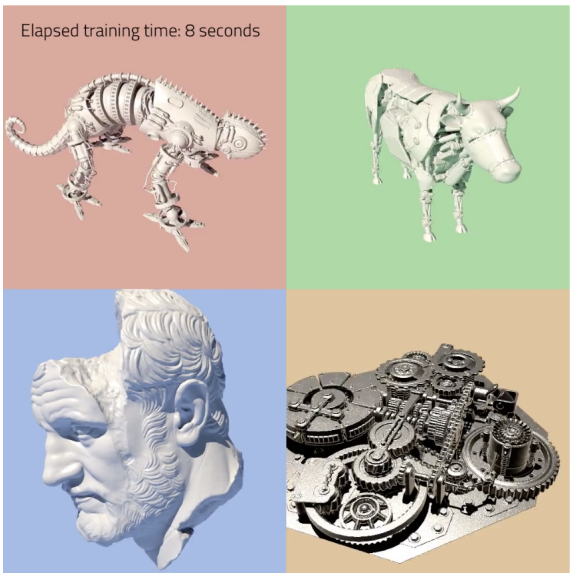
$$\varphi(x) = \|x\|_2^2 - r^2;$$

the sphere is defined implicitly by the level-set: $\{x : \varphi(x) = 0\}$. Points outside the sphere are characterized by $\varphi(x) > 0$ and points inside are characterized by $\varphi(x) < 0$; and this is why φ is known as the signed distance function (SDF). Such a function is also called a "field" in 3D Euclidean space, because it can be queried at any point $x \in \mathbb{R}^3$. It is easy to image how an implicit representation such as this can be converted to an explicit one, we simply discretize the space into a grid and query the signed distance function. Given an explicit representation, we can also convert it

into an implicit representation using powerful algorithms known as fast marching methods. Roughly speaking, they use a dynamic programming procedure, to calculate the closest unoccupied voxel. Usually, SDFs are stored in their analytical form (say polynomials) and this is how many complicated assets are stored in games for rendering later. We can also use a neural network to define the signed distance function φ.

4.5 Neural Radiance Fields (NeRFs)

The maps we have built so far only represent the geometry of the scene, i.e., the shapes and locations of objects. And while it is useful for many tasks, e.g., navigating without collisions, we can perform a richer set of tasks such as object recognition and tracking, semantic scene understanding (object locations and types with respect to other objects in their vicinity) if we retain information about photometric information in the map. To get an idea of the utility of such a representation, just think of how Google Maps 3D would look without all the colors, or the map of the world in your own head. A few impressive examples are the original NeRF paper https://www.matthewtancik.com/nerf, Instant NGP https://nvlabs.github.io/instant-ngp which is a very fast variant of NeRF, or Block NeRF https://waymo.com/research/block-nerf which was used for mapping an entire city. These techniques have become very popular over the last 3-4 years, mostly because they enable a very wide range of tasks, and can be built using visual information from cameras (low power, very large field of view almost 360 degree, dense data without gaps) rather than depth sensors. We should consider them the modern variants of point clouds.



We will assume in this section that we have access to RGB images taken by a calibrated camera (i.e., we know the focal length and the distortion parameters of the lens which together allow us to translate the pixel

coordinates into 3D Euclidean coordinates up to an arbitrary scale) from different, and known, viewpoints in $\mathrm{SE}(3)$. We will be able to represent the map built from this data in different ways, e.g., as RGB images synthesized from some new viewpoint that are consistent with the other images, as a voxel grid and also a density function that models whether a point in space is occupied or not.

Plenoptic function Imagine a person at a fixed point in space viewing along a particular direction $x \in \mathrm{SE}(3)$ (with roll angle fixed to be zero). The plenoptic function essentially describes the set of all things that this person will see:

$$\varphi(x, \lambda) \in \mathbb{R}_+$$

where λ is the wavelength of light. This is the radiance, which is the density of photons at a point traveling in some direction at a given wavelength, for every possible ray. The plenoptic function thus has units of energy per time, per unit area, per unit solid angle, per wavelength; it is also called the "light field". It is an idealized concept used in computer vision and computer graphics to understand the image generation process. Since a ray can be parameterized by 5 numbers (x has 6 dimensions, but we have fixed roll to be zero), the plenoptic function is a five-dimensional function. If we knew the plenoptic function, we can render an image from any viewpoint by simply calculating φ at points on the image plane. For example, the famous Lytro camera (https://en.wikipedia.org/wiki/Lytro, founded by Ren Ng who is also the last author of the original NeRF paper) records the "lightfield", i.e., a large number of pictures from different viewpoints and then render a new view.

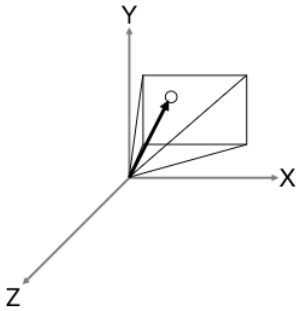
For building a NeRF, we will model the plenoptic function using a neural network. There is a minor difference that while the plenoptic function records the radiance that is incident at a point x, we will model each point in space as emitting radiance and the viewer looking at this point along a ray.

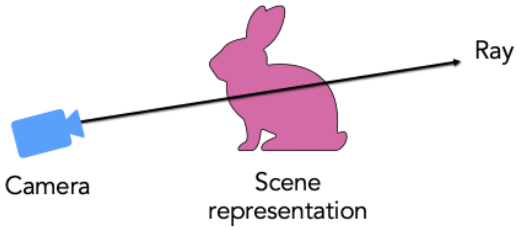
ℹ We can also add a few other variables in the plenoptic function, such as a time (for dynamic scenes), or polarization (the direction of the electric field, which can change upon reflection upon certain objects).

4.5.1 Volumetric rendering

For a perfect pinhole camera, a pixel (i, j) maps to a point $(\frac{i-0.5}{f}, \frac{j-0.5}{f}, -1)$ in the 3D space where f is the focal length. If we are using a lens with distortion, then the so-called intrinsic parameter matrix of the camera can be used to translate homogeneous points $(i, j, 1)$ from the image coordinates to Euclidean coordinates $(x, y, -1)$; the -1 reflects some unknown scale. The intrinsic matrix can be calibrated using a scene with a known geometry (say a chess board with a known distance between vertices).

ℹ The image formation plane of a camera has X axis pointing to the right, Y axis pointing up and the negative Z axis pointing out of the plane. Be careful of conventions and always double check things while debugging.





Let us now consider how the camera might the pixel corresponding to some ray that hits an object. As we discussed above, we will assume that each point on the object emits a radiance (if you find this discomforting, consider this emitted radiance as being reflected radiance from the surface of the rabbit due to some external light source). In computer graphics, one maintains a, say, signed distance function and calculates the point on the rabbit where the ray hits to calculate the pixel intensity. Let us instead perform something called volumetric rendering, where we will integrate the color of each point inside the rabbit that the ray encounters to model the intensity at the corresponding pixel. We will assume that the rabbit is made up of a bunch of tiny colored particles. A point at a distance s along a ray emitted from the viewpoint $\mathrm{SE}(3) \ni x = (R, t)$ and $\omega = \log R$ is given by $x(s) = t + s\omega$. We will now define a few quantities:

- The "probability" that this ray hits a particle in a small interval around s is given by $\sigma(x(s))\,\mathrm{d}s$ where σ represents the volume density.
- The "probability" that this ray reaches $x(s)$, i.e., does not hit a solid object before s, is given by the transmittance $p(s)$.
- The "color" of a point in 3D space is given by $c(x(s)) \in \mathbb{R}^3$ for RGB colors. The color c, as far as our treatment of rendering equation is concerned, is just a placeholder. For example, RGB colors can also be represented as spherical harmonics (which are a basis on the 3D sphere). We can also have c represent the depth, as also semantics captured by higher-dimensional feature vectors (such as those from CLIP) whereby we would be rendering a generalized image.

Let us focus our attention only upon the points along the ray. We can now write an equation of the form

$$\begin{aligned} p(s + \mathrm{d}s) &= p(s)(1 - \sigma(s)\,\mathrm{d}s) \\ \Rightarrow \frac{\mathrm{d}p}{p}\,\mathrm{d}s &= -\sigma(s)\,\mathrm{d}s \\ p(s) &= \exp\left(-\int_0^s \sigma(u)\,\mathrm{d}u\right). \end{aligned} \tag{4.29}$$

This ray hits a particle at distance s with probability $p(s)\sigma(s)\,\mathrm{d}s$. If we wanted to compute the average color at the pixel where the ray originates,

we should do (for some large value of t)

$$\int_0^t \underbrace{p(s)\sigma(s)}_{\text{weights}(s)} c(s)\,\mathrm{d}s \approx \sum_{i=1}^{N} \sigma(s_i)c(s_i) \int_{s_i}^{s_{i+1}} p(s)\,\mathrm{d}s \tag{4.30}$$

Here we have approximated the outer integral using quadrature to make it tractable: $s_i = i\frac{t}{N}$. Effectively we are assuming that the volume density and the color are piecewise constant inside the intervals (s_i, s_{i+1}). The approximation follows from the fact that for $s \in (s_i, s_{i+1})$, if the density is piecewise constant,

$$\begin{aligned} p(s) &= \exp\left(-\int_0^s \sigma(u)\,\mathrm{d}u\right) \\ &= \exp\left(-\int_0^{s_i} \sigma(u)\,\mathrm{d}u\right) \exp\left(-\int_{s_i}^{s} \sigma(u)\,\mathrm{d}u\right) \\ &= p(s_i)\exp\left(-\sigma(s_i)(s-s_i)\right). \end{aligned}$$

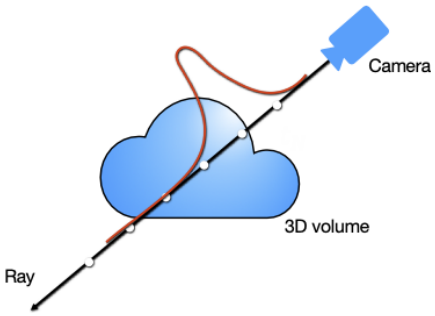
We can now simplify (4.30) as

$$\begin{aligned} \int_0^t \underbrace{p(s)\sigma(s)}_{\text{weights}(s)} c(s)\,\mathrm{d}s &= \sum_{i=1}^{N} \sigma(s_i)c(s_i)p(s_i) \int_{s_i}^{s_{i+1}} e^{-\sigma(s_i)(s-s_i)}\,\mathrm{d}s \\ &= \sum_{i=1}^{N} \sigma(s_i)c(s_i)p(s_i)\frac{e^{-\sigma(s_i)(s_{i+1}-s_i)}-1}{-\sigma(s_i)} \\ &= \sum_{i=1}^{N} c(s_i)p(s_i)\underbrace{\left(1-e^{-\sigma(s_i)(s_{i+1}-s_i)}\right)}_{\text{opacity}} \\ &= \sum_{i=1}^{N} c(s_i)\alpha_i \prod_{j=1}^{i-1}(1-\alpha_j). \end{aligned} \tag{4.31}$$

where the opacity $\alpha_j = 1 - e^{-\sigma(s_j)(s_{j+1}-s_j)}$. This is a very useful equation to remember. The color of a pixel is the weighted average of the color of the points in the 3D space with transmittance and the opacity (i.e., the probability that the ray stops at that point).

> A NeRF models the volume density and the color at a point using a neural network, i.e., a NeRF is a map
>
> $$\mathrm{SE}(3) \ni x \mapsto (\sigma(x), c(x))$$
>
> where the point x is understood as taken with a fixed roll angle, and therefore is a 5-dimensional quantity.

As such, this is an implicit representation of the scene. But given the density function, we can threshold it and easily calculate an explicit representation such as a voxel grid. Such techniques are used to speed



Just like we computed the color of a pixel as the average of the color along a ray, we can also compute its variance (or uncertainty)

$$\int_0^t p(s)\sigma(s)c(s)c(s)^\top\,\mathrm{d}s - \mu\mu^\top$$

where $\mu = \int_0^t p(s)\sigma(s)c(s)\,\mathrm{d}s$. This is very useful in robotics for exploration.
The expected depth along a ray can be calculated using

$$\int_0^t sp(s)\sigma(s)\,\mathrm{d}s\,;$$

with a corresponding expression for the uncertainty on the depth. We can therefore calculate the depth map from a NeRF.

up the quadrature in (4.30) by taking large steps in regions with a small volume density.

Representing viewpoints in a NeRF Consider the neural network (a multi-layer perceptron MLP) in the NeRF that takes in a point $x \in \mathbb{R}^3$ as input and gives the density $\sigma(x) \in \mathbb{R}_+$ and color $c(x) \in \mathbb{R}^3_+$ as the output. This is an implicit representation of the scene, manifest as the MLP. For each viewpoint of the images that we are going to use to fit the NeRF, for each pixel, we have to query this MLP for all the quadrature points along the ray. To give a rough sense of numbers, for 100 RGB images of 4 MP each, if we do quadrature over 10 points, we would need to query the MLP at 4 billion points $x \in \mathbb{R}^3$. This is the equivalent of one epoch in the standard procedure to train a neural network for image classification. After updating the parameters of the MLP using the mean square loss over the rendered images from the fitted plenoptic function and the original images, we will need to query the updated MLP again at all these points for the next epoch.

With a such a large "effective dataset", the MLP better be small if we are to render and train quickly (remember we want to build maps as the robot is moving around...building a map post hoc is less useful). But if we want to model scenes with complex geometry ($\sigma(x)$) and photometry ($c(x)$) then we need to use neural networks with a large capacity. A recurring problem that many researchers noticed while fitting NeRFs is that while low frequency texture was represented accurately in the rendered images, high-frequency texture was often missing (details in the objects):

NeRF (Naive) NeRF (with positional encoding)

The reason for this is that an MLP with a finite width and a certain number of layers cannot represent functions of arbitrarily high bandwidths (which are necessary for high-frequency texture), i.e., $\sigma(x)$ is a very sensitive function of changes in the location x in such regions and the MLP blurs out the structure because it cannot represent the actual structure. This problem is especially exacerbated if we use rudimentary optimization algorithms such as stochastic gradient descent (SGD) to fit the MLP (second-order algorithms like Newton's method or L-BFGS would work much better but they require computing quantities like the Hessian).

A neat workaround this issue is to use a different representation for

the inputs $x \in \mathbb{R}^3$. Instead of using x we use

$$\varphi(x) = (\sin(2^k x_1), \sin(2^k x_2), \sin(2^k x_3), \dots)_{k=0,\cdots,10}$$

where k is the frequency and we choose, say 10 different frequencies. The input layer of the MLP would therefore be 50- instead of 5-dimensional. This forces the MLP to work in the Fourier basis where there errors from different frequencies play an equal role in creating the mean-square error discrepancy between the rendered image and the actual images, this helps fit higher frequency structures better.

4.6 SLAM with NeRFs

NeRFs, which model the photometry and geometry of the scene, are a very good representation for developing a SLAM system. The key pieces of the problem are as follows.

- In SLAM, we would not know the viewpoints from which images were recorded. So these viewpoints, let us call them $x_{1:k} = (x_i \in \mathrm{SE}(3))_{i=1}^k$ are unknown in the problem.
- Suppose we are given RGB images recorded from each viewpoint, $y_{1:k} = (y(x_i))_{i=1}^k$. Let y_k^{ij} denote the intensity of pixel (i, j) for image y_k.
- We will assume that we have calibrated the camera beforehand, i.e., given a point $[i, j, 1]$ in homogeneous coordinates on the image from $x = (\omega, \delta)$, we can transform it into world coordinates as

$$x^{ij} \equiv \begin{bmatrix} \exp \hat{\omega} & \delta \\ 0 & 1 \end{bmatrix} K^{-1} \begin{bmatrix} i \\ j \\ -1 \end{bmatrix}.$$

  A point on a ray from the viewpoint to this pixel at a distance s is given by

$$s x^{ij}.$$

- NeRF-based SLAM techniques build a NeRF using the past images. The NeRF is a representation of the scene. Let us denote it by a function

$$\xi : x \mapsto c, \sigma$$

  that maps a point $x \in \mathrm{SE}(3)$ to the color and volume density at that point. We query the NeRF at a point $s x^{ij}$ to calculate the volume rendering equation and integrate the color along this ray to calculate the expected intensity using (4.31)

$$\hat{y}_k^{ij} \equiv \sum_{i=1}^{N} c(s_i) \alpha_i \prod_{j=1}^{i-1} (1 - \alpha_j) \quad \text{along ray } x_k^{ij}.$$

We should now ensure that the scene and viewpoints are consistent with each other. The mean-squared error between the pixel intensities of

the rendered images and the actual images is

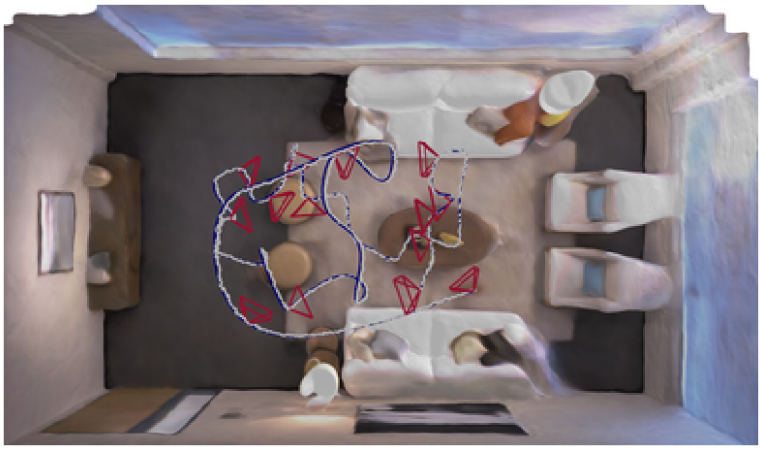
$$\ell(\xi, x_{1:k}) = \frac{1}{k} \sum_k \frac{1}{2N^{ij}} \sum_{i,j} \lambda_k^{ij} \left(\hat{y}_k^{ij} - y_k^{ij} \right)^2 - \log \lambda_k^{ij} \tag{4.32}$$

where there are N^{ij} are the total number of pixels in each image. It is useful to use a scaling factor λ_k^{ij} which is, say, inversely proportional to the variance of the color, this way the MSE loss above down-weights pixels where there is a high degree of uncertainty in the NeRF. If we have depth measurements in addition the RGB, we can set up a similar objective to compare the depth computed from the scene and that returned from a depth camera.

(4.32) is a function of both the scene ξ and the viewpoints $x_{1:k}$. In a standard NeRF, the latter are known so we only optimize the NeRF; this can be done using any standard optimization algorithm, e.g., SGD for fitting the neural network. In NeRF-based SLAM, we should optimize both these parameters

$$\hat{\xi}, \hat{x}_{1:k} = \text{argmin}\, \ell(\xi, x_{1:k}). \tag{4.33}$$

As the robot collects more images, we add these images to the dataset and continue fitting the objective. The image below from a paper titled "iMAP: Implicit Mapping and Positioning in Real-Time" https://arxiv.org/abs/2103.12352 shows the camera poses and the rendered NeRF map.



While the principle behind SLAM may seem straightforward, there are some details that one must be careful about. The objective in (4.32) is difficult to evaluate. As an example, suppose a robot takes 30 FPS video from its camera that gives 4 MP images. The number of terms in the objective is 120 million after one *second* of operation. To calculate the gradient of the objective, the neural network in the NeRF must calculate backprop updates 120 million times and even if each one takes 0.1 ms (the neural networks in NeRFs are tiny and there are a lot of libraries to calculate them even more efficiently than PyTorch, e.g., https://github.com/NVlabs/tiny-cuda-nn), this adds up to 20 minutes. As

ℹ Given a NeRF-based scene, it is easy to localize the robot by optimizing (4.32) only over the unknown pose x_k; this would be the MLE estimate of the pose given the map. We can also build a filter for updating the pose using the latest observation by modeling the observation returned from the NeRF for pixel (i, j) as $y^{ij} = g^{ij}(x) + (\lambda^{ij})^{-1/2}\nu$ where g^{ij} is the volume rendering equation, λ^{ij} is the inverse uncertainty and ν is standard Gaussian noise. The expression in (4.32) can therefore be interpreted as the negative log-likelihood of $\hat{y}^{ij} \sim N(y^{ij}, \left(\lambda^{ij}\right)^2)$; this is why there is a $-\log \lambda_k^{ij}$ term in the objective.

ℹ We need to solve optimization problems like (4.32) in robotics much more precisely than optimization problems while fitting deep networks. To wit, a small error in the estimated pose can lead to a collision. Algorithms like stochastic gradient descent (SGD) or Adam can give rough answers quickly, but for robotics it is often beneficial to use more sophisticated techniques like second-order optimization algorithms, e.g., L-BFGS. The poses $x_{1:k}$ are not arbitrary vectors but instead lie in $\text{SE}(3)$. We should therefore use solvers that are specific to this structure; older algorithms for bundle adjustment used to do this but they are quite cumbersome to use and expensive to run for the large amounts of data that we have from these images; these days it is popular to simply use gradient descent.

the size of the dataset grows, this issue only gets worse.

To work around this issue, typical NeRF-SLAM techniques try to estimate whether a new image sees a significantly different region of the scene (say, by calculating the uncertainty of the color or the depth, or some other heuristic). We take more gradient updates on the latest images, and fewer ones on the past images in (4.32). Instead of using all the rays in each image, we can maintain some statistics about the MSE at each pixel and calculate rays only for pixels separated by sufficient distances in the image plane.

4.7 Local Map

In this chapter, we primarily discussed maps of static environments as the robot moves around in the environment. The purpose of doing so is localization, namely, finding the pose of the robot by comparing the observations of the sensors with the map (think of the particle filter localization example in Chapter 3). In typical problems, we often maintain two kinds of maps, (i) a large occupancy grid for localization (say as big as city), and (ii) another smaller map, called the local map, that is used to maintain the locations of objects (typically objects that can move) in the vicinity of the robot, say a 100 m $\times$ 100 m area.

The local map is used for planning and control purposes, e.g., to check if the planned trajectory of the robot does not collide with any known obstacles. See an example of the local map at the 1:42 min mark at https://www.youtube.com/watch?v=2va15BE-7lQ. Some people also call the local map a "cost map" because occupied cells in the local map indicate a high collision cost of moving through that cell. The local map is typically constructed in the body frame and evolves as the robot moves around (objects appear in the front of the robot and are spawned in the local map and disappear from the map at the back as the robot moves forward).

> You should think of the map (and especially the local map) as the filtering estimate of the locations of various objects in the vicinity of the robot computed on the basis of multiple observations received from the robot's sensors.

4.8 Discussion

Occupancy grids are a very popular approach to represent the environment given the poses of the robot as it travels in this environment. We can also use occupancy grids to localize the robot in a future run (which is usually the purpose of creating them). Each cell in an occupancy grid stores the posterior probability of the cell being occupied on the basis of multiple observations $\{y_1, \ldots, y_k\}$ from respective poses $\{x_1, \ldots, x_k\}$. This is a very efficient representation of the 3D world around us with the one caveat that each cell is updated independently of the others. But since

Figure 4.6: The output of perception modules for a typical autonomous vehicle (taken from https://www.youtube.com/watch?v=tiwVMrTLUWg). The global occupancy grid is shown in gray (see the sides of the road). The local map is not shown in this picture but you can imagine that it has occupied voxels at all places where there are vehicles (purple boxes) and other stationary objects such as traffic light, nearby buildings etc. Typically, if we know that so and so voxel corresponds to a vehicle, we run an Extended Kalman Filter for that particular vehicle to estimate the voxels in the local map that it is likely to be in, in the next time-instant. The local map is a highly dynamic data structure that is rich in information necessary for planning trajectories of the robot.

one gets a large amount of data from typical range senors (a 64 beam Velodyne (https://velodynelidar.com/products/hdl-64e) returns about a 2 million points/sec and cheaper versions of this sensor will cost about $100), this caveat does not hurt us much in practice. You can watch this talk (https://www.youtube.com/watch?v=V8JMwE_L5s0) by the head of Uber's autonomous driving group to get more perspective about localization and mapping.

Chapter 5

Dynamic Programming

Reading

1. (Thrun) Chapter 15
2. (Sutton & Barto) Chapters 3–4
3. Optional: (Bertsekas) Chapter 1 and 4

This is the beginning of Module 2, this module is about "how to act". The first module was about "how to sense". The prototypical problem in the first module was how to assimilate the information gathered by all the sensors into some representation of the world. In the next few lectures, we will assume that this representation is good, that it is accurate in terms of its geometry (small variance of the occupancy grid) and in terms of its information (small innovation in the Kalman filter etc.). Let us also assume that it has all the necessary semantics, e.g., objects are labeled as cars, buses, pedestrians etc (we will talk about how to do this in Module 4).

The prototypical problem investigated in the next few chapters is how to move around in this world, or affect the state of this world to achieve a desired outcome, e.g., drive a car from some place A to another place B.

Our philosophy about notation Material on Dynamic Programming and Reinforcement Learning (RL), which we will cover in the following chapters, contains a lot of tiny details (much more than other areas in robotics/machine learning). These details are usually glossed over in most treatments. In the interest of simplicity, other courses or most research papers these days, develop an imprecise notation and terminology to focus on the problem. However, these details of RL matter enormously when you try to apply these techniques to real-world problems. Not knowing all the details or using imprecise terminology to think about RL is unlikely to make us good at real-world applications.

For this reason, the notation and the treatment in this chapter, and the following ones, will be a bit pedantic. We will see complicated notation and terminology for quantities, e.g., the value function, that you might see being written very succinctly in other places. We will mostly follow the notation of Dmitri Bertsekas' book on "Reinforcement Learning and Optimal Control" (http://www.mit.edu/ dimitrib/RLbook.html). You will get used to the extra notation and it will become second nature once you become more familiar with the concepts.

5.1 Formulating the optimal control problem

Let us denote the state of a robot (and the world) by $x_k \in X \subset \mathbb{R}^n$ at the k^{th} timestep. We can change this state using a control input $u_k \in U \subset \mathbb{R}^p$ and this change is written as

$$x_{k+1} = f_k(x_k, u_k) \tag{5.1}$$

for $k = 0, 1, \ldots, T-1$ starting from some initial given state x_0. This is deterministic nonlinear dynamical system (no noise ϵ in the dynamics). We will let the dynamics f_k also be a function of time k. The time T is some time-horizon up to which we care about running the system. The state-space is X (which we will assume does not change with time k) and the control-space is U.

Recall, that we can safely assume that the system is Markov. The reason for it is as follows. If it is not, and say if x_{k+1} depends upon both x_k and the previous step x_{k-1}, then we can expand the state-space to write a new dynamics in the expanded state-space. We will follow a similar program as that of Module 1: we first describe very general algorithms (dynamic programming) for general systems (Markov Decision Processes), then specialize our methods to a restricted class of systems that are useful in practice (linear dynamical systems) and then finally discuss a very general class of systems again with more sophisticated algorithms (motion-planning).

The central question in this chapter is how to pick a control u_k. We want to pick controls that lead to desirable trajectories of the system, e.g., results in a parallel-parked car at time T and does not collide against any other object for all times $k \in \{1, 2, \ldots, T\}$. We may also want to minimize some chosen quantity, e.g., when you walk to School, you find a trajectory that avoids a certain street with a steep uphill.

Finite, discrete state and control-space In this chapter we will

only consider problems with finitely-many states and controls, we will assume that the state-space X and the control-space U are finite, discrete sets.

Run-time cost and terminal cost We will take a very general view of the above problem and formalize it as follows. Consider a cost function

$$q_k(x_k, u_k) \in \mathbb{R}$$

which gives a scalar real-valued output for every pair (x_k, u_k). This models the fact that you do not want to walk more than you need to get to School, i.e., we would like to minimize q_k. You also want to make sure the trajectory actually reaches the lecture venue, we write this down as another cost $q_f(x_T)$. We want to pick control inputs $(u_0, u_1, \ldots, u_{T-1})$ such that

$$J(x_0; u_0, u_1, \ldots, u_{T-1}) = q_f(x_T) + \sum_{k=0}^{T-1} q_k(x_k, u_k) \qquad (5.2)$$

is minimized. The cost $q_f(x_T)$ is called the terminal cost, it is high if x_T is not the lecture room and small otherwise. The cost q_k is called the run-time cost, it is high for instance if you have to use large control inputs, e.g., x_k is a climb.

The optimal control problem Given a system $x_{k+1} = f_k(x_k, u_k)$, we want to find control *sequences* that minimize the total cost J above, i.e., we want to solve

$$J^*(x_0) = \min_{u_k \in U, k=0,\ldots,T-1} J(x_0; u_0, \ldots, u_{T-1}) \qquad (5.3)$$

It is important to realize that the function $J(x_0; u_0, \ldots, u_{T-1})$ depends upon an entire sequence of control inputs and we need to find them all to find the optimal cost $J^*(x_0)$ of, say reaching the School from your home x_0.

5.2 Dijkstra's algorithm

If the state-space X and control-space U are discrete and finite sets, we can solve (5.3) as a shortest path problem using very fast algorithms. Consider the following picture. This is what would be called a transition graph for a deterministic finite-state dynamics.

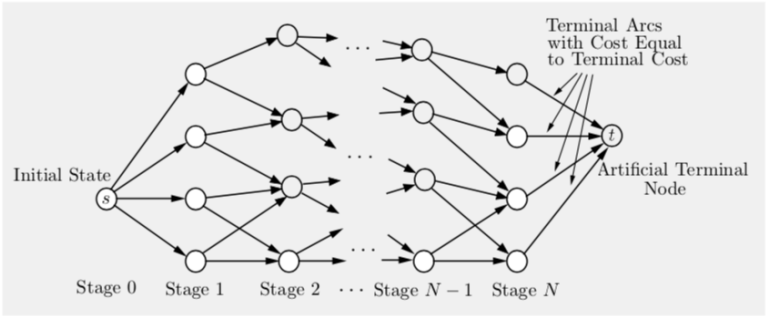


Figure 5.1: Transition graph for Dijkstra's algorithm

The graph has one source node x_0. Each node in the graph is x_k, each edge depicts taking a certain control u_k. Depending on which control we pick, we move to some other node x_{k+1} given by the dynamics $f(x_k, u_k)$. Note that this is *not* a transition like that of a Markov chain, everything is deterministic in this graph. On each edge we write down the cost

$$\text{cost}(x_k, x_{k+1}) := q_k(x_k, u_k)$$

where $x_{k+1} = f_k(x_k, u_k)$ and "close" the graph with a dummy terminal node with the cost $q_f(x_T)$ on every edge leading to an artificial terminal node (sink).

Minimizing the cost in (5.3) is now the same as finding the shortest path in this graph from the source to the sink. The algorithm to do so is quite simple and is called Dijkstra's algorithm after Edsgar Dijkstra who used it around 1956 as a test program for a new computer named ARMAC (http://www-set.win.tue.nl/UnsungHeroes/machines/armac.html).

1. Let Q be the set of nodes that are currently unvisited; all nodes in the graph are added to it at the beginning. S is an empty set. An array called dist maintains the distance of every node in the graph from the source node x_0. Initialize $\text{dist}(x_0) = 0$ and $\text{dist} = \infty$ for all other nodes.
2. At each step, if Q is not empty, pop a node $v \in Q$ such that $v \notin S$ with the smallest $\text{dist}(v)$. Add v to S. Update the dist of all nodes u connected to v. For each u, if

$$\text{dist}(u) > \text{dist}(v) + \text{cost}(u, v)$$

update the distance of u to be $\text{dist}(v) + \text{cost}(u, v)$. If the above condition is not true do nothing.

The algorithm terminates when the set Q is empty.

You might know that there are many other variants of Dijkstra's algorithm, e.g., the A^* algorithm that are quicker to find shortest paths. We will look at some of these in the next chapter.

❓ Shortest path algorithms do not work if there are cycles in the graph because the shortest path is not unique. Are there cycles in the above graph?

❓ What should one do if the state/control space is not finite? Can we still use Dijkstra's algorithm?

The quantity dist is quite special: observe that after Dijkstra's algorithm finishes running and the set Q is empty, the dist function gives the optimal cost to go from each node in the graph to the source node. We wanted to only find the cost to go from source x_0 to the sink node but ended up computing the cost to every node in the graph from the source.

5.2.1 Dijkstra's algorithm in the backwards direction

We can run Dijkstra's algorithm in the backwards direction to get the same answer as well. The sets Q and S are initialized as before. In this case we will let $\text{dist}(v)$ denote the distance of a node v to the sink node. The algorithm proceeds in the same fashion, it pops a node $v \in Q, v \notin S$ and updates the dist of all nodes u connected to v. For each u, if

$$\text{dist}(u) > \text{dist}(v) + \text{cost}(u, v)$$

then we update $\text{dist}(u)$ to be the right-hand side of this inequality. Running Dijkstra's algorithm in reverse (from sink to the source) is completely equivalent to running it in the forward direction (from source to the sink).

ℹ If Dijkstra's algorithm (forwards or backwards) is run on a graph with n vertices and m edges, its computational complexity is $\mathcal{O}(m + n \log n)$ if we use a priority queue to find the node $v \in Q, v \notin S$ with the smallest dist. The number of edges in the transition graph in Figure 5.1 is $m = \mathcal{O}(T|X|)$.

5.3 Principle of Dynamic Programming

The principle of dynamic programming is a formalization of the idea behind Dijkstra's algorithm. It was discovered by Richard Bellman in the 1940s. The idea behind dynamic programming is quite intuitive: it says that the remainder of an optimal trajectory is optimal.

We can prove this as follows. Suppose that we find the optimal control sequence $(u_0^*, u_1^*, \ldots, u_{T-1}^*)$ for the problem in (5.3). Our system is deterministic, so this control sequence results in a *unique* sequence of states $(x_0, x_1^*, \ldots, x_T^*)$. Each successive state is given by $x_{k+1}^* = f_k(x_k^*, u_k^*)$ with $x_0^* = x_0$. The principle of optimality, or the principle of dynamic programming, states that if one starts from a state x_k^* at time k and wishes to minimize the "cost-to-go"

$$q_f(x_T) + q_k(x_k^*, u_k) + \sum_{i=k+1}^{T-1} q_i(x_i, u_i)$$

over the (now assumed unknown) sequence of controls $(u_k, u_{k+1}, \ldots, u_{T-1})$, then the optimal control sequence for this truncated problem is exactly $(u_k^*, \ldots, u_{T-1}^*)$.

The proof of the above assertion is an easy case of proof by contradiction: if the truncated sequence were not optimal starting from x_k^* there

exists some other optimal sequence of controls for the truncated problem, say $(v_k^*, \ldots, v_{T-1}^*)$. If so, the solution of the original problem where one takes controls v_k^* from this new sequence for time-steps $k, k+1, \ldots, T-1$ would have a lower cost. Hence the original sequence of controls would not have been optimal.

Principle of dynamic programming. The essence of dynamic programming is to solve the larger, original problem by sequentially solving the truncated sub-problems. At each iteration, Dijkstra's algorithm constructs the functions

$$J_T^*(x_T), J_{T-1}^*(x_{T-1}), \ldots, J_0^*(x_0)$$

starting from J_T^* and proceeding backwards to $J_{T-1}^*, J_{T-2}^* \ldots$. The function $J_{T-k}^*(v)$ is just the array $\text{dist}(v)$ at iteration k of the *backwards* implementation of Dijkstra's algorithm. Mathematically, dynamic programming looks as follows.

1. Initialize $J_T^*(x) = q_f(x)$ for all $x \in X$.
2. For iteration $k = T-1, \ldots, 0$, set

$$J_k^*(x) = \min_{u_k \in U} \left\{ q_k(x, u_k) + J_{k+1}^*(f_k(x, u_k)) \right\} \tag{5.4}$$

for all $x \in X$.

After running the above algorithm we have the optimal cost-to-go $J_0^*(x)$ for each state $x \in X$, in particular, we have the cost-to-go for the initial state $J_0^*(x_0)$. If we remember the minimizer u_k^* in (5.4) while running the algorithm, we also have the optimal sequence $(u_0^*, u_1^*, \ldots, u_{T-1}^*)$. The function $J_0^*(x)$ (often shortened to simply $J^*(x)$) is the optimal cost-to-go from the state $x \in X$.

Again, we really only wanted to calculate $J_0^*(x_0)$ but had to do all this extra work of computing J_k^* for all the states.

Curse of dimensionality What is the complexity of running dynamic programming? The cost of the minimization over U is $\mathcal{O}(|U|)$, it is a bunch of comparisons between floats. The number of operations at each iteration for setting the values $J_k^*(x)$ for all $x \in X$ is $|X|$. So the total complexity is $\mathcal{O}(T\,|X|\,|U|)$.

The terms $|X|$ and $|U|$ are often the hurdle in implementing dynamic programming or any variant of it. Think of the grid-world in Problem 1 in HW 1, it had 200×200 cells which amounts to $|X| = 40,000$. This may seem a reasonable number but it explodes quickly as the dimensionality of the state-space grows. For a robot manipulator with six degrees-of-freedom, if we discretize each joint angle into 5 degree cells, the number of states is $|X| \approx 140$ billion. The number of states $|X|$ is exponential in the dimensionality of the state-space and dynamic programming quickly becomes prohibitive beyond 4 dimensions or so. Bellman called this the

curse of dimensionality.

Cost of dynamic programming is linear in the time-horizon Notice a very important difference between (5.4)

$$J_k^*(x) = \min_{u_k \in U} \left\{ q_k(x, u_k) + J_{k+1}^*(f_k(x, u_k)) \right\}$$

for iterations $i = T-1, \ldots, 0$ and (5.3)

$$J^*(x_0) = \min_{u_k \in U, k=0,\ldots,T-1} J(x_0; u_0, \ldots, u_{T-1}).$$

The latter has a minimization over a sequence of controls $(u_0, u_1, \ldots, u_{T-1})$ while the former has a minimization over only the control at time k, u_k over T iterations. The former is much much easier to solve because it is a sequence of $\mathcal{O}(T)$ smaller optimization problems: it is really easy to compute $\min_{u_k \in U}$) for each state x separately than to solve the gigantic minimization problem in (5.3) because in the latter case, the variable of optimization is the entire control trajectory and has size $|U|^T$.

❓ The principle of dynamic programming gives us a way to solve an optimization problem (5.3) over a really large space (the space of all control trajectories) using a linear in time-horizon number of optimization problems (5.4). Can we split any optimization problem into sub-problems like this?

Dynamic programming and Viterbi's algorithm We have seen the principle of dynamic programming in action before in Viterbi's algorithm in Chapter 2. The transition graph in Figure 5.1 is the same as the Trellis graph for Viterbi's algorithm, the run-time cost was

$$q_k(x_k, u_k) := -\log \mathrm{P}(Y_k \mid X_k) - \log \mathrm{P}(X_{k+1} \mid X_k)$$

and instead of a terminal cost q_f, we had an initial cost $-\log \mathrm{P}(X_1)$. Viterbi's algorithm computed the most likely path given observations of the HMM, i.e., the path $(X_1, \ldots, X_T)$ that maximizes the probability $\mathrm{P}(X_1, \ldots, X_T \mid Y_1, \ldots, Y_T)$ is simply the solution of dynamic programming for the Trellis graph.

❓ How should one modify dynamic programming if we have a non-additive cost, e.g., the runtime cost at time k given by q_k is a function of both x_k and x_{k-1}?

5.3.1 Q-factor

The quantity

$$Q_k^*(x, u) := q_k(x, u) + J_{k+1}^*(f_k(x, u))$$

is called the Q-factor. It is simply the expression that is minimized in the right-hand side of (5.4) and denotes the cost-to-go if control u was picked at state x (corresponding to cost $q_k(x, u)$) and the the optimal control trajectory was followed after that (corresponding to cost $J_k^*(f_k(x, u))$ from state $x' = f_k(x, u)$). This nomenclature was introduced by Watkins in his thesis.

Q-factors and the cost-to-go are equivalent ways of thinking about dynamic programming. Given the Q-factor, we can obtain the cost-to-go

J_k^* as

$$J_k^*(x) = \min_{u_k \in U} Q_k^*(x, u_k). \tag{5.5}$$

which is precisely the dynamic programming update (by definition) in (5.4). We can also write dynamic programming completely in terms of Q-factors as follows.

Dynamic programming written in terms of the Q-factor

1. Initialize $Q_T^*(x, u) = q_f(x)$ for all $x \in X$ and all $u \in U$.
2. For iteration $k = T-1, \ldots, 0$, set

$$Q_k^*(x, u) = q_k(x, u) + \min_{u' \in U} Q_{k+1}^*(f_k(x, u), u'). \tag{5.6}$$

for all $x \in X$ and all $u \in U$.

As yet, it may be seem unnecessary to think of the Q-factor (which is a larger array with $|X| \times |U|$ entries) instead of the cost-to-go (which only has $|X|$ entries in the array).

Value function The following terminology is commonly used in the literature

$$\text{value function} \equiv \text{cost-to-go } J^*(x)$$
$$\text{action-value function} \equiv \text{Q-factor } Q^*(x, u).$$

Since the two functions are equivalent, we will call both as "value functions". The difference will be clear from context.

5.4 Stochastic dynamic programming: Value Iteration

Let us now see how dynamic programming looks for a Markov Decision Process (MDP). As we saw in Chapter 3, we can think of MDPs as stochastic dynamical systems denoted by

$$x_{k+1} = f_k(x_k, u_k) + \epsilon_k; \; x_0 \text{ is given.}$$

We will assume that we know the statistics of the noise ϵ_k at each time-step (say it is a Gaussian). Stochastic dynamical systems are very different from deterministic dynamical systems, given the same sequence of controls $(u_0, \ldots, u_{T-1})$, we may get different state trajectories $(x_0, x_1, \ldots, x_T)$ depending upon the realization of noise $(\epsilon_0, \ldots, \epsilon_{T-1})$. How should we find a good control trajectory then? One idea is to modify (5.3) to

minimize the expected value of the cost over all possible state-trajectories

$$J(x_0; u_0, \ldots, u_{T-1}) = \mathop{\mathrm{E}}_{(\epsilon_0,\ldots,\epsilon_{T-1})}\left[q_f(x_T) + \sum_{k=0}^{T-1} q_k(x_k, u_k)\right] \quad (5.7)$$

Suppose we minimized the above expectation and obtained the value function $J^*(x_0)$ and the optimal control trajectory $(u_0^*, \ldots, u_{T-1}^*)$. As the robot starts executing this trajectory, the realized versions of the noise ϵ_k might differ a lot from their expected value, and the robot may find itself in very different states x_k than the average-case states considered in (5.10).

Draw the picture of a one-dimensional stochastic dynamical system (random walk on a line) and see that the realized trajectory of the system can be very different from the average trajectory.

Feedback controls The concept of feedback control is a powerful way to resolve this issue. Instead of seeking $u_k^* \in U$ as the solutions of (5.10), we instead seek a *function*

$$u_k(x) : X \mapsto U \quad (5.8)$$

that maps the state-space X to a control U. Effectively, given a feedback control $u_k(x)$ the robot knows what control to apply at its current realized state $x_k \in X$, namely $u_k(x_k)$, even if the realized state x_k is very different from the average-case state. Feedback controls are everywhere and are critical to using controls in the real world. For instance, when you tune the shower faucet to give you a comfortable water temperature, you are constantly estimating the state (feedback using the temperature) and turning the faucet accordingly. Doing this without feedback would leave you terribly cold or scalded. We will denote the space of all feedback controls $u_k(\cdot)$ that depend on the state $x \in X$ by

$$u_k(\cdot) \in \mathcal{U}(X).$$

Control policy A sequence of feedback controls

$$\pi = (u_0(\cdot), u_1(\cdot), \ldots, u_{T-1}(\cdot)). \quad (5.9)$$

is called a control policy. This is an object that we will talk about often. It is important to remember that a control policy is set of controllers (usually feedback controls) that are executed at each time-step of a dynamic programming problem.

The **stochastic optimal control problem** finds a sequence of

feedback controls $(u_0(\cdot), u_1(\cdot), \ldots, u_T(\cdot))$ that minimizes

$$J(x_0; u_0(\cdot), \ldots, u_{T-1}(\cdot)) = \mathop{\mathrm{E}}_{(\epsilon_0,\ldots,\epsilon_{T-1})} \left[q_f(x_T) + \sum_{k=0}^{T-1} q_k(x_k, u_k(x_k)) \right].$$

The value function is given by

$$J^*(x_0) = \min_{u_k(\cdot)\in\mathcal{U}(X),\ k=0,\ldots,T-1} J(x_0; u_0(\cdot), \ldots, u_{T-1}(\cdot)) \tag{5.10}$$

The optimal sequence of feedback controls (in short, the optimal control trajectory) is the one that achieves this minimum.

All this sounds very tricky and abstract but you will quickly get used to the idea of feedback control because it is quite natural. You can think of feedback control as being analogous to the innovation term in the Kalman filter $K(y_k - C\mu_{k+1|k})$ which corrects the estimate $\mu_{k+1|k}$ to get a new estimate $\mu_{k+1|k+1}$ using the *current* observation y_k. Filtering would not work at all if the innovation term did not depend upon the actual observation y_k and only depended upon some average observation.

Dijkstra's algorithm no longer works, as is, if the edges in the graph are stochastic but we can use the principal of dynamic programming to write the solution for the stochastic optimal control problem. The idea remains the same, we compute a sequence of cost-to-go functions $J_T^*(x), J_{T-1}^*(x), \ldots, J_0^*(x)$, and in particular $J_0^*(x_0)$, proceeding *backwards*.

Finite-horizon dynamic programming for stochastic systems.

1. Initialize $J_T^*(x) = q_f(x)$ for all $x \in X$.
2. For all times $k = T-1, \ldots, 0$, set

$$J_k^*(x) = \min_{u_k(\cdot)\ \in\ \mathcal{U}(X)} \left\{ q_k(x, u_k(x)) + \mathop{\mathrm{E}}_{\epsilon_k} \left[J_{k+1}^*(f_k(x, u_k(x)) + \epsilon_k) \right] \right\} \tag{5.11}$$

for all $x \in X$.

Just like (5.4), we solve a sub-problem for one time-instant at each iteration. But observe a few importance differences in (5.11) compared to (5.4).

1. There is an expectation over the noise ϵ_k in the second term in the curly brackets. The second term in the curly brackets is the average of the cost-to-go of the truncated sub-problems from time $k+1, \ldots, T$ over all possible starting states $x' = f_k(x_k, u_k(x_k)) + \epsilon_k$. This makes sense, after taking the control $u_k(x_k)$, we may find the robot at any of the possible states $x' \in X$ depending upon different realizations of noise ϵ_k and the cost-to-go from x_k is therefore the average of the cost-to-go from each of those states (according to the principal of dynamic programming).
2. The minimization in (5.11) is performed over a function

$$\mathcal{U}(X) \ni u_k(\cdot) : X \mapsto U.$$

Since our set of states and controls is finite, this involves finding a table of size $|X| \times |U|$ for each iteration. In (5.4), we only had

to search over a set of values $u_k \in U$ of size $|U|$. At the end of dynamic programming, we have a sequence of feedback controls

$$(u_0^*(\cdot), u_1^*(\cdot), \ldots, u_{T-1}^*(\cdot)).$$

Each feedback control $u_k^*(x)$ tells us what control the robot should pick if it finds itself at a state x at time k.

3. If we know the dynamical system, not in its functional form $x_{k+1} = f_k(x_k, u_k) + \epsilon_k$ but rather as a transition matrix $\mathrm{P}(x_{k+1} \mid x_k, u_k)$ (like we had in Chapter 2) then the expression in (5.11) simply becomes

$$J_k^*(x) = \min_{u_k(\cdot) \in \mathcal{U}(X)} \left\{ q_k(x, u_k(x)) + \mathop{\mathrm{E}}_{x' \sim \mathrm{P}(\cdot \mid x_k, u_k(x_k))} \left[J_{k+1}^*(x') \right] \right\} \tag{5.12}$$

Why should we only care about minimizing the average cost in the objective in (5.10)? Can you think of any other objective we may wish to use?

Computational complexity The form in (5.12) helps us understand the computational complexity, each sub-problem performs $|X| \times |X| \times |U|$ amount of work and therefore the total complexity of stochastic dynamic programming is

$$\mathcal{O}(T|X|^2|U|).$$

Naturally, the quadratic dependence on the size of the state-space is an even bigger hurdle while implementing dynamic programming for stochastic systems.

5.4.1 Infinite-horizon problems

In the previous section, we put a lot of importance on the horizon T for dynamic programming. This is natural: if the horizon T changes, say you are in a hurry to get to school, the optimal trajectory may take control inputs that incur a lot of runtime cost simply to reach closer to the goal state (something that keeps the terminal cost small). In most, real-world problems, it is not very clear what value of T we should pick. We therefore formulate the dynamic programming problem as something that also allows a trajectory of infinite steps but also encourages the length of the trajectory to be small enough in order to be meaningful. Such problems are called infinite-horizon problems ($T \to \infty$).

Stationary dynamics and run-time cost We think of infinite-horizon problems in the following way: at any time-step, the length of the trajectory *remaining* for the robot to traverse is infinite. It helps in this case to solve a restricted set of problems where the system dynamics and run-time cost do not change as a function of time (they only change as a function of the state and the control). We will set

$$q(x, u) \equiv q_k(x, u),$$
$$f(x, u) \equiv f_k(x, u)$$

for all $x \in X$ and $u \in U$. Such a condition is called stationarity. If the system is stochastic, we also require that the distribution of noise ϵ_k does not change as a function of time (it could change in (5.11) but we did not write it so). The infinite-horizon setting is never quite satisfied in practice but it is a reasonable formulation for problems that run for a long length of time.

Infinite-horizon objective The objective that we desire be minimized by an infinite-horizon control policy

$$\pi = (u_0(\cdot), u_1(\cdot), \ldots, u_T(\cdot), u_{T+1}(\cdot), \ldots,)$$

is defined in terms of an asymptotic limit

$$J(x_0; \pi) = \lim_{T \to \infty} \mathop{\mathrm{E}}_{(\epsilon_0, \ldots, \epsilon_{T-1})} \left[\sum_{k=0}^{T-1} \gamma^k q(x_k, u_k(x_k)) \right]. \tag{5.13}$$

and we again wish to solve for the optimal cost-to-go

$$J^*(x_0) = \operatorname*{argmin}_{\pi} J^*(x_0; \pi). \tag{5.14}$$

Thus the infinite horizon costs of a policy is the limit of its finite horizon costs as the horizon tends to infinity. Notice a few important differences when compared to (5.7).

1. The objective is a limit, it is effectively the cost of the trajectory as it is allowed to stretch for a larger and larger time-horizon.

2. There is no terminal cost in the objective function; this makes sense because an explicit terminal state x_T does not exist anymore. In infinite-horizon problems, you should think of the terminal cost as being incorporated inside the run-time cost $q(x, u)$ itself, e.g., move the robot to minimize the fuel used at *this* time instant but also move it in a way that it reaches the goal *at some time in the future*.

3. **Discount factor—** Depending upon what controls we pick, the summation
$$\sum_{k=0}^{T} q(x_k, u_k(x_k))$$
can diverge to infinity as $T \to \infty$ and thereby a meaningful solution to the infinite-horizon problem may not exist. In order to avoid this, we use a scalar
$$\gamma \in (0, 1)$$
known as the discount factor in the formulation. It puts more emphasis on costs incurred earlier in the trajectory than later ones and thereby encourages the length of the trajectory to be small. Notice that $\sum_{k=0}^{\infty} \alpha^k = 1/(1-\alpha)$ if $|a| < 1$, so if the cost $|q(x_k, u_k(x_k))| < 1$, then we know that the objective in (5.13) always converges.

Stochastic shortest path problems It is important to remember that the discount factor is chosen by the user, no one prescribes it. There is also a class of problems where we may choose $\gamma = 1$ but in these cases, there should exist some *essentially terminal* state in the state space where we can keep taking a control such that the runtime cost $q(x, u)$ is zero. Otherwise, the objective will diverge. The goal region in the grid-world problem could be an example of such state. Such problems are called stochastic shortest path problems because the time-horizon is not *actually* infinite, we just do not know how many time-steps it will take for the robot to go to the goal location. Naturally, stochastic shortest path problems are a generalization of the shortest path problem solved by Dijkstra's algorithm. The algorithms we discuss next will work for such problems.

Stationary policy It seems a bit cumbersome to carry around an infinitely long sequence of feedback controls in infinite-horizon problems. Since there is an infinitely-long trajectory yet to be traveled *at any given time-step*, the optimal control action that we take should only depend upon the current state. This is indeed true mathematically. If $J^*(x)$ is the optimal cost-to-go in the infinite-horizon problem starting from a state x, using the principle of dynamic programming, we should also have that we can split this cost as the best one-step cost of the current state x added to the optimal cost-to-go from the state $f(x, u)$ realized after taking the optimal control u:

$$J^*(x) = \min_{u(x) \in \mathcal{U}(X)} \mathop{\mathrm{E}}_{\epsilon} \left[q(x, u(x)) + J^*(f(x, u(x)) + \epsilon) \right]. \qquad (5.15)$$

We will study this equation in depth soon. But if we find the minimum at $u^*(x)$ for this equation, then we can run the policy

$$\pi^* = (u^*(\cdot), u^*(\cdot), \ldots, u^*(\cdot), \ldots)$$

for the entire infinite horizon. Such a policy is called a stationary policy. Intuitively, since the future optimization problem (tail of dynamic programming) from a given state x looks the same regardless of the time at which we start, optimal policies for the infinite-horizon problem can be found even inside the restricted class of policies where the feedback control does not change with time k.

We will almost exclusively deal with stationary policies in this course.

5.4.2 Dynamic programming for infinite-horizon problems

We wish to compute the optimal cost-to-go of starting from a state x and taking an infinitely long trajectory that minimizes the objective (5.13). We will exploit the equation in (5.15) and develop an iterative algorithm to compute the optimal cost-to-go $J^*(x)$.

Value Iteration . The algorithm proceeds iteratively to maintain a sequence of approximations

$$\forall x \in X, \quad J^{(0)}(x), J^{(1)}(x), J^{(2)}(x), \ldots,$$

to the optimal value function $J^*(x)$. Such an algorithm is called "value iteration".

1. Initialize $J^{(0)}(x) = 0$ for all $x \in X$.
2. Update using the Bellman equation at each iteration, i.e., for $i = 1, 2, \ldots, N$, set

$$J^{(i+1)}(x) = \min_{u \in U} \mathop{\mathrm{E}}_{\epsilon} \left[q(x,u) + \gamma J^{(i)}(f(x,u) + \epsilon) \right]. \quad (5.16)$$

   for all $x \in X$ until the value function converges at all states, e.g.,

$$\forall x \in X, \quad \left| J^{(i)}(x) - J^{(i+1)}(x) \right| < \text{small tolerance}.$$

3. Compute the feedback control and the stationary policy $\pi^* = (u^*(\cdot), \ldots,)$ corresponding to the value function estimate $J^{(N)}$ as

$$u^*(x) = \operatorname*{argmin}_{u \in U} \mathop{\mathrm{E}}_{\epsilon} \left[q(x,u) + \gamma J^{(N)}(f(x,u) + \epsilon) \right] \quad (5.17)$$

   for all $x \in X$.

ℹ If the dynamics is given as a transition matrix, we can replace the expectation over noise E_ϵ as an expectation over the next state $x' \sim \mathrm{P}(x' \mid x, u(x))$ in (5.16) to run value iteration. Everything else remains the same

Let us observe a few important things in the above sequence of updates. First, at each iteration, we are updating the values of all $|X|$ states. This involves $|X|^2|U|$ amount of work per iteration. How many such iterations N do we need until the value function converges? We will see in a bit, that

$$\forall x \in X, \quad J^*(x) = \lim_{N \to \infty} J^{(N)}(x).$$

Again, we really only wanted to compute the cost-to-go $J^*(x_0)$ from some initial state x_0 but computed the value function at all states $x \in X$.

Q-Iteration Just like we wrote dynamic programming in terms of the Q-factor, we can also write value iteration to find the optimal Q-factor $Q^*(x,u)$, i.e., the optimal cost-to-go of the infinitely-long trajectory that starts at state x, takes a control u at the first time-step and therefore follows the optimal policy.

1. We can again initialize $Q^{(0)}(x,u) = q(x,u)$ for all $x \in X$ and $u \in U$.

2. The Bellman update in terms of the Q-factor becomes

$$Q^{(i+1)}(x,u) = \mathop{\mathrm{E}}_{\epsilon}\left[q(x,u) + \gamma \min_{u' \in U} Q^{(i)}(f(x,u)+\epsilon, u')\right] \tag{5.18}$$

   and update this for all $x \in X$ and all $u \in U$.

3. The feedback control is the control at that state that minimizes the Q-factor

$$\forall x \in X, \quad u^*(x) = \operatorname*{argmin}_{u' \in U}\ Q^{(N)}(x,u') \tag{5.19}$$

   and the control policy is

$$\pi^* = (u^*, u^*, \ldots,)$$

   Notice how we can directly find the u' that has the smallest value of $Q^{(N)}$ and set it to be our feedback control.

5.4.3 An example

Let us consider a grid-world example. A robot would like to reach a goal region (marked in green) and we are interested in computing the cost-to-go from different parts of the domain. Gray cells are obstacles that the robot cannot enter. At each step the robot can move in four directions (north, east, west, south) with a small dynamics noise which keeps it at the original cell in spite of taking the control. These pictures are when the run-time-cost is *negative*, i.e., the robot gets a certain reward $q(x,u)$ for taking the control u at cell x. Dynamic programming (and value iteration) also works in this case and we simply replace all minimizations by maximizations in the equations.

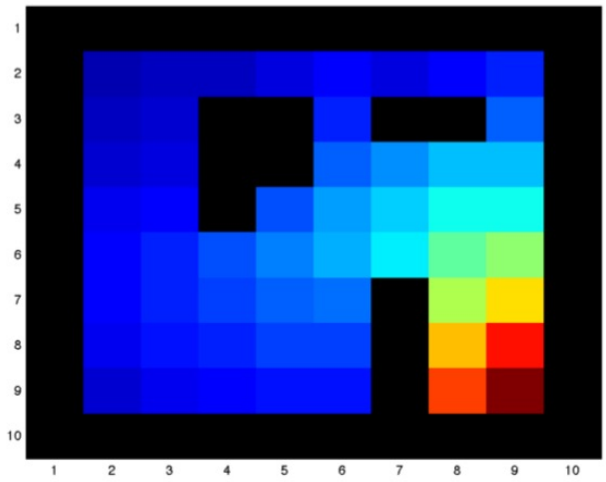
0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0

0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0.75	0
0	0	0	0	0	0	0	0.75	1	0
0	0	0	0	0	0	0	0	0	0

0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0.51	0
0	0	0	0	0	0	0	0.56	1.43	0
0	0	0	0	0	0	0	1.43	1.9	0
0	0	0	0	0	0	0	0	0	0

0	0	0	0	0	0	0	0	0	0
0	0.44	0.54	0.59	0.82	1.15	0.85	1.09	1.52	0
0	0.59	0.69	0	0	1.52	0	0	2.13	0
0	0.75	0.90	0	0	2.12	2.55	2.98	3.00	0
0	0.95	1.18	0	2.00	2.70	3.22	3.80	3.88	0
0	1.20	1.55	1.87	2.41	2.92	3.51	4.52	5.00	0
0	1.15	1.47	1.74	2.05	2.25	0	5.34	6.47	0
0	0.99	1.26	1.49	1.72	1.74	0	6.69	8.44	0
0	0.74	0.99	1.17	1.34	1.27	0	7.96	9.94	0
0	0	0	0	0	0	0	0	0	0

The final value function after 50 iterations looks as follows.



5.4.4 Some theoretical results on value iteration

We list down some very powerful theoretical results for value iteration. These results are valid under a very general set of conditions and make value iteration work for a large number of real-world problems; they are at the the heart of all modern algorithms. We will not derive them (it is easy but cumbersome) but you should commit them to memory and try to understand them intuitively.

Value iteration converges. Given *any* initialization $J^{(0)}(x)$ for all $x \in X$, the sequence of value iteration estimates $J^{(i)}(x)$ converges to the optimal cost

$$\forall x \in X, \quad J^*(x) = \lim_{N \to \infty} J^{(N)}(x)$$

The solution is unique. The optimal cost-to-go $J^*(x)$ of (5.14) satisfies the Bellman equation

$$J^*(x) = \min_{u \in U} \mathop{\mathrm{E}}_{\epsilon} \left[q(x,u) + \gamma J^*(f(x,u) + \epsilon) \right].$$

The function J^* is also the *unique* solution of this equation. In other words, if we find some other function $J'(x)$ that satisfies the Bellman equation, we are guaranteed that J' is indeed the optimal cost-to-go.

Policy evaluation: Bellman equation for a particular policy. Consider a stationary policy $\pi = (u(\cdot), u(\cdot), \ldots)$. The cost of executing this policy starting from a state x, is $J(x;\pi)$ from (5.13), also denoted by $J^\pi(x)$ for short. It satisfies the equation

$$J^\pi(x) = q(x,u(x)) + \gamma \mathop{\mathrm{E}}_{\epsilon} \left[J^\pi(f(x,u(x)) + \epsilon) \right] \tag{5.20}$$

and is also the unique solution of this equation. In other words, if we have a policy in hand, and wish to find the cost-to-go of this policy, i.e., "evaluate the policy" we can initialize $J^{(0)}(x) = 0$ for all $x \in X$ and perform the sequence of iterative updates to this initialization

$$J^{(i+1)}(x) = q(x,u(x)) + \gamma \mathop{\mathrm{E}}_{\epsilon} \left[J^{(i)}(f(x,u(x)) + \epsilon) \right]. \tag{5.21}$$

As the number of updates goes to infinity, the iterate converges to $J^\pi(x)$

$$\forall x \in X, \quad J^\pi(x) = \lim_{N \to \infty} J^{(N)}(x).$$

Policy evaluation is equivalent to solving a linear system of equations. Observe that the corresponding equation for policy equation (5.20) does not have the minimization over controls. This allows us to write the updates in (5.21) as the solution of a linear system of equations. Since we are in a finite state-space, we can write the cost-to-go as a large vector

$$J^\pi := [J^\pi(x_1), J^\pi(x_2), \ldots, J^\pi(x_n)]^\top$$

where n is the number of total states in the state-space. We create a similar vector for the run-time cost term

$$q^u := [q(x_1,u(x_1)), q(x_2,u(x_2)), \ldots, q(x_n,u(x_n))].$$

We know that the expectation over noise ϵ is equivalent to an expectation over the next state of the system, let us rewrite the dynamics part $f(x,u(x)) + \epsilon$ in terms of the Markov transition matrix

$$T_{x,x'} = \mathrm{P}(x' \mid x, u(x))$$

as

$$\gamma \mathop{\mathrm{E}}_{\epsilon} \left[J^\pi(f(x,u(x)) + \epsilon) \right] = \gamma \sum_{x'} T_{x,x'} J^\pi(x') = \gamma \, T J^\pi$$

to get a linear system

$$J^\pi = q^u + \gamma T J^\pi \tag{5.22}$$

which can be solved easily for $J^\pi = (I - \gamma\, T)^{-1}\, q^u$ to get the cost-to-go of a particular control policy π.

5.5 Stochastic dynamic programming: Policy Iteration

Value iteration converges exponentially quickly, but asymptotically. Note that the number of states $|X| = n$ is finite and so is the number of controls $|U|$. So this should seem funny, one would expect that we should be able to find the optimal cost $J^*(x)$ in finite time if the problem is finite. After all we need to find $|X|$ numbers $J^*(x_1), \ldots, J^*(x_n)$. This intuition is correct and, in this section, we will discuss an algorithm called policy iteration which is a more efficient version of value iteration.

The idea behind policy iteration is quite simple: given a stationary policy for an infinite-horizon problem $\pi = (u(\cdot), \ldots, u(\cdot))$, we can evaluate this policy to obtain its cost-to-go $J^\pi(x)$. If we now set the feedback control to be

$$\tilde{u}(x) = \underset{u \in U}{\operatorname{argmin}} \underset{\epsilon}{\mathrm{E}} \left[q(x,u) + \gamma J^\pi(f(x,u) + \epsilon) \right], \tag{5.23}$$

i.e., we construct a *new control policy* that finds the best control to execute in the first step $\tilde{u}(\cdot)$ and thereafter it executes the old feedback control $u(\cdot)$

$$\pi^{(1)} = (\tilde{u}(\cdot), u(\cdot), \ldots),$$

then the cost-to-go of the new policy $\pi^{(1)}$ has to be better:

$$\forall x \in X, \quad J^{\pi^{(1)}}(x) \leq J^\pi(x).$$

Why? It is simply because (5.23) is at least an improvement upon the feedback control $u(\cdot)$. The cost-to-go cannot improve only if the old feedback control $u(\cdot)$ where optimal to begin with.

We don't have to stop at one time-step, we can patch the old policy at the first two time-steps to get

$$\pi^{(2)} = (\tilde{u}(\cdot), \tilde{u}(\cdot), \ldots),$$

and have by the same logic

$$\forall x \in X, \quad J^{\pi^{(2)}}(x) \leq J^{\pi^{(1)}}(x) \leq J^\pi(x).$$

If we build a new stationary policy

$$\tilde{\pi} = (\tilde{u}(\cdot), \tilde{u}(\cdot), \tilde{u}(\cdot), \ldots), \tag{5.24}$$

we similarly have

$$\forall x \in X, \quad J^{\tilde{\pi}}(x) \leq J^\pi(x).$$

This suggests an iterative way to compute the optimal stationary policy

π^* starting from some initial stationary policy.

Policy Iteration The algorithm proceeds to maintain a sequence of stationary policies

$$\pi^{(k)} = (u^{(k)}(\cdot), u^{(k)}(\cdot), u^{(k)}(\cdot), \dots)$$

that converges to the optimal policy π^*.

Initialize $u^{(0)}(x) = 0$ for all $x \in X$. This gives the initial stationary policy $\pi^{(0)}$. At each iteration $k = 1, \dots,$ we do the following two things.

1. **Policy evaluation** Use multiple iterations of (5.21) to evaluate the old policy $\pi^{(k)}$. In other words, initialize $J^{(0)}(x) = 0$ for all $x \in X$ and iterate upon

   $$J^{(i+1)}(x) = q(x, u^{(i)}(x)) + \gamma \mathop{\mathrm{E}}_{\epsilon} \left[J^{(i)}(f(x, u^{(i)}(x) + \epsilon)) \right]$$

   for all $x \in X$ until convergence. In practice, we can use the linear system of equations in (5.22) to solve for $J^{\pi^{(k)}}$ directly.

2. **Policy improvement** Update the feedback controller using (5.23) to be

   $$u^{(k+1)}(x) = \operatorname*{argmin}_{u \in U} \mathop{\mathrm{E}}_{\epsilon} \left[q(x, u) + \gamma J^{\pi^{(k)}}(f(x, u) + \epsilon) \right]$$

   for all $x \in X$ and compute the updated stationary policy

   $$\pi^{(k+1)} = (u^{(k+1)}(\cdot), u^{(k+1)}(\cdot), \dots)$$

The algorithm terminates when the controller does not change *at any state*, i.e., when the following condition is satisfied

$$\forall x \in X, \quad u^{(k+1)}(x) = u^{(k)}(x).$$

Just like value iteration converges to the optimal value function, it can be shown that policy iteration produces a sequence of improved policies

$$\forall x \in X, \quad J^{\pi^{(k+1)}}(x) \leq J^{\pi^{(k)}}(x)$$

and converges to the optimal cost-to-go

$$\forall x \in X, \quad J^*(x) = \lim_{N \to \infty} J^{\pi^{(N)}}(x).$$

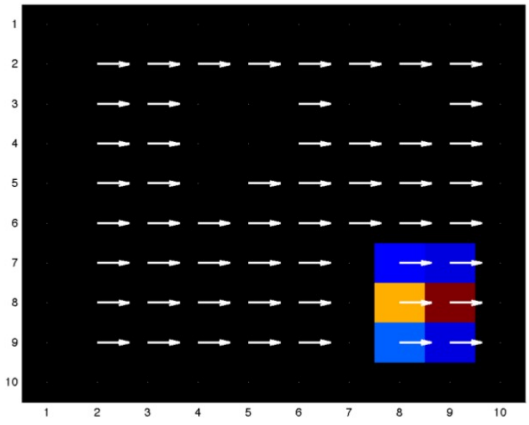
The key property of policy iteration is that we need a finite number of updates to the policy to find the optimal policy. Notice that this does not mean always mean that we are doing less work than value iteration

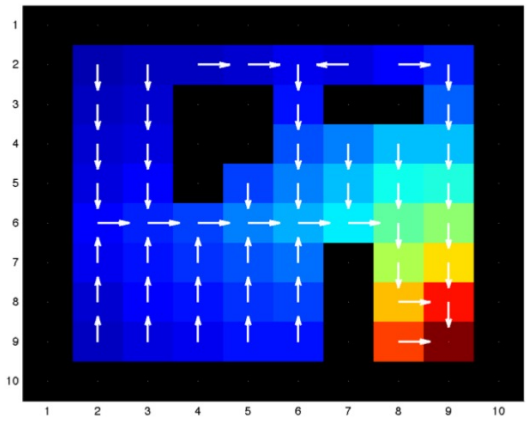
For large problems, we use methods for solving large linear systems such as Lanczos iteration. Typical policy evaluation problems are also sparse (why?) so we can use things like the Kaczmarz method to solve the linear system.

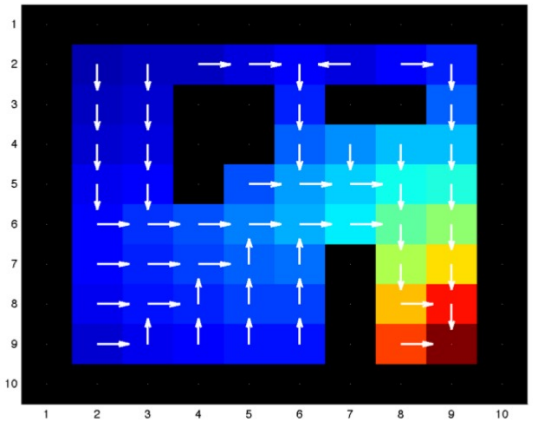
in policy iteration. Observe that the policy evaluation step in the policy iteration algorithm performs a number of Bellman equation updates. But typically, it is observed in practice that policy iteration is much cheaper computationally than value iteration.

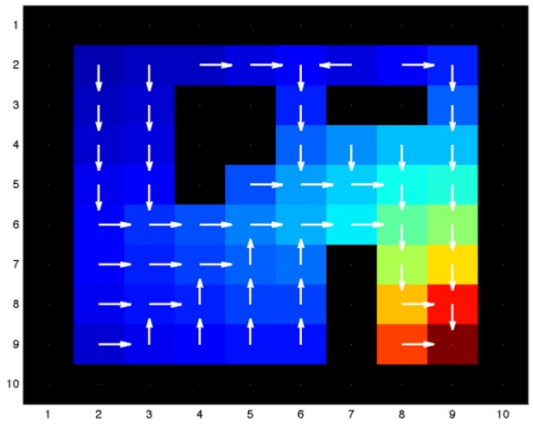
5.5.1 An example

Let us go back to our example for value iteration. In this case, we will visualize the controller $u^{(k)}(x)$ at each cell x as arrows pointing to some other cell. The cells are colored by the value function for that particular stationary policy.









The evaluated value for the policy after 4 iterations is optimal, compare this to the example for value iteration.

0	0	0	0	0	0	0	0	0	0
0	0.45	0.56	0.61	0.84	1.17	0.87	1.11	1.5411	0
0	0.61	0.71	0	0	1.54	0	0	2.16	0
0	0.78	0.93	0	0	2.16	2.59	3.02	3.03	0
0	0.98	1.21	0	2.03	2.74	3.26	3.84	3.91	0
0	1.23	1.58	1.90	2.44	2.95	3.54	4.56	5.03	0
0	1.18	1.50	1.78	2.09	2.28	0	5.38	6.51	0
0	1.02	1.29	1.52	1.76	1.77	0	6.74	8.49	0
0	0.76	1.02	1.20	1.37	1.30	0	8.01	10	0
0	0	0	0	0	0	0	0	0	0

Chapter 6

Linear Quadratic Regulator (LQR)

Reading

1. http://underactuated.csail.mit.edu/lqr.html, Lecture 3-4 at https://ocw.mit.edu/courses/aeronautics-and-astronautics/16-323-principles-of-optimal-control-spring-2008/lecture-notes
2. Optional: Applied Optimal Control by Bryson & Ho, Chapter 4-5

This chapter is the analogue of Chapter 3 on Kalman filtering. Just like Chapter 2, the previous chapter gave us two algorithms, namely value iteration and policy iteration, to solve dynamic programming problems for a finite number of states and a finite number of controls. Solving dynamic programming problems is difficult if the state/control space are infinite. In this chapter, we will look at an important and powerful special case, called the Linear Quadratic Regulator (LQR), when we can solve dynamic programming problems easily. Just like a lot of real-world state-estimation problems can be solved using the Kalman filter and its variants, a lot of real-world control problems can be solved using LQR and its variants.

6.1 Discrete-time LQR

Consider a deterministic, *linear* dynamical system given by

$$x_{k+1} = Ax_k + Bu_k; \quad x_0 \text{ is given.}$$

where $x_k \in \mathbb{R}^d$ and $u_k \in \mathbb{R}^m$ which implies that $A \in \mathbb{R}^{d\times d}$ and $B \in \mathbb{R}^{d\times m}$. In this chapter, we are interested in calculating a feedback control $u_k = u(x_k)$ for such a system. Just like we formulated the problem

in dynamic programming, we want to pick a feedback control which leads to a trajectory that achieves a minimum of some run-time cost and a terminal cost. We will assume that both the run-time and terminal costs are *quadratic* in the state and control input, i.e.,

$$q(x,u) = \frac{1}{2}x^\top Q x + \frac{1}{2}u^\top R u \tag{6.1}$$

where $Q \in \mathbb{R}^{d\times d}$ and $R \in \mathbb{R}^{m\times m}$ are symmetric, positive semi-definite matrices

$$Q = Q^\top \succeq 0, \quad R = R^\top \succeq 0.$$

Effectively, if Q were a diagonal matrix, a large diagonal entry would Q_{ii} models our desire that the trajectory of the system should not have a large value of the state x_i along its trajectories. We want these matrices to be positive semi-definitive to prevent dynamic programming from picking a trajectory which drives down the run-time cost to negative infinity by picking.

Example Consider the discrete-time equivalent of the so-called double integrator $\ddot{z}(t) = u(t)$. The linear system in this case (obtained by creating two states $x := [z(t), \dot{z}(t)]$ is

$$x_{k+1} = \begin{bmatrix} 1 & \Delta t \\ 0 & 1 \end{bmatrix} x_k + \begin{bmatrix} 0 \\ \Delta t \end{bmatrix} u_k.$$

ℹ This system is called the double integrator because of the structure $\ddot{z} = u$; if z denotes the position of an object the equation is simply Newton's law which connects the force applied u to the acceleration.

First, note that a continuous-time linear dynamical system $\dot{x} = Ax$ is asymptotically stable, i.e., from any initial condition $x(0)$ its trajectories go to the equilibrium point $x = 0$ ($x(t) \to 0$ as $t \to \infty$). Asymptotic stability for continuous-time dynamical systems occurs if all eigenvalues of A are strictly negative. A discrete-time linear dynamical system $x_{k+1} = Ax_k$ is asymptotically stable if all eigenvalues of A have magnitude strictly smaller than 1, $|\lambda(A)| < 1$.

A typical trajectory of the double integrator will look as follows. Suppose we would like to pick a different controller that more quickly brings the system to its equilibrium. One way of doing so is to minimize

$$J = \sum_{k=0}^{T} \|x_k\|^2$$

which represents how far away both the position and velocity are from zero over all times k. The following figure shows the trajectory that achieves a small value of J.

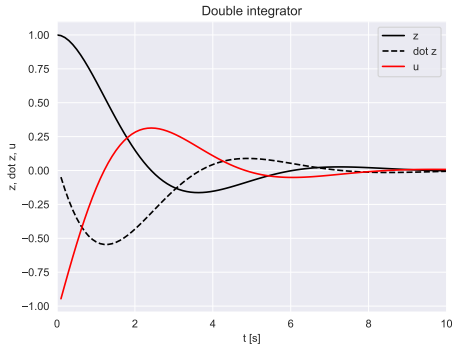


Figure 6.1: The trajectory of $z(t)$ as a function of time t for a double integrator $\ddot{z}(t) = u$ where we have chosen a stabilizing (i.e., one that makes the system asymptotically stable) controller $u = -z(t) - \dot{z}(t)$. Notice how the trajectory starts from some initial condition (in this case $z(0) = 1$ and $\dot{z}(0) = 0$) and moves towards its equilibrium point $z = \dot{z} = 0$.

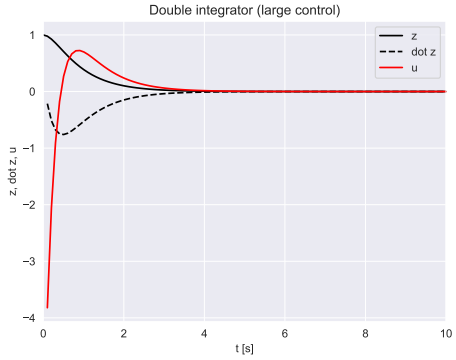


Figure 6.2: The trajectory of $z(t)$ as a function of time t for a double integrator $\ddot{z}(t) = u$ where we have chosen a large stabilizing control at each time $u = -5z(t) - 5\dot{z}(t)$. Notice how quickly the state trajectory converges to the equilibrium without much oscillation as compared to Figure 6.1 but how large the control input is at certain times.

This is obviously undesirable for real systems where we may want the control input to be bounded between some reasonable values (a car cannot accelerate by more than a certain threshold). A natural way of enforcing this is to modify our our desired cost of the trajectory to be

$$J = \sum_{k=0}^{T} \left(\|x_k\|^2 + \rho \|u_k\|^2 \right)$$

where the value of the parameter ρ is something chosen by the user to give a good balance of how quickly the trajectory reaches the equilibrium point and how much control is exerted while doing so. Linear-Quadratic-Regulator (LQR) is a generalization of this idea, notice that the above example is equivalent to setting $Q = I_{d\times d}$ and $R = \rho I_{m\times m}$ for the run-time cost in (6.1).

Back to LQR With this background, we are now ready to formulate the Linear-Quadratic-Regulator (LQR) problem which is simply dynamic programming for a linear dynamical system with quadratic run-time cost. In order to enable the system to reach the equilibrium state even if we have only a finite time-horizon, we also include a quadratic cost

$$q_f(x) = \frac{1}{2} x^\top Q_f x. \tag{6.2}$$

The dynamic programming problem is now formulated as follows.

Finite time-horizon LQR problem Find a sequence of control inputs $(u_0, u_1, \dots, u_{T-1})$ such that the function

$$J(x_0; u_0, u_1, \dots, u_{T-1}) = \frac{1}{2} x_T^\top Q_f x_T + \frac{1}{2} \sum_{k=0}^{T-1} \left(x_k^\top Q x_k + u_k^\top R u_k \right) \tag{6.3}$$

is minimized under the constraint that $x_{k+1} = Ax_k + Bu_k$ for all times $k = 0, \dots, T-1$ and x_0 is given.

6.1.1 Solution of the discrete-time LQR problem

We know the principle of dynamic programming and can apply it to solve the LQR problem. As usual, we will compute the cost-to-go of a trajectory that starts at some state x and goes further by $T-k$ time-steps, $J_k(x)$ backwards. Set

$$J_T^*(x) = \frac{1}{2} x^\top Q_f x \quad \text{for all } x.$$

Using the principle of dynamic programming, the cost-to-go J_{T-1} is given by

$$\begin{aligned} J_{T-1}^*(x_{T-1}) &= \min_u \left\{ \frac{1}{2} \left(x_{T-1}^\top Q x_{T-1} + u^\top R u \right) + J_T^*(A x_{T-1} + B u) \right\} \\ &= \min_u \left\{ \frac{1}{2} \left(x_{T-1}^\top Q x_{T-1} + u^\top R u + (Ax + Bu)^\top Q_f (A x_{T-1} + B u) \right) \right\}. \end{aligned}$$

We can now take the derivative of the right-hand side with respect to u to get

$$\begin{aligned} 0 &= \frac{\mathrm{dRHS}}{\mathrm{d}u} \\ &= Ru + B^\top Q_f (A x_{T-1} + B u) \\ \Rightarrow u_{T-1}^* &= -(R + B^\top Q_f B)^{-1} B^\top Q_f A \, x_{T-1} \\ &\equiv -K_{T-1} \, x_{T-1}. \end{aligned} \tag{6.4}$$

where

$$K_{T-1} = (R + B^\top Q_f B)^{-1} B^\top Q_f A$$

is (surprisingly) also called the Kalman gain. The second derivative is positive semi-definite

$$\frac{\mathrm{d}^2\text{RHS}}{\mathrm{d}u^2} = R + B^\top Q_f B \succeq 0$$

so we know that u^*_{T-1} is a minimum of the convex quantity on the right-hand side. Notice that the optimal control u^*_{T-1} is a linear function of the state x_{T-1}. Let us now expand the cost-to-go J_{T-1} using this optimal value (the subscript $T-1$ on the curly bracket simply means that all quantities are at time $T-1$)

$$\begin{aligned} J^*_{T-1}(x_{T-1}) &= \frac{1}{2}\left\{x^\top Q x + {u^*}^\top R u^* + (Ax + Bu^*)^\top Q_f (Ax + Bu^*)\right\}_{T-1} \\ &= \frac{1}{2} x^\top_{T-1} \left\{Q + K^\top R K + (A - BK)^\top Q_f (A - BK)\right\}_{T-1} x_{T-1} \\ &\equiv \frac{1}{2} x^\top_{T-1} P_{T-1} x_{T-1} \end{aligned}$$

where we set the stuff inside the curly brackets to the matrix P which is also positive semi-definite. This is great, the cost-to-go is also a quadratic function of the state x_{T-1}. Let us assume that this pattern holds for all time steps and the cost-to-go of the optimal LQR trajectory starting from a state x and proceeding forwards for $T-k$ time-steps is

$$J^*_k(x) = \frac{1}{2} x^\top P_k x.$$

We can now repeat the same exercise to get a recursive formula for P_k in terms of P_{k+1}. This is the *solution* of dynamic programming for the LQR problem and it looks as follows.

$$\begin{aligned} P_T &= Q_f \\ K_k &= \left(R + B^\top P_{k+1}\, B\right)^{-1}\, B^\top P_{k+1}\, A \\ P_k &= Q + K_k^\top R\, K_k + \left(A - BK_k\right)^\top P_{k+1}\, \left(A - BK_k\right), \end{aligned} \tag{6.5}$$

for $k = T-1, T-2, \ldots, 0$. There are a number of important observations to be made from this calculation:

1. The optimal controller $u^*_k = -K_k x_k$ is a linear function of the state x_k. This is only true for linear dynamical systems with quadratic costs. Notice that both the state and control space are infinite sets but we have managed to solve the dynamic programming problem to get the optimal controller. We could not have done it if the run-time/terminal costs were not quadratic or if the dynamical system were not linear. Can you say why?

2. The cost-to-go matrix P_k and the Kalman gain K_k do not depend upon the state and can be computed ahead of time if we know what the time horizon T is going to be.

3. The Kalman gain changes with time k. Effectively, the LQR

controller picks a large control input to quickly reduce the run-time cost at the beginning (if the initial condition were such that the run-time cost of the trajectory would be very large) and then gets into a balancing act where it balances the control effort and the state-dependent part of the run-time cost. LQR is an optimal way to strike a balance between the two examples in Figure 6.1 and Figure 6.2.

The careful reader will notice how the equations in (6.5) and our remarks about them are similar to the update equations of the Kalman filter and our remarks there. In fact we will see shortly how spookily similar the two are. The key difference is that Kalman filter updates run forwards in time and update the covariance while LQR updates run backwards in time and update the cost-to-go matrix P. This is not surprising because LQR is an optimal control problem, its update equations should run backward in time like the Dijkstra's algorithm.

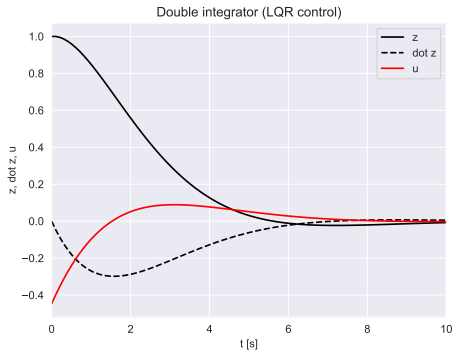


Figure 6.3: The trajectory of $z(t)$ as a function of time t for a double integrator $\ddot{z}(t) = u$ where we have chosen a controller obtained from LQR with $Q = I$ and $R = 5$. This gives the controller to be about $u = -0.45z(t) - 1.05\dot{z}(t)$. Notice how we still get stabilization but the control acts more gradually. Using different values of R, we can get many different behaviors. Another key aspect of LQR as compared to Figure 6.1 where the control was chosen in an ad hoc fashion is to let us prescribe the quality of state trajectories using high-level quantities like Q, R.

ℹ If you are trying this example yourself, I used the formula for continuous-time LQR and then discretized the controller while implementing it. We will see this in Section 6.2

6.2 Hamilton-Jacobi-Bellman equation

This section will show how the principle of dynamic programming looks for continuous-time deterministic dynamical systems

$$\dot{x} = f(x, u), \quad \text{with } x(0) = x_0.$$

As we discussed in Chapter 3, we can think of this as the limit of discrete-time dynamical system $x_{k+1} = f^{\text{discrete}}(x_k, u_k)$ as the time discretization goes to zero. Just like we have a sequence of controls in the discrete-time case, we have a continuous curve that determines the control (let us also call it the control sequence)

$$\{u(t) : \ t \in \mathbb{R}_+\}$$

which gives rise to a trajectory of the states

$$\{x(t):\ t \in \mathbb{R}_+\}$$

for the dynamical system. Let us consider the case when we want to find control sequences that minimize the integral of the cost along the trajectory that stops at some fixed, finite time-horizon T:

$$q_f(x(T)) + \int_0^T q(x(t), u(t))\ \mathrm{d}t\,.$$

This cost is again a function of the run-time cost and a terminal cost.

Since $\{x(t)\}_{t\geq 0}$ and $\{u(t)\}_{t\geq 0}$ are continuous curves and the cost is now a function of a continuous-curve, mathematicians say that the cost is a "functional" of the state and control trajectory.

Continuous-time optimal control problem We again want to solve for

$$J^*(x_0) = \min_{u(t),\ t\in[0,T]} \left\{ q_f(x(T)) + \int_0^T q(x(t), u(t))\ \mathrm{d}t \right\} \quad (6.6)$$

with the system satisfying $\dot{x} = f(x, u)$ at each time instant. Notice that the minimization is over a function of time $\{u(t) : t \in [0, T]\}$ as opposed to a discrete-time sequence of controls that we had in the discrete-time case. We will next look at the Hamilton-Jacobi-Bellman equation which is a method to solve optimal-control problems of this kind.

The principle of dynamic programming principle is still valid: if we have an optimal control trajectory $\{u^*(t) : t \in [0, T]\}$ we can chop it up into two parts at some intermediate time $t \in [0, T]$ and claim that the tail is optimal. In preparation for this, let us define the cost-to-go of going forward by $T - t$ time as

$$J^*(x, t) = \min_{u(s),\ s\in[t,T]} \left\{ q_f(x(T)) + \int_t^T q(x(s), u(s))\ \mathrm{d}s \right\},$$

the cost incurred if the trajectory starts at state x and goes forward by $T-t$ time. This is very similar to the cost-to-go $J_k^*(x)$ we had in discrete-time dynamic programming. Dynamic programming now gives

$$\begin{aligned} J^*(x(t), t) &= \min_{u(s),\ t\leq s\leq T} \left\{ q_f(x(T)) + \int_t^T q(x(s), u(s))\ \mathrm{d}s \right\} \\ &= \min_{u(s),\ t\leq s\leq T} \left\{ q_f(x(T)) + \int_t^{t+\Delta t} q(x(s), u(s))\ \mathrm{d}s + \int_{t+\Delta t}^T q(x(s), u(s))\ \mathrm{d}s \right\} \\ &= \min_{u(s),\ t\leq s\leq t+\Delta t} \left\{ J^*(x(t+\Delta t), t+\Delta t) + \int_t^{t+\Delta t} q(x(s), u(s))\ \mathrm{d}s \right\}. \end{aligned}$$

We now take the Taylor approximation of the term $J^*(x(t + \Delta t), t + \Delta t)$

as follows

$$\begin{aligned} & J^*(x(t+\Delta t), t+\Delta t) - J^*(x(t), t) \\ & \approx \partial_x J^*(x(t), t)\ (x(t+\Delta t) - x(t)) + \partial_t J^*(x(t), t)\Delta t \\ & \approx \partial_x J^*(x(t), t)\ f(x(t), u(t))\ \Delta t + \partial_t J^*(x(t), t)\Delta t \end{aligned}$$

where $\partial_x J^*$ and $\partial_t J^*$ denote the derivative of J^* with respect to its first and second argument respectively. We substitute this into the minimization and collect terms of Δt to get

$$0 = \partial_t\, J^*(x(t), t) + \min_{u(t)\in U} \left\{ q(x(t), u(t)) + f(x(t), u(t))\ \partial_x J^*(x(t), t) \right\}. \tag{6.7}$$

Notice that the minimization in (6.7) is only over *one* control input $u(t) \in U$, this is the control that we should take at time t. (6.7) is called the Hamilton-Jacobi-Bellman (HJB) equation. Just like the Bellman equation

$$J_k^*(x) = \min_{u\in U} \left\{ q_k(x, u) + J_{k+1}^*(f(x, u)) \right\}.$$

has two quantities x and the time k, the Hamilton-Jacobi-Bellman equation also has two quantities x and continuous time t. Just like the Bellman equation is solved backwards in time starting from T with $J_k^*(x) = q_f(x)$, the HJB equation is solved backwards in time by setting

$$J^*(x, T) = q_f(x).$$

> You should think of the HJB equation as the continuous-time, continuous-space analogue of Dijkstra's algorithm when the number of nodes in the graph goes to infinity and the length of each edge is also infinitesimally small.

6.2.1 Infinite-horizon HJB

The infinite-horizon problem with the HJB equation is easy: since we know that the optimal cost-to-go is not a function of time, we have

$$\partial_t J^*(x, t) = 0$$

and therefore $J^*(x)$ satisfies

$$0 = \min_{u\in U} \left\{ q(x, u) + f(x, u)\ \partial_x J^*(x) \right\}. \tag{6.8}$$

In this case, the above equation makes sense only if the integral of the run-time cost with the optimal controller $\int_0^\infty q(x(t), u^*(x(t)))\ \mathrm{d}t$ remains bounded and does not diverge to infinity. Therefore typically in this problem we will set $q(0, 0) = 0$, i.e., there is no cost for the system being at the origin with zero control, otherwise the integral of the run-time cost will never be finite. This also gives the boundary condition $J^*(0) = 0$ for the HJB equation.

6.2.2 Solving the HJB equation

The HJB equation is a partial differential equation (PDE) because there is one cost-to-go from every state $x \in X$ and for every time $t \in [0, T]$. It belongs to a large and important class of PDEs, collectively known as Hamilton-Jacobi-type equations. As you can imagine, since dynamic programming is so pervasive and solutions of DP are very useful in practice for a number of problems, there have been many tools invented to solve the HJB equation. These tools have applications to a wide variety of problems, from understanding how sound travels in crowded rooms to how light diffuses in an animated movie scene, to even obtaining better algorithms to train deep networks (https://arxiv.org/abs/1704.04932). HJB equations are usually never exactly solvable and a number of approximations need to be made in order to solve it.

> In this course, we will not solve the HJB equation. Rather, we are interested in seeing how the HJB equation looks for continuous-time linear dynamical systems (both deterministic and stochastic ones) and LQR problems for such systems, as done in the following section.

An example We will look at a classical example of the so-called car-on-the-hill problem given below. The state of the problem is the position

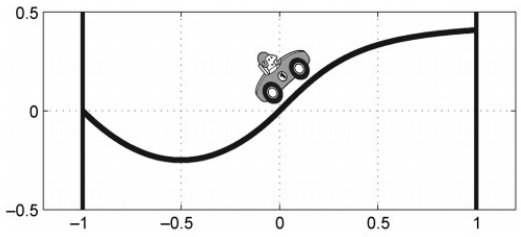


Figure 6.4: A car whose position is given by $z(t)$ would like to climb the hill to its right and reach the top with minimal velocity. The car rolls on the hill without friction. The run-time cost is zero everywhere inside the state-space. Terminal cost is -1 for hitting the left boundary ($z = -1$) and $-1 - \dot{z}/2$ for reaching the right boundary ($z = 1$). The car is a single integrator, i.e., $\dot{z} = u$ with only two controls ($u = 4$ and $u = -4$) and cannot exceed a given velocity (in this case $|\dot{z}| \leq 4$. This looks like a simple dynamic programming problem but it is quite hard due to the constraint on the velocity. The car may need to make multiple swing ups before it gains enough velocity (but not too much) to climb up the hill.

and velocity $(z, \dot{z})$ and we can solve a two-dimensional HJB equation to obtain the optimal cost-to-go from any state, as done by the authors Yuval Tassa and Tom Erez in "Least Squares Solutions of the HJB Equation With Neural Network Value-Function Approximators" (https://homes.cs.washington.edu/todorov/courses/amath579/reading/NeuralNet.pdf). In practice, while solving the HJB PDE, one discretizes the state-space at given set of states and solves the HJB equation (6.7) on this grid using

numerical methods (these authors used neural networks to solve it). The end result looks as follows.

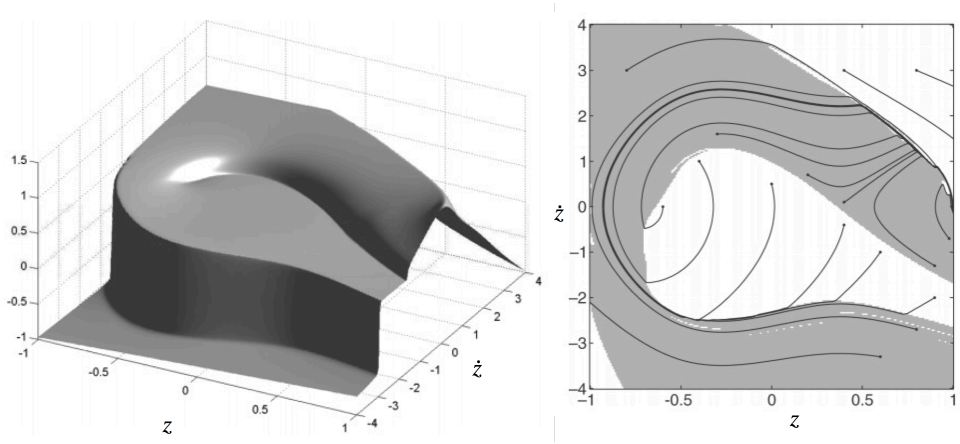


Figure 6.5: The left-hand side picture shows the infinite-horizon cost-to-go $J^*(z, \dot{z})$ for the car-on-the-hill problem. Notice how the value function is non-smooth at various places. This is quite typical of difficult dynamic programming problems. The right-hand side picture shows the optimal trajectories of the car $(z(t), \dot{z}(t))$; gray areas indicate maximum control and white areas indicate minimum control. The black lines show a few optimal control sequences taken the car starting from various states in the state-space. Notice how the optimal control trajectory can be quite different even if the car starts from nearby states (-0.5,1) and (-0.4,1.2)). This is also quite typical of difficult dynamic programming problems.

6.2.3 Continuous-time LQR

Consider a linear continuous-time dynamical system given by

$$\dot{x} = A\,x + B\,u; \quad x(0) = x_0.$$

In the LQR problem, we are interested in finding a control trajectory that minimizes, as usual, a cost function that is quadratic in states and controls, except that we have an integral of the run-time cost because our system is a continuous-time system

$$\frac{1}{2}\,x(T)^\top Q_f\,x(T) + \frac{1}{2}\int_0^T x(t)^\top Q\,x(t) + u(t)^\top R\,u(t)\;\mathrm{d}t\,.$$

This is a very nice setup for using the HJB equation from the previous section.

Let us use our intuition from the discrete-time LQR problem and say that the optimal cost is quadratic in the states, namely,

$$J^*(x,t) = \frac{1}{2}x(t)^\top P(t)\,x(t);$$

notice that as usual the optimal cost-to-go is a function of the states x

and the time t because is the optimal cost of the continuous-time LQR problem if the system starts at a state x at time t and goes on until time $T \geq t$. We will now check if this J^* satisfies the HJB equation (we don't write the arguments $x(t)$, $u(t)$ etc. to keep the notation clear)

$$-\partial_t J^*(x,t) = \min_{u \in U} \left\{ \frac{1}{2} \left(x^\top Q x + u^\top R\, u \right) + (A\, x + B\, u)^\top\, \partial_x J^*(x,t) \right\} \tag{6.9}$$

from (6.7). The minimization is over the control input that we take *at time* t. Also notice the partial derivatives

$$\begin{aligned} \partial_x J^*(x,t) &= P(t)\, x. \\ \partial_t J^*(x,t) &= \frac{1}{2} x^\top \dot{P}(t)\, x. \end{aligned}$$

It is convenient in this case to see that the minimization can be performed using basic calculus (just like the discrete-time LQR problem), we differentiate with respect to u and set it to zero.

$$\begin{aligned} 0 &= \frac{\mathrm{d}\text{RHS of HJB}}{\mathrm{d}u} \\ \Rightarrow u^*(t) &= -R^{-1}\, B^\top P(t)\, x(t) \\ &\equiv -K(t)\, x(t). \end{aligned} \tag{6.10}$$

where $K(t) = R^{-1}\, B^\top P(t)$ is the Kalman gain. The controller is again linear in the states $x(t)$ and the expression for the gain is very simple in this case, much simpler than discrete-time LQR. Since $R \succ 0$, we also know that $u^*(t)$ computed here is the global minimum. If we substitute this value of $u^*(t)$ back into the HJB equation we have

$$\{\} \;\Big|_{u^*(t)} = \frac{1}{2} x^\top \left\{ PA + A^\top P + Q - PBR^{-1}B^\top P \right\}\, x.$$

If order to satisfy the HJB equation, we must have that the expression above is equal to $-\partial_t J^*(x,t)$. We therefore have, what is called the Continuous-time Algebraic Riccati Equation (CARE), for the matrix $P(t) \in \mathbb{R}^{d \times d}$

$$-\dot{P} = PA + A^\top P + Q - PBR^{-1}B^\top P. \tag{6.11}$$

This is an ordinary differential equation for the matrix P. The derivative $\dot{P} = \frac{\mathrm{d}P}{\mathrm{d}t}$ stands for differentiating every entry of P individually with time t. The terminal cost is $\frac{1}{2} x(T)^\top Q_f\, x(T)$ which gives the boundary condition for the ODE as

$$P(T) = Q_f.$$

Notice that the ODE for the $P(t)$ travels backwards in time.

Continuous-time LQR has particularly easy equations, as you can see in (6.10) and (6.11) compared to those for discrete-time ((6.4) and (6.5)). Special techniques have been invented for solving the Riccati equation. I

used the function scipy.linalg.solve_continuous_are to obtain Figure 6.3 using the continuous-time equations; the corresponding function for solving Discrete-time Algebraic Riccati Equation (DARE) which is given in (6.5) is scipy.linalg.solve_discrete_are. The continuous-time point-of-view also gives powerful connections to the Kalman filter, where you can show that the Kalman filter and LQR are duals of each other: in fact the equations for the Kalman filter (in continuous-time) and continuous-time LQR turn out to be exactly the same after you interchange appropriate quantities (!).

Infinite-horizon LQR Just like the infinite-horizon HJB equation has $\partial_t J^*(x,t) = 0$, if we have an infinite-horizon LQR problem, the cost matrix P should not be a function of time

$$\dot{P} = 0.$$

The continuous-time algebraic Riccati equation in (6.11) now becomes

$$PA + A^\top P + Q - PBR^{-1}B^\top P.$$

with the cost-to-go being given by $J^*(x) = \frac{1}{2}x^\top Px$.

6.3 Stochastic LQR

We will next look at a very powerful result. Say we have a stochastic linear dynamical system

$$\dot{x}(t) = Ax(t) + Bu(t) + B_\epsilon \epsilon(t);\ x(0) \text{ is given}$$

where $\epsilon(t)$ is standard Gaussian noise $\epsilon(t) \sim N(0, I)$ that is uncorrelated in time and would like to find a control sequence $\{u(t) : t \in [0,T]\}$ that minimizes a quadratic run-time and terminal cost

$$\mathop{\mathrm{E}}_{\epsilon(t):t\in[0,T]}\left[\frac{1}{2}x(T)^\top Q_f x(T) + \frac{1}{2}\int_0^T x(t)^\top Qx(t) + u(t)^\top Ru(t)\ \mathrm{d}t\right].$$

over a finite-horizon T. Notice that since the system is stochastic now, we should minimize the expected value of the cost over all possible realizations of the noise $\{\epsilon(t) : t \in [0,T]\}$. This is a very challenging problem, conceptually it is the equivalent of dynamic programming for an MDP with an infinite number of states $x(t) \in \mathbb{R}^d$ and an infinite number of controls $u(t) \in \mathbb{R}^m$.

However, it turns out that the optimal controller that we should pick in this case is also given by the standard LQR problem

$$\begin{aligned} u^*(t) &= -R^{-1}B^\top P(t)\ x(t) \\ \text{with } -\dot{P} &= PA + A^\top P + Q - PBR^{-1}B^\top P;\ P(T) = Q_f. \end{aligned}$$

We will not do the proof (it is easy but tedious, you can try to show it

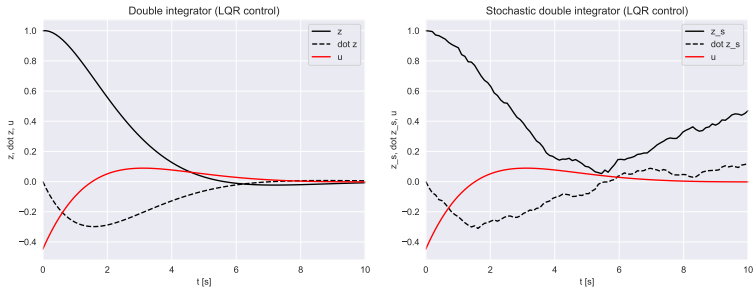


Figure 6.6: Comparison of the state trajectories of deterministic LQR and stochastic LQR problem with $B_\epsilon = [0.1, 0.1]$. The left panel is the same as that in Figure 6.3. The control input is the same in both cases but notice that the states in the plot on the right need not converge to the equilibrium due to noise. The cost of the trajectory will also be higher for the stochastic LQR case due to this. The total cost is $J^*(x_0) = 32.5$ for the deterministic case (32.24 for the quadratic state-cost and 0.26 for the control cost). The total cost $J^*(x_0)$ is much higher for the stochastic case, it is 81.62 (81.36 for the quadratic state cost and 0.26 for the control cost).

by writing the HJB equation for the stochastic LQR problem). This is a very surprising result because it says that even if the dynamical system had noise, the optimal control we should pick is exactly the same as the control we would have picked had the system been deterministic. It is a special property of the LQR problem and not true for other dynamical systems (nonlinear ones, or ones with non-Gaussian noise) or other costs.

We know that the control $u^*(t)$ is the same as the deterministic case. Is the cost-to-go $J^*(x,t)$ also the same? If you think about this, the cost-to-go in the stochastic case has to be a bit larger than the deterministic case because the noise $\epsilon(t)$ is always going to non-zero when we run the system, the LQR cost $J^*(x_0, 0) = \frac{1}{2} x_0^\top P(0) x_0$ is, after all, only the cost of the deterministic problem. It turns out that the cost for the stochastic LQR case for an initial state x_0 is

$$\begin{aligned} J^*(x_0, 0) &= \mathop{\mathrm{E}}_{\epsilon(t):t\in[0,T]} \left[\frac{1}{2} x(T)^\top Q_f x(T) + \frac{1}{2} \int_0^T \ldots \, \mathrm{d}t \right] \\ &= \frac{1}{2} x_0^\top P(0) x_0 + \frac{1}{2} \int_0^T \mathrm{tr}(P(t) B_\epsilon B_\epsilon^\top) \, \mathrm{d}t \,. \end{aligned}$$

The first term is the same as that of the deterministic LQR problem. The second term is the penalty we incur for having a stochastic dynamical system. This is the minimal cost achievable for stochastic LQR but it is not the same as that of the deterministic LQR.

6.4 Linear Quadratic Gaussian (LQG)

Our development in the previous sections and the previous chapter was based on a Markov Decision Process, i.e., we know the state $x(t)$ at each instant in time t even if this state $x(t)$ changes stochastically. We said that the optimal control for the linear dynamics is still $u^*(t) = -K(t)\, x(t)$.

What should one do if we cannot observe the state exactly?

Imagine a "continuous-time" form the observation equation in the Kalman filter where we receive observations of the form

$$y(t) = Cx(t) + D\nu.$$

where $\nu \sim N(0, I)$ is standard Gaussian noise that corrupts our observations y. If we extrapolate the definitions of the Kalman filter mean and covariance to this continuous-time setting, we can write the KF as follows. We know that the Kalman filter is the optimal estimate of the state given all past observations, so it computes

$$\mu(t) = \mathop{\mathrm{E}}_{\epsilon(s),\nu(s):\ s\in[0,t]} \left[x(t) \mid y(s) : s \in [0,t]\right].$$

There exists a "continuous-time version" of the Kalman filter (which was actually invented first), called the Kalman-Bucy filter. If the covariance of the estimate is

$$\Sigma(t) = \mathop{\mathrm{E}}_{\epsilon(s),\nu(s):\ s\in[0,t]} \left[x(t)\, x(t)^\top \mid y(s) : s \in [0,t]\right],$$

the Kalman-Bucy filter updates $\mu(t), \Sigma(t)$ using the differential equation

$$\begin{aligned}
\frac{\mathrm{d}}{\mathrm{d}t}\mu(t) &= Ax(t) + Bu(t) + K(t)\left(y(t) - C\mu(t)\right) \\
\frac{\mathrm{d}}{\mathrm{d}t}\Sigma(t) &= A\Sigma(t) + \Sigma(t)A^\top + B_\epsilon B_\epsilon^\top - K(t)DD^\top K(t)^\top \qquad (6.12)\\
\text{where } K(t) &= \Sigma(t)\, C^\top (DD^\top)^{-1}.
\end{aligned}$$

This equation is very close to the Kalman filter equations you saw in Chapter 3. In particular, notice the close similarity of the expression for the Kalman gain $K(t)$ with the Kalman gain of the LQR problem. You can read more at https://en.wikipedia.org/wiki/Kalman_filter.

Linear Quadratic Gaussian (LQG) It turns out that we can plug

As we discussed while introducing stochastic dynamical systems, there are various mathematical technicalities associated with conditioning on a continuous-time signal $\{y(s) : s \in [0,t]\}$. To be precise mathematicians define what is called a "filtration" $\mathcal{Y}(t)$ which is the union of the Borel σ-fields constructed using increasing subsets of the set $\{y(s) : s \in [0,t]\}$. Let us not worry about this here.

in the Kalman filter estimate $\mu(t)$ of the state $x(t)$ in order to compute optimal control for LQR if we know the state only through observations $y(t)$

$$u^*(t) = -K(t)\,\mu(t). \tag{6.13}$$

It is almost as if, we can blindly run a Kalman Filter in parallel with the deterministic LQR controller and get the optimal control for the stochastic LQR problem even if we did not observe the state of the system exactly. This method is called Linear Quadratic Gaussian (LQG).

This is a very powerful and surprising result. It is only true for linear dynamical systems with linear observations, Gaussian noise in both the dynamics and the observations and quadratic run-time and terminal costs. It is not true in other cases. However, it is so elegant and useful that it inspires essentially all other methods that control a dynamical system using observations from sensors.

Certainty equivalence For instance, even if we are using a particle filter to estimate the state of the system, we usually use the mean of the state estimate at time t given by $\mu(t)$ "as if" it were the true state of the system. Even if we were using some other feedback control $u(x)$ different than the LQR control (say feedback linearization), we usually plug in this estimate $\mu(t)$ in place of $x(t)$. Doing so is called "certainty equivalence" in control theory/robotics, which is a word borrowed from finance where one takes decisions (controls) directly using the estimate of the state (say stock price) while fully knowing the the stock price will change in the future stochastically.

6.4.1 (Optional material) The duality between the Kalman Filter and LQR

We can re-write the covariance in (6.12) using the identity

$$\frac{\mathrm{d}}{\mathrm{d}t}\left(\Sigma(t)^{-1}\right) = \Sigma(t)^{-1}\dot{\Sigma}(t)\Sigma(t)^{-1}$$

to get

$$\dot{S} = C^\top \left(DD^\top\right)^{-1} C - A^\top S - SA - SB_w B_w^\top S \tag{6.14}$$

where we have defined $S := \Sigma^{-1}$.

Notice that the two equations, updates to the LQR cost matrix in (6.11)

$$-\dot{P} = PA + A^\top P + Q - PBR^{-1}B^\top P$$

look quite similar to this equation. In fact, they are identical and you can substitute the following.

LQR	Kalman-Bucy filter
P	Σ^{-1}
A	$-A$
$BR^{-1}B$	$B_w B_w^\top$
Q	$C^\top \left(DD^\top\right)^{-1} C$
t	$T-t$

Let us analyze this equivalence. Notice that the inverse of the Kalman filter covariance is like the cost matrix of LQR. This is conceptually easy to understand, our figure of merit for filtering is the inverse covariance matrix (smaller the better) and our figure of merit for the LQR problem is the cost matrix P (smaller the better). Similarly, smaller the LQR cost, better the controller. The "dynamics" of the Kalman filter is the reverse of the dynamics of the LQR problem, this shows that the P matrix is updated backwards in time while the covariance Σ is updated forwards in time. The next identity

$$BR^{-1}B^\top = B_w B_w^\top$$

is very interesting. Imagine a situation where we have a fully-actuated system with $B = I$ and B_w being a diagonal matrix. This identity suggests that larger the control cost R_{ii} of a particular actuator i, lower is the noise of using that actuator $(B_w)_{ii}$, and vice-versa. This is how muscles in your body have evolved: muscles that are cheap to use (low R) are also very noisy in what they do whereas muscles that are expensive to use (large R) which are typically the biggest muscles in the body are also the least noisy and most precise. You can read more about this in the paper titled "General duality between optimal control and estimation" by Emanuel Todorov. The next identity

$$Q = C^\top \left(DD^\top\right)^{-1} C$$

is related to the quadratic state-cost in LQR. Imagine the situation where both Q, D are diagonal matrices. If the noise in the measurements D_{ii} is large, this is equivalent to the state-cost matrix Q_{ii} being small; roughly there is no way we can achieve a low state-cost $x^\top Q x$ in our system that consists of LQR and a Kalman filter (this combination is known as Linear Quadratic Gaussian LQG as saw before) if there is lots of noise in the state measurements. The final identity

$$t = T - t$$

is the observation that we have made many times before: dynamic programming travels backwards in time and the Kalman filter travels forwards in time.

6.5 Iterative LQR (iLQR)

This section is analogous to the section on the Extended Kalman Filter. We will study how to solve optimal control problems for a nonlinear

dynamical system

$$\dot{x} = f(x, u);\ x(0) = x_0 \text{ is given.}$$

We will consider a deterministic continuous-time dynamical system, the modifications to following section that one would make if the system is discrete-time, or stochastic, are straightforward and follow the same strategy. First consider the problem where the run-time and terminal costs are quadratic

$$\frac{1}{2}x(T)^\top Q_f x(T) + \frac{1}{2}\int_0^T x(t)^\top Q\, x(t) + u(t)^\top R u(t)\ \mathrm{d}t\,.$$

Receding horizon control and Model Predictive Control (MPC) One easy way to solve the dynamic programming problem, i.e., find a control trajectory of the *nonlinear* system that minimizes this cost functional, approximately, is by linearizing the system about the initial state x_0 and some reference control u_0 (this can usually be zero). Let the linear system be

$$\dot{z} = A_{x_0,u_0} z + B_{x_0,u_0} v;\ z(0) = 0; \tag{6.15}$$

where $A_{x_0,u_0} = \frac{\mathrm{d}f}{\mathrm{d}x}\big|_{x=x_0,u=u_0}$ and $B_{x_0,u_0} = \frac{\mathrm{d}f}{\mathrm{d}u}\big|_{x=x_0,u=u_0}$ are the Jacobians of the nonlinear function $f(x, u)$ with respect to the state and control respectively. The state of the linearized dynamics is

$$z := x - x_0, \text{ and } v := u - u_0,$$

We have emphasized the fact that the matrices A_{x_0,u_0}, B_{x_0,u_0} depend upon the reference state and control using the subscript. Given the above linear system, we can find a control sequence $u^*(\cdot)$ that minimizes the cost functional using the standard LQR formulation. Notice now that even we computed this control trajectory using the approximate linear system, it can certainly be *executed* on the nonlinear system, i.e., at run-time we will simply set $u \equiv u^*(z)$.

The linearized dynamics in (6.15) is potentially going to be very different from the nonlinear system. The two are close in the neighborhood of x_0 (and u_0) but as the system evolves using our control input to move further away from x_0, the linearized model no longer is a faithful approximation of the nonlinear model. A reasonable way to fix matters is to linearize about another point, say the state and control after $t = 1$ seconds, x_1, u_1 to get a new system

$$\dot{z} = A_{x_1,u_1} z + B_{x_1,u_1} v;\ z(0) = 0$$

and take the LQR-optimal control corresponding to this system for the next second.

The above methodology is called "receding horizon control". The idea is that we compute the optimal control trajectory $u^*(\cdot)$ using an approximation of the original system and recompute this control every few seconds when our approximation is unlikely to be accurate. This is a very

popular technique to implement optimal controllers in typical applications. The concept of using an approximate model (almost invariably, a linear model with LQR cost) to plan for the near-term future and resolving the problem in receding horizon fashion once the system is at the end of this short time-horizon is called "Model Predictive Control".

MPC is, perhaps, the second most common control algorithm implemented in the world. It is responsible for running most complex engineering systems that you can think of—power grids, oil refineries, chemical plants, rockets, aircrafts etc. Essentially, one never implements LQR directly, it is always implemented inside an MPC. For instance, in autonomous driving, the trajectory that the vehicle plans for traveling between two points A and B depends upon the current locations of the other cars/pedestrians in its vicinity, and potentially some prediction model of where they will be in the future. As the vehicle starts moving along this trajectory, the rest of the world evolves around it and we recompute the optimal trajectory to take into account the actual locations of the cars/pedestrians in the future.

Can you guess what is *the* most common control algorithm in the world?

6.5.1 Iterative LQR (iLQR)

Now let us consider the situation when in addition to a nonlinear system,

$$\dot{x} = f(x, u);\ x(0) = x_0,$$

the run-time and terminal cost is also nonlinear

$$q_f(x(T)) + \int_0^T q(x(t), u(t))\ \mathrm{d}t\,.$$

We can solve the dynamic programming problem in this case approximately using the following iterative algorithm.

Assume that we are given an initial control trajectory $u^{(0)}(\cdot) = \left\{u^{(0)}(t) : t \in [0, T]\right\}$. Let $x^{(0)}(\cdot)$ be the state trajectory that corresponds to taking this control on the nonlinear system, with of course $x^{(0)}(0) = x_0$. At each iteration k, the Iterative LQR algorithm performs the following steps.

Step 1 Linearize the nonlinear system about the state trajectory $x^{(k)}(\cdot)$ and $u^{(k)}(\cdot)$ using

$$z(t) := x(t) - x^{(k)}(t), \text{ and } v(t) := u(t) - u^{(k)}(t)$$

to get a new system

$$\dot{z} = A^{(k)}(t)z + B^{(k)}(t)v; z(0) = 0$$

where

$$A^{(k)}(t) = \frac{\mathrm{d}f}{\mathrm{d}x}\Big|_{x(t)=x^{(k)}(t), u(t)=u^{(k)}(t)}$$
$$B^{(k)}(t) = \frac{\mathrm{d}f}{\mathrm{d}u}\Big|_{x(t)=x^{(k)}(t), u(t)=u^{(k)}(t)}$$

and compute the Taylor series approximation of the nonlinear cost up to the second order

$$q_f(x(T)) \approx \text{constant} + z(T)^\top \frac{\mathrm{d}q_f}{\mathrm{d}x}\Big|_{x(T)=x^{(k)}(T)} + z(t)^\top \frac{\mathrm{d}^2 q_f}{\mathrm{d}x^2}\Big|_{x(T)=x^{(k)}(T)} z(t),$$

$$q(x,u,t) \approx \text{constant} + z(t)^\top \underbrace{\frac{\mathrm{d}q}{\mathrm{d}x}\Big|_{x(t)=x^{(k)}(t),u(t)=u^{(k)}(t)}}_{\text{affine term}} + v(t)^\top \underbrace{\frac{\mathrm{d}q}{\mathrm{d}u}\Big|_{x(t)=x^{(k)}(t),u(t)=u^{(k)}(t)}}_{\text{affine term}} + z(t)^\top \underbrace{\frac{\mathrm{d}^2 q}{\mathrm{d}x^2}\Big|_{x(t)=x^{(k)}(t),u(t)=u^{(k)}(t)}}_{\equiv Q} z(t) + v(t)^\top \underbrace{\frac{\mathrm{d}^2 q}{\mathrm{d}u^2}\Big|_{x(t)=x^{(k)}(t),u(t)=u^{(k)}(t)}}_{\equiv R} v(t).$$

This is an LQR problem with run-time cost that depends on time (like our discrete-time LQR formulation, the continuous-time formulation simply has Q, R to be functions of time t in the Riccati equation) and which also has terms that are affine in the state and control in addition to the usual quadratic cost terms.

Step 2 Solve the above linearized problem using standard LQR formulation to get the new control trajectory

$$u^{(k+1)}(t) := u^{(k)}(t) - Kz(t).$$

Simulate the *nonlinear* system using the control $u^{(k+1)}(\cdot)$ to get the new state trajectory $x^{(k+1)}(\cdot)$.

Some important comments to remember about the iLQR algorithm.

1. There are many ways to pick the initial control trajectory $u^{(0)}(\cdot)$, e.g., using a spline to get an arbitrary control sequence, using a spline to interpolate the states to get a trajectory $x^{(0)}(\cdot)$ and then back-calculate the control trajectory, using the LQR solution based on the linearization about the initial state, feedback linearization/differential flatness (https://en.wikipedia.org/wiki/Feedback_linearization) etc.

2. The iLQR algorithm is an approximate solution to dynamic programming for nonlinear system with general, nonlinear run-time and terminal costs. This is because the the algorithm uses a linearization about the previous state and control trajectory to compute the new

❓ How will you solve for the optimal controller for a linear dynamics for the cost

$$\int_0^T \left(q^\top x + \frac{1}{2} x^\top Q x\right) \mathrm{d}t\,,$$

i.e., when in addition the quadratic cost, we also have an affine term?

control trajectory. iLQR is not guaranteed to find the optimal solution of dynamic programming, although in practice with good implementations, it works excellently.

3. We can think of iLQR as an algorithm to track a given state trajectory $x^g(t)$ by setting

$$q_f = 0, \text{and } q(x,u) = \|x^g(t) - x(t)\|^2.$$

This is often how iLQR is typically used in practice, e.g., to make an autonomous race car closely follow the racing line (see the paper "BayesRace: Learning to race autonomously using prior experience" https://arxiv.org/abs/2005.04755 and https://www.youtube.com/watch?v=dgIpf0Lg8Ek for a clever application of using MPC to track a challenging race line), or to make a drone follow a given desired trajectory (https://www.youtube.com/watch?v=QREeZvHg0lQ).

Differential Dynamic Programming (DDP) is a suite of techniques that is a more powerful version of iterated LQR. Instead of linearizing the dynamics and taking a second order Taylor approximation of the cost, DDP takes a second order approximation of the Bellman equation directly. The two are not the same; DDP is the more correct version of iLQR but is much more challenging computationally.

Broadly speaking, iLQR and DDP are used to perform control for some of the most sophisticated robots today, you can see an interesting discussion of the trajectory planning of some of the DARPA Humanoid Robotics Challenge at https://www.cs.cmu.edu/~cga/drc/atlas-control. Techniques like feedback linearization work excellently for drones where we do not really care for optimal cost (see "Minimum snap trajectory generation and control for quadrotors" https://ieeexplore.ieee.org/document/5980409) while LQR and its variants are still heavily utilized for satellites in space.

Chapter 7

Imitation Learning

Reading

1. The DAGGER algorithm (https://www.cs.cmu.edu/~sross1/publications/Ross-AIStats11-NoRegret.pdf)
2. https://www.youtube.com/watch?v=TUBBIgtQL_k
3. An Algorithmic Perspective on Imitation Learning (https://arxiv.org/pdf/1811.06711.pdf)

This is the beginning of Module 3 of the course. The previous two modules have been about how to to estimate the state of the world around the robot (Module 1) and how to move the robot (or the world) to a desired state (Module 2). Both of these required that we maintain a model of the dynamics of the robot; this model may be inaccurate and we fudged over this inaccuracy by modeling the remainder as "noise" in Markov Decision Processes.

The next few lectures introduce different aspects of what is called Reinforcement Learning (RL). This is a very large field and you can think of using techniques from RL in many different ways.

1. **Dynamic programming with function approximation.** If we are solving a dynamic programming problem, we can think of writing down the optimal cost-to-go $J^*(x,t)$ as a function of some parameters, e.g., the cost-to-go is

$$J_\varphi(x,t) = \frac{1}{2}x(t)^\top \underbrace{(\text{some function of } A,B,Q,R)}_{\text{function of } \varphi} x(t)$$

for LQR. We know the stuff inside the brackets to be exactly $P(t)$ but, if we did not, it could be written down as some generic function

of parameters φ. We know that any cost-to-go that satisfies the Bellman equation is the optimal cost-to-go, so we can now "fit" the candidate function $J_\varphi(x,t)$ to satisfy the Bellman equation. Similarly, one may also express the optimal feedback control $u(\cdot)$ using some parameters θ as

$$u_\theta(\cdot).$$

We will see how to fit such functions in this chapter.

2. **Learning from data.** It may happen that we do not know very much about the dynamical system, e.g., we do not know a good model for what drives customers as they buy items in an online merchandise platform, or a robot traveling in a crowded area may not have a good model for how large crowds of people walk around it. One may collect data from these systems fit some model of the form $\dot{x} = f(x,u)$ to the data and then go back to the techniques of Module 2. It is typically not clear how much data one should collect. RL gives a suite of techniques to learn the cost-to-go in these situations by collecting and assimilating the data *automatically*. These techniques go under the umbrella of policy gradients, on-policy methods etc. One may also simply "memorize" the data provided by an expert operator, this is called Imitation Learning and we will discuss it next.

Some motivation Imitation Learning is also called "learning from demonstrations". This is in fact one of the earliest successful examples of using a neural network for driving. The ALVINN project at CMU by Dean Pomerleau in 1988 (https://www.youtube.com/watch?v=2KMAAmkz9go) used a two-layer neural network with 5 hidden neurons, about 1000 inputs from the pixels of a camera and 30 outputs. It successfully drove in different parts of the United States and Germany. Imitation learning has also been responsible for numerous other early-successes of RL, e.g., acrobatic maneuvers on an RC helicopter (http://ai.stanford.edu/ acoates/-papers/AbbeelCoatesNg_IJRR2010.pdf).

Imitation Learning seeks to record data from experts, e.g., humans,

and reproduce these desired behaviors on robots. The key questions we should ask, and which we will answer in this chapter, are as follows.

1. Who should demonstrate (experts, amateurs, or novices) and how should we record data (what states, controls etc.)?
2. How should we learn from this data? e.g., fit a supervised regression model for the policy. How should one ignore bad behaviors in non-expert data?
3. And most importantly, what can we do if the robot encounters a situation which was not in the dataset.

7.1 A crash course in supervised learning

Nature gives us data X and targets Y for this data.

$$X \to Y.$$

Nature does not usually tell us what property of a datum $x \in X$ results in a particular prediction $y \in Y$. We would like to learn to imitate Nature, namely predict y given x.

What does such learning mean? It is simply a notion of being able to identify patterns in the input data without explicitly programming a computer for prediction. We are often happy with a learning process that identifies correlations: if we learn correlations on a few samples $(x^1, y^1), \ldots, (x^n, y^n)$, we may be able to predict the output for a new datum x^{n+1}. We may not need to know *why* the label of x^{n+1} was predicted to be so and so.

Let us say that Nature possesses a probability distribution P over (X, Y). We will formalize the problem of machine learning as Nature drawing n independent and identically distributed samples from this distribution. This is denoted by

$$D_{\text{train}} = \left\{(x^i, y^i) \sim P\right\}_{i=1}^{n}$$

is called the "training set". We use this data to identify patterns that help make predictions on some future data.

What is the task in machine learning? Suppose D_{train} consists of $n = 50$ RGB images of size 100×100 of two kinds, ones with an orange inside them and ones without. 10^4 is a large number of pixels, each pixel taking any of the possible 255^3 values. Suppose we discover that one particular pixel, say at location $(25, 45)$, takes distinct values in all images inside our training set. We can then construct a predictor based on this pixel. This predictor, it is a binary classifier, perfectly maps the training

❓ How many such binary classifiers are there at most?

images to their labels (orange: +1 or no orange: -1). If x_{ij}^k is the $(ij)^{\text{th}}$ pixel for image x^k, then we use the function

$$f(x) = \begin{cases} y^k & \text{if } x_{ij}^k = x_{ij} \text{ for some } k = 1, \ldots, n \\ -1 & \text{otherwise.} \end{cases}$$

This predictor certainly solves the task. It correctly works for all images in the training set. Does it work for images outside the training set?

Our task in machine learning is to learn a predictor that works *outside* the training set. The training set is only a source of information that Nature gives us to find such a predictor.

> Designing a predictor that is accurate on D_{train} is trivial. A hash function that memorizes the data is sufficient. This is NOT our task in machine learning. We want predictors that generalize to new data outside D_{train}.

Generalization If we never see data from outside D_{train} why should we hope to do well on it? The key is the distribution P. Machine learning is formalized as constructing a predictor that works well on new data that is also drawn independently from the distribution P. We will call this set of data the "test set"

$$D_{\text{test}}$$

and it is constructed similarly. This assumption is important. It provides coherence between past and future samples: past samples that were used to train and future samples that we will wish to predict upon. How to find such predictors that work well on new data? The central idea in machine learning is to restrict the set of possible binary functions that we consider.

> We are searching for a predictor that generalizes well but only have the training data to select predictors.

The *right* class of functions f cannot be too large, otherwise we will find our binary classifier above as the solution, and that is not very useful. The class of functions cannot be too small either, otherwise we won't be able to predict difficult images. If the predictor does not even work well on the training set, there is no reason why we should expect it to work on the test set.

> Finding this correct class of functions with the right balance is what machine learning is all about.

❓ Can you now think how is machine learning different from other fields you might know such as statistics or optimization?

7.1.1 Fitting a machine learning model

Let us now solve a classification problem. We will again go around the model selection problem and consider the class of linear classifiers. Assume binary labels $Y \in \{-1, 1\}$. To keep the notation clear, we will use the trick of appending a 1 to the data x and hide the bias term b in the linear classifier. The predictor is now given by

$$\begin{aligned} f(x; w) &= \text{sign}(w^\top x) \\ &= \begin{cases} +1 & \text{if } w^\top x \geq 0 \\ -1 & \text{else.} \end{cases} \end{aligned} \tag{7.1}$$

We have used the sign function denoted as sign to get binary $\{-1, +1\}$ outputs form our real-valued prediction $w^\top x$. This is the famous perceptron model of Frank Rosenblatt.

We want the predictions of the model to match those in the training data and devise an objective to to fit/train the perceptron.

$$\ell_{\text{zero-one}}(w) := \frac{1}{n} \sum_{i=1}^{n} \mathbf{1}_{\{y^i \neq f(x^i; w)\}}. \tag{7.2}$$

The indicator function inside the summation measures the number of mistakes the perceptron makes on the training dataset. The objective here is designed to find weights w that minimizes the average number of mistakes, also known as the training error. Such a loss that measures the mistakes is called the zero-one loss, it incurs a penalty of 1 for a mistake and zero otherwise.

❓ Can you think of some quantity other than the zero-one error that we may wish to optimize?

Surrogate losses The zero-one loss is the clearest indication of whether the perceptron is working well. It is however non-differentiable, so we cannot use powerful ideas from optimization theory to minimize it. This is why surrogate losses are constructed in machine learning. These are proxies for the loss function, typically for the classification problems and look as follows. The exponential loss is

$$\ell_{\text{exp}}(w) = e^{-y\ (w^\top x)}$$

or the logistic loss is

$$\ell_{\text{logistic}}(w) = \log\left(1 + e^{-y w^\top x}\right).$$

Stochastic Gradient Descent (SGD) SGD is a very general algorithm to optimize objectives typically found in machine learning. We can use it so long as we have a dataset and an objective that is differentiable. Consider an optimization problem where we want to solve for

$$w^* = \underset{w}{\text{argmin}}\ \frac{1}{n} \sum_{i=1}^{n} \ell^i(w)$$

where the function ℓ^i denotes the loss on the sample (x^i, y^i) and $w \in \mathbb{R}^p$ denotes the weights of the classifier. Solving this problem using SGD corresponds to iteratively updating the weights using

$$w^{(t+1)} = w^{(t)} - \eta \frac{\mathrm{d}\ell^{\omega_t}(w)}{\mathrm{d}w}\Big|_{w=w^{(t)}},$$

i.e., we compute the gradient one sample with index ω_t in the dataset. The index ω_t is chosen uniformly randomly from

$$\omega_t \in \{1, \ldots, n\}.$$

In practice, at each time-step t, we typically select a few (not just one) input data ω_t from the training dataset and average the gradient $\frac{\mathrm{d}\ell^{\omega_t}(w)}{\mathrm{d}w}\Big|_{w=w^{(t)}}$ across them; this is known as a "mini-batch". The gradient of the loss $\ell^{\omega_t}(w)$ with respect to w is denoted by

$$\nabla \ell^{\omega_t}(w^{(t)}) := \frac{\mathrm{d}\ell^{\omega_t}(w)}{\mathrm{d}w}\Big|_{w=w^{(t)}} = \begin{bmatrix} \nabla_{w_1}\ell^{\omega_t}(w^{(t)}) \\ \nabla_{w_2}\ell^{\omega_t}(w^{(t)}) \\ \vdots \\ \nabla_{w_p}\ell^{\omega_t}(w^{(t)}) \end{bmatrix} \in \mathbb{R}^p.$$

The gradient $\nabla \ell^{\omega_t}(w^{(t)})$ is therefore a vector in $\mathbb{R}^p$. We have written

$$\nabla_{w_1}\ell^{\omega_t}(w^{(t)}) = \frac{\mathrm{d}\ell^{\omega_t}(w)}{\mathrm{d}w_1}\Big|_{w=w^{(t)}}$$

for the scalar-valued derivative of the objective $\ell^{\omega_t}(w^{(t)})$ with respect to the first weight $w_1 \in \mathbb{R}$. We can therefore write SGD as

$$w^{(t+1)} = w^{(t)} - \eta \nabla \ell^{\omega_t}(w^{(t)}). \tag{7.3}$$

The non-negative scalar $\eta \in \mathbb{R}_+$ is called the step-size or the learning rate. It governs the distance traveled along the negative gradient $-\nabla \ell^{\omega_t}(w^{(t)})$ at each iteration.

7.1.2 Deep Neural Networks

The Perceptron in (7.1) is a linear model: it computes a linear function of the weights $w^\top x$ and uses this function to make the predictions $f(x; w) = \text{sign}(w^\top x)$. Linear models try to split the data (say we have binary labels $Y = \{-1, 1\}$) using a hyper-plane with w denoting the normal to this hyper-plane. This does not work for all situations of course, as the figure below shows, there is no hyper-plane that cleanly separates the two classes (i.e., achieves zero mis-prediction error) but there *is* a nonlinear function that can do the job.

A deep neural network is one such nonlinear function. First consider a "two-layer" network

$$f(x; v, S) = \text{sign}\left(v^\top \sigma\left(S^\top x\right)\right)$$

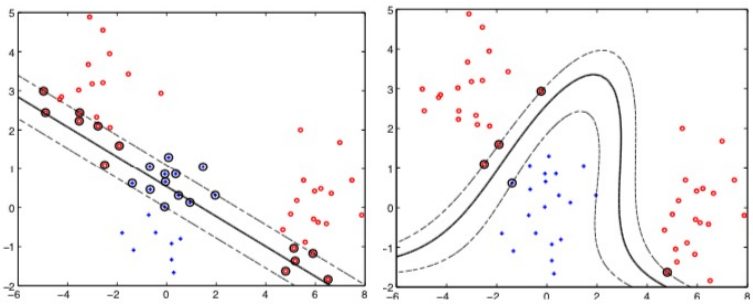


Figure 7.1

where the matrix $S \in \mathbb{R}^{d \times p}$ and a vector $v \in \mathbb{R}^p$ are the parameters or "weights" of the classifier. The "nonlinearity" σ is usually set to be what is called a Rectified Linear Unit (ReLU)

$$\begin{aligned} \sigma(x) := \text{ReLU}(x) &= |x|_+ \\ &= \max(0, x). \end{aligned} \tag{7.4}$$

Just like the case of a Perceptron, we can use an objective $\frac{1}{n}\sum_{i=1}^n \ell^i(v, S)$ that depends on both v, S to fit this classifier on training data. A deep neural network takes the idea of a two-layer network to the next step and has multiple "layers", each with a different weight matrix $S_1, \ldots, S_L$. The classifier is therefore given by

$$f(x; v, S_1, \ldots, S_L) = \text{sign}\left(v^\top \sigma\left(S_L^\top \ldots \sigma\left(S_2^\top\ \sigma(S_1^\top x)\right)\ldots\right)\right). \tag{7.5}$$

We call each operation of the form $\sigma\left(S_k^\top \ldots\right)$, as a *layer*. Consider the second layer: it takes the features generated by the first layer, namely $\sigma(S_1^\top x)$, multiplies these features using its feature matrix $S_2^\top$ and applies a nonlinear function $\sigma(\cdot)$ to this result element-wise before passing it on to the third layer.

> A deep network creates new features by composing older features.

This composition is very powerful. Not only do we not have to pick a particular feature vector, we can create very complex features by sequentially combining simpler ones. For example Figure 7.2 shows the features (more precisely, the kernel) learnt by a deep neural network. The first layer of features are called Gabor-like, and incidentally they are similar to the features learned by the human brain in the first part of the visual cortex (the one closest to the eyes). These features are *combined* linearly along with a nonlinear operation to give richer features (spirals, right angles) in the middle panel. The third layer combines the lower features to get even more complex features, these look like patterns (notice a soccer ball in the bottom left), a box on the bottom right etc.

Deep networks are universal function approximators The multi-layer neural network is a powerful class of classifiers: depending upon how many layers we have and what is the dimensionality of the the weight matrices S_k at each layer, we can fit *any* training data. In fact, this statement, which is called the *universal approximation property* holds even for a two-layer neural network $v^\top \sigma(S^\top x)$ if the number of columns in S is big enough. This property is the central reason why deep networks are so widely applicable, we can model complex machine learning problems if we choose a big enough deep network.

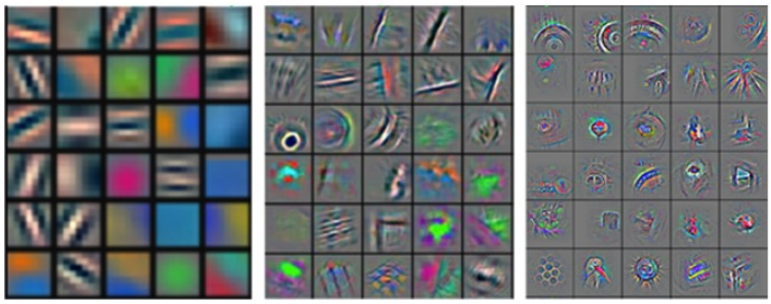


Figure 7.2

Logits for multi-class classification. The output

$$\hat{y} = v^\top \sigma\left(S_L^\top \ldots \sigma\left(S_2^\top\ \sigma(S_1^\top x)\right)\ldots\right)$$

is called the logits corresponding to the different classes. This name comes from logistic regression where logits are the log-probabilities of an input datum belonging to one of the two classes. A deep network provides an easy way to solve a multi-class classification problem, we simply set

$$v \in \mathbb{R}^{p\times C}$$

where C is the total number of classes in the data. Just like logistic regression predicts the logits of the two classes, we would like to *interpret* the vector $\hat{y}$ as the log-probabilities of an input belonging to one of the classes.

❓ What would the shape of w be if you were performing regression using a deep network?

Weights It is customary to not differentiate between the parameters of different layers of a deep network and simply say *weights* when we want to refer to all parameters. The set

$$w := \{v, S_1, S_2, \ldots, S_L\}$$

is the set of *weights*. This set is typically stored in PyTorch as a set of matrices, one for each layer. Using this new notation, we will write down a deep neural network classifier as simply

$$f(x, w) \tag{7.6}$$

and fitting the deep network to a dataset involves the optimization problem

$$w^* = \underset{w}{\text{argmin}} \ \frac{1}{n}\sum_{i=1}^{n} \ell(y^i, \hat{y}^i). \tag{7.7}$$

We will also sometimes denote the loss of the i^{th} sample as

$$\ell^i(w) := \ell(y^i, \hat{y}^i).$$

Backpropagation The Backpropagation algorithm is a method to compute the gradient of the objective while fitting a deep network using SGD, i.e., it computes $\nabla_w \ell^i(w)$. For the purposes of this course, the details of how this is done are not essential, so we will skip them. You can read more in the notes of ESE 546 at https://pratikac.github.io/pub/20_ese546.pdf.

PyTorch We will use a library called PyTorch (https://pytorch.org) to code up deep neural networks for the reinforcement learning part of this course. You can find some excellent tutorials for it at https://pytorch.org/tutorials/beginner/basics/intro.html. We have also uploaded two recitations from the Fall 2020 offering of ESE 546 on Canvas which guide you through various typical use-cases of PyTorch. You are advised to go through, at least, the first recitation if you are not familiar with PyTorch. For the purposes of this course, you do not need to know the intricacies of PyTorch, we will give you enough code to work with deep networks so that you can focus on implementing the reinforcement learning-specific parts.

7.2 Behavior Cloning

With that background, we are ready to tackle what is potentially the simplest problem in RL. We will almost exclusively deal with discrete-time systems for RL. Let us imagine that we are given access to n trajectories each of length $T+1$ time-steps from an expert demonstrator for our system. We write this as a training dataset

$$D = \left\{(x_t^i, u_t^i)_{t=0,1,\ldots,T}\right\}_{i=1,\ldots,n}$$

At each step, we record the state $x_t^i \in \mathbb{R}^d$ and the control that the expert took at that state u_t^i. We would like to learn a deterministic feedback control for the robot that is parametrized by parameters θ

$$u_\theta(x) : X \mapsto U \subset \mathbb{R}^m.$$

using the training data. The idea is that if $u_\theta(x^i(t)) \approx u^i(t)$ for all i and all times t, then we can simply run our learned controller $u_\theta(x)$ on the robot instead of having the expert. A simple example is a baby deer learning to imitate how its mother in how to run.

Parameterizing the controller Our function u_θ may represent many different families of controllers. For example, $u_\theta(x) = \theta x$ where $\theta \in \mathbb{R}^{d\times p}$ is a linear controller; this is much like the control for LQR except that we can fit θ to the expert's data instead of solving the LQR problem to find the Kalman gain. We could also think of some other complicated function, e.g., a two-layer neural network,

$$u_\theta(x) = v\,\sigma\left(S^\top x\right)$$

where $S \in \mathbb{R}^{d\times p}$ and $v \in \mathbb{R}^{m\times p}$ and $\sigma : \mathbb{R}^m \mapsto \mathbb{R}^m$ is some nonlinearity, say ReLU. As we did above, we will use

$$\theta := (v, S)$$

to denote all the weights of this two-layer neural network. Multi-layer neural networks are also another possible avenue. In general, we want to the parameterization of the controller to be rich enough to fit some complex controller that the expert may have used on the system.

How to fit the controller? Given our chosen model for $u_\theta(x)$, say a two-layer neural network with weights θ, fitting the controller involves finding the best value for the parameters θ such that $u_\theta(x_t^i) \approx u_t^i$ for data in our dataset. There are many ways to do this, e.g., we can solve the following optimization problem

$$\widehat{\theta} = \underset{\theta}{\operatorname{argmin}}\ \ell(\theta) := \frac{1}{n}\sum_{i=1}^{n} \underbrace{\frac{1}{T+1}\sum_{t=0}^{T} \left\|u_t^i - u_\theta(x_t^i)\right\|_2^2}_{\ell^i(\theta)} \tag{7.8}$$

The difficulty of solving the above problem depends upon how difficult the model $u_\theta(x)$ is, for instance, if the model is linear $\theta\ x$, we can solve (7.8) using ordinary least squares. If the model is a neural network, one would have to use SGD to solve the optimization problem above. After fitting this model, we have a new controller

$$u_{\widehat{\theta}}(x) \in \mathbb{R}^m$$

that we can use *anywhere* in the domain $X \subset \mathbb{R}^d$, even at places where we had no expert data. This is known as Behavior Cloning, i.e., cloning the controls of the expert into a parametric model.

Generalization performance of behavior cloning Note that the data provided by the expert is not iid, of course the state x_{t+1}^i in the expert's trajectory depends upon the previous state x_t^i. Standard supervised learning makes the assumption that Nature gives training data that is independent and identically distributed from the distribution P. While it is still reasonable to fit the regression loss in (7.8) for such correlated data, one should remember that if the expert trajectories do not go to all parts of the state-space, the learned controller fitted on the training data

may not work outside these parts. Of course, if we behavior clone the controls taken by a generic driver, they are unlikely to be competitive for racing, and vice-versa. It is very important to realize that this does *not* mean that BC does not generalize. Generalization in machine learning is a concept that suggests that the model should work well on data *from the same distribution*. What does the the "distribution" of the expert mean, in this case, it simply refers to the distribution of the states that the expert's trajectories typically visit, e.g, a race driver typically drives at the limits of tire friction and throttle, this is different from a usual city-driver who would rather maximize the longevity of their tires and engine-life.

ℹ Discuss generalization performance in behavior cloning.

7.2.1 Behavior cloning with a stochastic controller

In this case, we have always chosen feedback feedback controllers that are deterministic, i.e., there is a single value of control u that is taken at the state x. Going forward, we will also talk about stochastic controllers, i.e., controllers which sample a control from a distribution. There can be a few reasons of using such a controller. First, we will see in later lectures how this may help in training a reinforcement learning algorithm; this is because in situations where you do not know the system dynamics precisely, it helps to "hedge" the feedback to take a few different control actions instead of simply the one that the value function deems as the maximizing one. This is not very different from having a few different stocks in your portfolio. Second, we benefit from this hedging even at test-time when we run a stochastic feedback control, e.g., in situations where the limited training data may not want to always pick the best control (because the best control was computed using an imprecise model of the system dynamics and could be wrong), but rather hedge our bets by choosing between a few different controls.

A stochastic feedback control is denoted by

$$u \sim u_\theta(\cdot \mid x) = \mathrm{P}(\cdot \mid x)$$

notice that $u_\theta(\cdot \mid x)$ is a probability distribution on the control space U that depends on the state x, and in this case the parameters θ. The control taken at a state x is a sample drawn from this probability distribution. The deterministic controller is a special case of this setup where

$$u_\theta(u|\, x) = \delta_{u_\theta(x)}(u) \equiv u_\theta(x)$$

is a Dirac-delta distribution at $u_\theta(x)$. If the control space U is discrete, then $u_\theta(\cdot \mid x)$ could be a categorical distribution. If the control space U is continuous, then you may wish to think of the controls being sampled from a Gaussian distribution with some mean $\mu_\theta(x)$ and variance $\sigma^2_\theta(x)$

$$\mathbb{R}^m \ni u \sim u_\theta(\cdot \mid x) = N(\mu_\theta(x), \Sigma_\theta(x)).$$

Maximum likelihood estimation Let's pick a particular stochastic controller, say a Gaussian. How should we fit the parameters θ for this? We would like to find parameters θ that make the expert's data in our dataset very likely. The log-likelihood of each datum is

$$\log u_\theta(u_t^i \mid x_t^i)$$

and maximizing the log-likelihood of the entire dataset amounts to solving

$$\widehat{\theta} = \operatorname*{argmin}_\theta \frac{1}{n} \sum_{i=1}^{n} \underbrace{\frac{1}{T+1} \sum_{t=0}^{T} -\log u_\theta(u_t^i \mid x_t^i)}_{\ell^i(\theta)} . \tag{7.9}$$

Fitting BC with a Gaussian controller Notice that if we use a Gaussian distribution

$$u_\theta(\cdot \mid x) = N\left(\mu_\theta(x), I\right)$$

as our stochastic controller, the objective in (7.9) is the same as that in (7.8).

$$u_\theta(\cdot \mid x) = N\left(\mu_\theta(x), \sigma_\theta^2(x) I\right)$$

we have that

$$-\log u_\theta(u \mid x) = \frac{\|\mu_\theta(x) - u\|_2^2}{\sigma_\theta^2(x)} + 2cp \log \sigma_\theta(x).$$

where c is a constant.

7.2.2 KL-divergence form of Behavior Cloning

Background on KL divergence The Kullback-Leibler (KL) divergence is a quantity to measure the distance between two probability distributions. There are many similar distances, for example, given two probability distributions $p(x)$ and $q(x)$ supported on a discrete set X, the total variation distance between them is

$$\mathrm{TV}(p, q) = \frac{1}{2} \sum_{x \in X} |p(x) - q(x)| .$$

Hellinger distance (https://en.wikipedia.org/wiki/Hellinger_distance), f-divergences (https://en.wikipedia.org/wiki/F-divergence) and the Wasserstein metric (https://en.wikipedia.org/wiki/Wasserstein_metric) are a few other examples of ways to measure how different two probability distributions are from each other.

The Kullback-Leibler divergence (KL) between two distributions is given by

$$\mathrm{KL}(p \,||\, q) = \sum_{x \in X} p(x) \log \frac{p(x)}{q(x)}. \tag{7.10}$$

This is a distance and not a metric, i.e., it is always non-zero and zero

if and only if the two distributions are equal, but the KL-divergence is not symmetric (like a metric has to be). Also, the above formula is well-defined only if for all x where $q(x) = 0$, we also have $p(x) = 0$. Notice that it is not symmetric

$$\mathrm{KL}(q \,||\, p) = \sum_{x \in X} q(x) \log \frac{q(x)}{p(x)} \qquad \neq \mathrm{KL}(p \,||\, q).$$

The funny notation $\mathrm{KL}(p \,||\, q)$ was invented by Shun-ichi Amari (https://en.wikipedia.org/wiki/Shun%27ichi_Amari) to emphasize the fact that the KL-divergence is asymmetric. The KL-divergence is always positive: you can show this using an application of Jensen's inequality. For distributions with continuous support, we integrate over the entire space X and define KL divergence as

$$\mathrm{KL}(p \,||\, q) = \int_{\mathcal{X}} p(x) \log \frac{p(x)}{q(x)} \, \mathrm{d}x \,.$$

Behavior Cloning Let us now imagine the expert is also a parametric stochastic feedback controller $u_{\theta^*}(\cdot \mid x)$. Our data is therefore drawn by running this controller for n trajectories, T time-steps on the system. This dataset now consists of samples from

$$p_{u_{\theta^*}}(x, u)$$

which is the joint distribution on the state-space X and the control-space U. We have denoted the parameters of the feedback controller which creates this distribution as the subscript u_{θ^*}. Our behavior cloning controller creates a similar distribution $p_{u_\theta}(x, u)$ and the general version of the objective in (7.9) is therefore

$$\widehat{\theta} = \operatorname*{argmin}_{\theta} \mathrm{KL}\left(p_{u_{\theta^*}} \,||\, p_{u_\theta}\right); \tag{7.11}$$

The objective in (7.9) corresponds to this for Gaussian stochastic controllers, but we can just as easily imagine some other distribution for the stochastic controller of the expert and the robot.

> Written this way, BC can be understood as finding a controller $\widehat{\theta}$ whose distribution on the states and controls is close to the distribution of states and controls of the expert.

7.2.3 Some remarks on Behavior Cloning

Worst-case performance Performance of Behavior Cloning can be quite bad in the worst case. The authors in "Efficient reductions for imitation learning" (https://www.cs.cmu.edu/~sross1/publications/Ross-AIStats11-NoRegret.pdf) show that if the learned controller $u_{\widehat{\theta}}$ differs from the control taken by the expert controller u_{θ^*} with a probability ϵ at

each time-step, over a horizon of length T time-steps, it can be $\mathcal{O}(T^2\epsilon)$ off from the cost-to-go of the expert *as averaged over states that the learned controller visits*. This is because once the robot makes a mistake and goes away from the expert's part in the state-space, future states of the robot and the expert can be very different.

Model-free nature of BC Observe that our learned controller $u_{\widehat{\theta}}(\cdot \mid x)$ is a feedback controller and works for entire state-space X. We did not need to know the dynamics of the system to build this controller. The data from the expert is conceptually the same as the model $\dot{x} = f(x, u)$ of the dynamics, and you can learn controllers from both. Do you however notice a catch?

Draw a picture of the amplifying errors of running behavior cloning in real-time.

7.3 DAgger: Dataset Aggregation

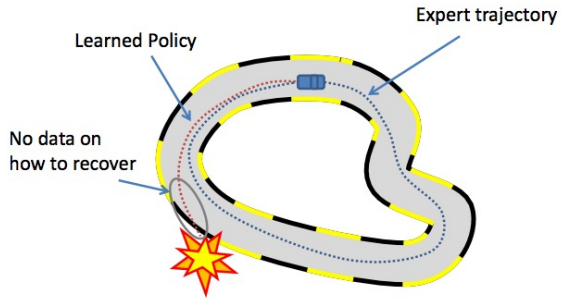
The expert's dataset in Behavior Cloning determines the quality of the controller learned. If we collected very few trajectories from the expert, they may not cover all parts of the state-space and the behavior cloned controller has no data to fit the model in those parts.

Let us design a simple algorithm, of the same spirit as iterative-LQR, to mitigate this. We start with a candidate controller, say $u_{\theta^{(0)}}(x)$; one may also start with a stochastic controller $u_{\theta^{(0)}}(\cdot \mid x)$ instead.

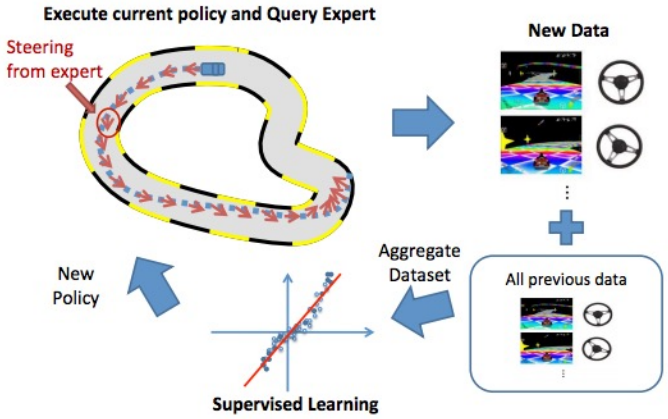
DAgger: Let the dataset $D^{(0)}$ be the data collected from the expert. Initialize $u_{\theta^{(0)}} = u_{\widehat{\theta}}$ to be the BC controller learned using data $D^{(0)}$. At iteration k

1. The robot queries the expert for a fraction p of the time-steps and uses its learned controller $u_{\theta^{(k-1)}}$ for the other time-steps. If the expert corresponds to some controller u_{θ^*}, then the robot controller at a state x is
$$u \sim p\, \delta_{u_{\theta^*}(x)} \; + (1-p)\, \delta_{u_{\theta^{(k-1)}}(x)}.$$
2. Use $u(x)$ to collect a dataset $D = \left\{(x_t^i, u_t^i)_{t=0,\dots,T}\right\}_{i=1,\dots,n}$ with n trajectories.
3. Set the new dataset to be $D^{(k)} = D^{(k-1)} \cup D$
4. Fit a controller $u_{\theta^{(k)}}$ using behavior cloning to the new dataset $D^{(k)}$.

The above algorithm iteratively updates the BC controller $u_{\widehat{\theta}}$ by drawing new data from the expert. The robot first bootstraps off the expert's data, this simply means that it uses the expert's data to fit its controller $u_{\theta^{(0)}}(x)$. As we discussed above, this controller may veer off the expert's trajectory if the robot starts at states that are different from the dataset, or even if it takes a slightly different control than the expert midway through a trajectory.



To fix this, the robot collects more data at each iteration. It uses a combination of the expert and its controller to collect such data. This, allows *collecting a dataset of expert's controls in states that the robot visits* and iteratively expands the dataset $D^{(k)}$.



In the beginning we may wish to be close to the expert's data and use a large value of p, as the fitted controller $u_{\theta_{k+1}}$ becomes good, we can reduce the value of p and rely less on the expert.

DAgger is an iterative algorithm which expands the controller to handle larger and larger parts of the state-space. Therefore, the cost-to-go of the controller learned via DAgger is $\mathcal{O}(T)$ off from the cost-to-go of the expert *as averaged over states that the learned controller visits.*

DAgger with expert annotations at each step DAgger is a conceptual framework where the expert is queried repeatedly for new control actions. This is obviously problematic because we need to expert on hand at each iteration. We can also cook up a slightly version of DAgger where we start with the BC controller $u_{\theta^{(k)}} = u_{\widehat{\theta}}$ and at each step, we run the controller on the real system and ask the expert to relabel the data after that run. The dataset $D^{(k)}$ collected by the algorithm expands at each iteration and although the states x_t^i are those visited by our controller, their annotations are those given by the expert. This is a much more natural way of implementing DAgger.

❓ What criterion can we use to stop these iterations? We can stop when the incremental dataset collected D_k is not that different from the cumulative dataset D, we know that the new controllers are not that different. We can also stop when the parameters of our learned controller are $\theta^{(k+1)} \approx \theta^{(k)}$.

Chapter 8

Policy Gradient Methods

Reading

1. Sutton & Barto, Chapter 9–10, 13
2. Simple random search provides a competitive approach to reinforcement learning at https://arxiv.org/abs/1803.07055
3. Proximal Policy Optimization Algorithms https://arxiv.org/abs/1707.06347
4. Are Deep Policy Gradient Algorithms Truly Policy Gradient Algorithms? https://arxiv.org/abs/1811.02553
5. Asynchronous Methods for Deep Reinforcement Learning http://proceedings.mlr.press/v48/mniha16.pdf

This chapter discusses methods to learn the controller that minimizes a given cost functional over trajectories of an unknown dynamical system. We will use what is called the "policy gradient" which will be the main section of this chapter.

Recall from the last chapter that we were able to fit stochastic controllers of the form $u_{\widehat{\theta}}(\cdot \mid x)$ that is a probability distribution on the control-space U for each $x \in X$. We fitted u_θ using data from the expert in imitation learning. We did not learn the cost-to-go for the fitted controller, like we did in the lectures on dynamic programming. This is a clever choice: it is often easier to learn the controller in a typical problem than to *compute* the optimal cost-to-go as a parametric function $J^*(x)$.

❓ Can you give another instance when we have computed a controller previously in the class without coming up with its cost-to-go?

8.1 Standard problem setup in RL

Dynamics and rewards In this and the next few chapters we will always consider discrete-time stochastic dynamical systems with a stochastic controller with parameters (weights) θ. We denote them as follows

$$\begin{aligned} x_{k+1} &\sim p(\cdot \mid x_k, u_k) \text{ with noise denoted by } \epsilon_k \\ u_k &\sim u_\theta(x_k). \end{aligned}$$

We will also change perspective and instead of minimizing the infinite-horizon sum of a runtime cost, maximize the sum of a runtime reward

$$r(x, u) := -q(x, u).$$

We do so simply to conform to tradition and standard notation in reinforcement learning; the two are mathematically completely equivalent. We are interested in maximizing the expected value of the cumulative rewards over infinite-horizon trajectories of the system

$$J(\theta; x_0) = \mathop{\mathrm{E}}_{x_1, x_2, \ldots} \left[\underbrace{\sum_{k=0}^{\infty} \gamma^k \, r(x_k, u_k)}_{\text{discounted return}} \mid x_0 \right]; \tag{8.1}$$

where each $u_k \sim u_\theta(\cdot \mid x_k)$ and each $x_{k+1} \sim p(\cdot \mid x_k, u_k)$.

Trajectory space Let us write out one trajectory of such a system a bit more explicitly. We know that the probability of the next state x_{k+1} given x_k is $p(x_{k+1} \mid x_k, u_k)$. The probability of taking a control u_k at state x_k is $u_\theta(u_k \mid x_k)$. We denote an infinite trajectory by

$$\tau = x_0, u_0, x_1, u_1, \ldots.$$

The probability of this entire trajectory occurring is

$$p_\theta(\tau) = \prod_{k=0}^{\infty} p(x_{k+1} \mid x_k, u_k) \; u_\theta(u_k \mid x_k);$$

we have emphasized that the distribution of trajectories depends on the weights of controller θ. If we take the logarithm,

$$\log p_\theta(\tau) = \sum_{k=0}^{\infty} \log p(x_{k+1} \mid x_k, u_k) + \log u_\theta(u_k \mid x_k).$$

Given a trajectory $\tau = x_0, u_0, x_1, u_1, \ldots$, the sum

$$R(\tau) = \sum_{k=0}^{\infty} \gamma^k \, r(x_k, u_k) \tag{8.2}$$

is called the discounted return of the trajectory τ. Sometimes we will also talk of the undiscounted return of the trajectory which is the sum of the rewards up to some fixed finite horizon T without the discount factor pre-multiplier. Using this notation, we can write out objective from (8.1) as

$$J(\theta; x_0) = \mathop{\mathrm{E}}_{\tau \sim p_\theta(\tau)} [R(\tau) \mid x_0] \tag{8.3}$$

where $p(\tau)$ is the probability distribution of an infinitely long trajectory τ.

Observe what is probably the most important point in policy-gradient based reinforcement learning: the probability of trajectory is an infinite product of terms. All terms are smaller than 1 (they are probabilities), so it is essentially zero even if the state-space and the control-space are finite (even if they are small). Any given infinite (or long) trajectory is quite rare under the probability distribution of the stochastic controller. Policy-gradient methods sample lots of trajectories from the system and average the returns across these trajectories. Since the set of trajectories of even a small MDP is so large, sampling lots of trajectories, or even the most likely ones, is also very hard. This is a key challenge in getting RL algorithms to work.

Our goal in this chapter is to compute the best stochastic controller which maximizes the average discounted return. Mathematically, this amounts to finding

$$\widehat{\theta} = \operatorname*{argmax}_\theta J(\theta; x_0) := \mathop{\mathrm{E}}_{\tau \sim p_\theta(\tau)} [R(\tau) \mid x_0]\,. \tag{8.4}$$

The objective $J(\theta)$ is called the average return of the controller u_θ.

Computing the average return $J(\theta)$ Before we move on to optimizing $J(\theta)$, let us discuss how to compute it for given weights θ of the stochastic controller. We can sample n trajectories from the system and compute the an estimate of the expectation

$$\widehat{J}(\theta) \approx \frac{1}{n} \sum_{i=0}^{n} \sum_{k=0}^{T} \gamma^k r(x_k^i, u_k^i) \tag{8.5}$$

for some large time-horizon T and where each $u_k^i \sim u_\theta(\cdot \mid x_k^i)$.

Contrast (8.5) with the complexity of policy evaluation which was simply a system of linear equations. Evaluating the policy without having access to the dynamical system is harder.

8.2 Cross-Entropy Method (CEM)

Let us first consider a simple method to compute the best controller. The basic idea is to solve the problem

$$\widehat{\theta} = \operatorname*{argmax}_\theta J(\theta)$$

using gradient descent. We would like to update weights θ iteratively

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

where the step-size is $\eta > 0$ and $\nabla J(\theta)$ is the gradient of the objective $J(\theta)$ with respect to weights θ. Instead of computing the exact $\nabla J(\theta)$ which we will do in the next section, let us simply compute the gradient using a finite-difference approximation. The i^{th} entry of the gradient is

$$(\widehat{\nabla} J(\theta))_i = \frac{J(\theta + \epsilon \, e_i) - J(\theta - \epsilon \, e_i)}{2\epsilon} \approx \frac{\widehat{J}(\theta + \epsilon \, e_i) - \widehat{J}(\theta - \epsilon \, e_i)}{2\epsilon}.$$

where $e_i = [0, 0, \ldots, 0, 1, 0, \ldots]$ is a vector with 1 on the i^{th} entry. Each quantity $\widehat{J}$ is computed as the empirical average return of n trajectories from the system. We compute all entries of the objective using this approximation and update the parameters using

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

A more efficient way to compute the gradient using finite-differences Instead of picking perturbations e_i along the cardinal directions, let us sample them from a Gaussian distribution

$$\xi^i \sim N(0, \sigma^2 I)$$

for some user-chosen covariance σ^2. We can however no longer use the finite-difference formula to compute the derivative because the noise e is not aligned with the axes. We can however use a Taylor series approximation as follows. Observe that

$$J(\theta + \xi) \approx J(\theta) + \langle \nabla J(\theta), \xi \rangle$$

where $\langle \cdot, \cdot \rangle$ is the inner product. Given m samples $\xi^1, \ldots, \xi^m$ observe that

$$\begin{aligned} \widehat{J}(\theta + \xi^1) &= \widehat{J}(\theta) + \left\langle \nabla J(\theta), \, \xi^1 \right\rangle \\ \widehat{J}(\theta + \xi^2) &= \widehat{J}(\theta) + \left\langle \nabla J(\theta), \, \xi^2 \right\rangle \\ &\vdots \\ \widehat{J}(\theta + \xi^m) &= \widehat{J}(\theta) + \left\langle \nabla J(\theta), \, \xi^m \right\rangle. \end{aligned} \tag{8.6}$$

is a linear system of equations in $\nabla J(\theta) \in \mathbb{R}^p$ where $\theta \in \mathbb{R}^p$. All quantities $\widehat{J}$ are estimated as before using trajectories drawn from the system. We solve this linear system, e.g., using least-squares if $m > p$, to get an estimate of the gradient $\widehat{\nabla} J(\theta) \in \mathbb{R}^p$.

The Cross-Entropy Method is a more crude but simpler way to

implement the above least-squares formulation. At each iteration it updates the parameters using the formula

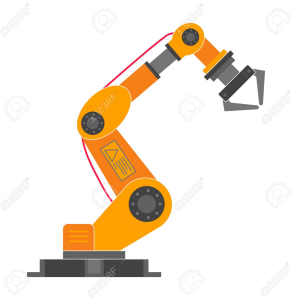
$$\theta^{(k+1)} = \mathop{\mathrm{E}}_{\theta \sim N(\theta^{(k)}, \sigma^2 I)} \left[\theta \, \mathbf{1}_{\{\widehat{J}(\theta) > \widehat{J}(\theta^{(k)})\}} \right]. \tag{8.7}$$

In simple words, the CEM samples a few stochastic controllers u_θ from a Gaussian (or any other distribution) centered around the current controller $u_{\theta^{(k)}}$ and updates the weights θ^k in a direction that leads to an increase in $\widehat{J}(\theta) > \widehat{J}(\theta^{(k)})$.

8.2.1 Some remarks on sample complexity of simulation-based methods

CEM may seem to be a particularly bad method to maximize $J(\theta)$, after all we are perturbing the weights of the stochastic controller randomly and updating the weights if they result in a better average return $\widehat{J}(\theta)$. This is likely to work well if the dimensionality of weights $\theta \in \mathbb{R}^p$, i.e., p, is not too large. But is unlikely to work well if we are sampling θ in high-dimensions. Typical applications are actually the latter, remember that we are interested in using a deep network as a stochastic controller and θ are the weights of the neural networks.

Let us do a quick computation, if the state is $x \in \mathbb{R}^d$ and $u \in \mathbb{R}^m$ with $d = 12$ (joint angles and velocities) and $m = 6$ for a six-degree of freedom robot manipulator



and if we use a two-layer neural network with 64 neurons in the hidden layer, the total number of weights $\theta \in \mathbb{R}^p$ for the function $u_\theta(\cdot \mid x) = N(\mu_\theta(x), \sigma^2_\theta(x) I)$ where $\sigma^2(x)$ is a vector in $\mathbb{R}^m$, is

$$\begin{aligned} p &= \underbrace{(12 \times 64 + 64) + (64 \times 6 + 6)}_{\text{for } \mu_\theta(x)} + \underbrace{(12 \times 64 + 64) + (64 \times 6 + 6)}_{\text{for } \sigma^2_\theta(x)} \\ &= 2,444. \end{aligned}$$

This is a very high-dimensional space to sample exhaustively. Note that it is quite large even if the input and output dimensions of the neural network are not too large. To appreciate the complexity of computing the gradient $\nabla J(\theta)$, let us think of how to compute it using finite-differences.

We need two estimates $\widehat{J}(\theta - \epsilon e_i)$ and $\widehat{J}(\theta + \epsilon e_i)$ for every dimension $i \in \{1, \ldots, p\}$. Each estimate requires us to obtain n trajectories from the system. Since the number of trajectories that a robot can take is quite diverse, we should use a large n, so let's pick $n = 100$. The total number of *trajectories* required to update the parameters $\theta^{(k)}$ *at each iteration* is

$$2\, p\, n \approx 10^6.$$

This is an absurdly large number, and things are even more daunting when we realize that each update of the weights requires us to sample these many trajectories from the system. It is not reasonable to sample such a large number of trajectories from an actual robot, that too for each update of the weights.

ℹ For comparison, a busy espresso bar in a city makes about 500 shots per day. The espresso machine would have to work for 5 years *without breaking down* to make 10^6 shots.

Using fast simulators for RL If we expand our horizon and think of learning controllers in simulation, things feel much more reasonable. While running a large number of trajectories may degrade a real robot beyond use, doing so requires just computation time in a robot simulator. There is a large number of simulators that are available with various capabilities, e.g., Gazebo (http://gazebosim.org) is a sophisticated simulator inside ROS that uses a number of Physics engines such as Bullet (https://pybullet.org/wordpress), MuJoCo (http://www.mujoco.org) is incredibly fast although not very good modeling contact, Unity is a popular platform to simulate driving and more complicated scenes (https://docs.nvidia.com/isaac/isaac/doc/simulation/unity3d.html), Drake (https://drake.mit.edu) is better at contact modeling but more complex and slower. Most robotics companies develop their own driving simulators in-house. The assigned reading (#2) for this chapter is a paper which develops a very fast implementation of the CEM for use in simulators.

Working well in simulation does not mean that a controller works well on the real robot It is important to realize that a simulator is not equivalent to the physical robot. Each simulator makes certain trade-offs in capturing the dynamics of the real system and it is not a given that a controller that was learned using data from a simulator will work well on a real robot. For instance, OpenAI had to develop a large number of tricks (which took about a year) to modify the simulator to enable the learned policy to work well on a robot (https://openai.com/blog/learning-dexterity) for a fairly narrow set of tasks.

8.3 The Policy Gradient

In this section, we will study how to take the gradient of the objective $J(\theta)$, without using finite-differences.

We would like to solve the optimization problem

$$\max_{\theta} J(\theta) := \mathop{\mathrm{E}}_{\tau \sim p_\theta(\tau)} [R(\tau) \mid x_0]$$

We will suppress the dependence on x_0 to keep the notation clear. The expectation is taken over all trajectories starting at state x_0 realized using the stochastic controller $u_\theta(\cdot \mid x)$. We to update weights θ using gradient descent which amounts to

$$\theta^{(k+1)} = \theta^{(k)} + \eta \nabla_\theta \mathop{\mathrm{E}}_{\tau \sim p_\theta(\tau)} [R(\tau)] \,.$$

First let us note that the distribution p_θ using which we compute the expectation also depends on the weights θ. This is why we cannot simply move the derivative ∇_θ inside the expectation

$$\nabla_\theta \mathop{\mathrm{E}}_{\tau \sim p_\theta(\tau)} [R(\tau)] \neq \mathop{\mathrm{E}}_{\tau \sim p_\theta(\tau)} [\nabla_\theta R(\tau)] \,.$$

We need to think of a new technique to compute the gradient above. Essentially, we would like to do the chain rule of calculus but where one of the functions in the chain is an expectation. The likelihood-ratio trick described next allows us to take such derivatives. Here is how the computation goes

$$\begin{aligned}
\nabla_\theta \mathop{\mathrm{E}}_{\tau \sim p_\theta} [R(\tau)] &= \nabla_\theta \int R(\tau) p_\theta(\tau) \mathrm{d}\tau \\
&= \int R(\tau) \nabla_\theta p_\theta(\tau) \, \mathrm{d}\tau \\
&\text{(move the gradient inside, integral is over trajectories } \tau \text{ which do not depend on} \theta \text{ themselves)} \\
&= \int R(\tau) p_\theta(\tau) \frac{\nabla p_\theta(\tau)}{p_\theta(\tau)} \mathrm{d}\tau \\
&= \int R(\tau) p_\theta(\tau) \nabla \log p_\theta(\tau) \mathrm{d}\tau \\
&= \mathop{\mathrm{E}}_{\tau \sim p_\theta(\tau)} [R(\tau) \nabla \log p_\theta(\tau)] \\
&\approx \frac{1}{n} \sum_{i=1}^{n} R(\tau^i) \nabla \log p_\theta(\tau^i)
\end{aligned} \tag{8.8}$$

This is called the likelihood-ratio trick to compute the policy gradient. It simply multiplies and divides by the term $p_\theta(\tau)$ and rewrites the term $\frac{\nabla p_\theta}{p_\theta} = \nabla \log p_\theta$. It gives us a neat way to compute the gradient: we sample n trajectories $\tau^1, \ldots, \tau^n$ from the system and average the return of each trajectory $R(\tau^i)$ weighted by the gradient of the likelihood of taking each trajectory $\log p_\theta(\tau^i)$. The central point to remember here is

that the gradient

$$\begin{aligned}\nabla_\theta \log p_\theta(\tau^i) &= \nabla_\theta \sum_{k=0}^{T} \log p(x_{k+1}^i \mid x_k^i, u_k^i) + \log u_\theta(u_k^i \mid x_k^i) \\ &= \sum_{k=0}^{T} \nabla_\theta \log u_\theta(u_k^i \mid x_k^i)\end{aligned} \tag{8.9}$$

is computed using backpropagation for a neural network. This expression is called the policy gradient because it is the gradient of the objective $J(\theta)$ with respect to the parameters of the controller/policy θ.

Variance of policy gradient The expression for the policy gradient may seem like a sleight of hand. It is a clean expression to get the gradient of the objective but also comes with a number of problems. Observe that

$$\begin{aligned}\nabla \mathop{\mathrm{E}}_{\tau\sim p_\theta(\tau)}[R(\tau)] &= \mathop{\mathrm{E}}_{\tau\sim p_\theta(\tau)}\left[R(\tau)\frac{\nabla p_\theta(\tau)}{p_\theta(\tau)}\right] \\ &\approx \frac{1}{n}\sum_{i=1}^{n} R(\tau^i)\,\frac{\nabla\, p_\theta(\tau^i)}{p_\theta(\tau^i)}.\end{aligned}$$

If we sample trajectories τ^i that are not very likely under the distribution $p_\theta(\tau)$, the denominator in some of the summands can be very small. For trajectories that are likely, the denominator is large. The empirical estimate of the expectation using n trajectories where some terms are very small and some others very large, therefore has a large variance. So one does need lots of trajectories from the system/simulator to compute a reasonable approximation of the policy gradient.

8.3.1 Reducing the variance of the policy gradient

Control variates You will perhaps appreciate that computing the accurate policy gradient is very hard in practice. Control variates is a general concept from the literature on Monte Carlo integration and is typically introduced as follows. Say we have a random variable X and we would like to guess its expected value $\mu = \mathrm{E}[X]$. Note that X is an unbiased estimator of μ but it may have a large variance. If we have another random variable Y with known expected value $\mathrm{E}[Y]$, then

$$\hat{X} = X + c(Y - \mathrm{E}[Y]) \tag{8.10}$$

is also an unbiased estimator for μ for any value of c. The variance of $\hat{X}$ is

$$\mathrm{Var}(\hat{X}) = \mathrm{Var}(X) + c^2\,\mathrm{Var}(Y) + 2c\ \mathrm{Cov}(X, Y).$$

which is minimized for

$$c^* = -\frac{\mathrm{Cov}(X, Y)}{\mathrm{Var}(Y)}$$

for which we have

$$\text{Var}(\hat{X}) = \text{Var}(X) - c^{*2}\ \text{Var}(Y) = \big(1 - \text{Corr}^2(X,Y)\big)\,\text{Var}(X).$$

By subtracting $Y - \text{E}[Y]$ from our observed random variable X, we have reduced the variance of X if the correlation between X and Y is non-zero. Most importantly, note that no matter what Y we plug into the above expression, we can never increase the variance of X; the worst that can happen is that we pick a Y that is completely uncorrelated with X and end up achieving nothing.

Baseline We will now use the concept of a control variate to reduce the variance of the policy gradient. This is known as "building a baseline". The simplest baseline one can build is to subtract a constant value from the return. Consider the PG given by

$$\begin{aligned}\nabla J(\theta) &= \mathop{\text{E}}_{\tau\sim p_\theta}\left[R(\tau)\,\nabla \log p_\theta(\tau)\right] \\ &= \mathop{\text{E}}_{\tau\sim p_\theta(\tau)}\left[(R(\tau) - b)\ \nabla \log p_\theta(\tau)\right].\end{aligned}$$

Observe that

$$\begin{aligned}\mathop{\text{E}}_{\tau\sim p_\theta(\tau)}[b\nabla \log p_\theta(\tau)] &= \int \text{d}\tau b\, p_\theta(\tau)\nabla \log p_\theta(\tau) \\ &= \int \text{d}\tau b\, \nabla p_\theta(\tau) = b\,\nabla \int \text{d}\tau\, p_\theta(\tau) = b\,\nabla 1 = 0.\end{aligned}$$

Example 1: Using the average returns of a mini-batch as the baseline What is the simplest baseline b we can cook up? Let us write the mini-batch version of the policy gradient

$$\widehat{\nabla} J(\theta) := \frac{1}{\cancel{b}} \sum_{i=1}^{\cancel{b}} \left[R(\tau^i)\nabla \log p_\theta(\tau^i)\right].$$

where $\tau^1, \ldots, \tau^{\cancel{b}}$ are trajectories that are a part of our mini-batch. We can set

$$b = \frac{1}{\cancel{b}} \sum_{i=1}^{\cancel{b}} R(\tau^i)$$

can use the variance-reduced gradient

$$\widehat{\nabla} J(\theta) = \frac{1}{\cancel{b}} \sum_{i=1}^{\cancel{b}} \left[\big(R(\tau^i) - b\big)\ \nabla \log p_\theta(\tau^i)\right].$$

This is a one-line change in your code for policy gradient so there is no reason not to do it.

Example 2: A weighted averaged of the returns using the log-likelihood of the trajectory The previous example showed how we can use one

constant baseline, namely the average of the discounted returns of all trajectories in a mini-batch. What is the **best** constant b we can use? We can perform a similar computation as done in the control variate introduction to minimize the variance of the policy gradient to get the following.

$$\begin{aligned}\delta\left(\widehat{\nabla}_{\theta_i} J(\theta)\right) &= \mathop{\mathrm{E}}_{\tau}\left[\left((R(\tau) - b_i)\nabla_{\theta_i}\log p_\theta(\tau)\right)^2\right] - \left(\mathop{\mathrm{E}}_{\tau}\left[\left((R(\tau) - b_i)\right)\nabla_{\theta_i}\log p_\theta(\tau)\right]\right)^2 \\ &= \mathop{\mathrm{E}}_{\tau}\left[\left((R(\tau) - b_i)\nabla_{\theta_i}\log p_\theta(\tau)\right)^2\right] - \left(\widehat{\nabla}_{\theta_i} J(\theta)\right)^2.\end{aligned}$$

Set

$$\frac{\delta\left(\widehat{\nabla}_{\theta_i} J(\theta)\right)}{\mathrm{d}b_i} = 0$$

in the above expression to get

$$b_i = \frac{\mathrm{E}_\tau\left[\left(\nabla_{\theta_i}\log p_\theta(\tau)\right)^2 R(\tau)\right]}{\mathrm{E}_\tau\left[\left(\nabla_{\theta_i}\log p_\theta(\tau)\right)^2\right]}$$

which is the baseline you should subtract from the gradient of the i^{th} parameter θ_i to result in the largest variance reduction. This expression is just the expected return but it is weighted by the magnitude of the gradient, this again is 1–2 lines of code.

❓ Show that any function that only depends on the state x that can be written as a potential (i.e., whose derivative, the force, is conservative) can be used as a baseline in the policy gradient. This technique is known as reward shaping.

8.4 An alternative expression for the policy gradient

We will first define an important quantity that helps us think of RL algorithms.

Definition 8.1 (Discounted state visitation frequency). Given a stochastic controller $u_\theta(\cdot \mid x)$ the discounted state visitation frequency for a discrete-time dynamical system is given by

$$d^\theta(x) = \sum_{k=0}^{\infty} \gamma^k \,\mathrm{P}(x_k = x \mid x_0, u_k \sim u_\theta(\cdot \mid x_k)).$$

The distribution $d^\theta(x)$ is the probability of visiting a state x computed over all trajectories of the system that start at the initial state x_0. If $\gamma = 1$, this is the steady-state distribution of the Markov chain underlying the Markov Decision Process where at each step the MDP choses the control $u_k \sim u_\theta(\cdot \mid x_k)$. The fact that we have defined the discounted distribution is a technicality; this version is seen in the policy gradient expression. You will also notice that $d^\theta(x)$ is not a normalized distribution. The normalization constant is difficult to characterize both theoretically and empirically and we will not worry about it here; RL algorithms do not require it.

Q-function Using the discounted state visitation frequency, the policy gradient that we saw before can be written in terms of the value function as follows.

$$\begin{aligned}\nabla J(\theta) &= \mathop{\mathrm{E}}_{\tau\sim p_\theta}\left[R(\tau)\nabla \log p_\theta(\tau)\right] \\ &= \mathop{\mathrm{E}}_{x\sim d^\theta}\mathop{\mathrm{E}}_{u\sim u_\theta(\cdot|x)}\left[q^\theta(x,u)\nabla_\theta \log u_\theta(u \mid x)\right].\end{aligned} \tag{8.11}$$

The function $q^\theta(x,u)$ is similar to the cost-to-go that we have studied in dynamic programming and is called the Q-function

$$q^\theta(x,u) = \mathop{\mathrm{E}}_{\tau\sim p_\theta(\tau)}\left[R(\tau) \mid x_0 = x, u_0 = u\right]. \tag{8.12}$$

It is the infinite-horizon discounted cumulative reward (i.e., the return) if the system starts at state x and takes the control u in the first step and runs the controller $u_\theta(\cdot \mid x)$ for all steps thereafter. We make the dependence of q^θ on the parameters θ of the controller explicit.

ℹ The derivation of this expression is easy although tedious, you can find it in the Appendix of the paper "Policy gradient methods for reinforcement learning with function approximation" at https://papers.nips.cc/paper/1713-policy-gradient-methods-for-reinforcement-learning-with-function-approximation.pdf.

Compare the above formula for the policy gradient with the one we had before in (8.8)

$$\begin{aligned}\widehat{\nabla} J(\theta) &= \mathop{\mathrm{E}}_{\tau\sim p_\theta(\tau)}\left[R(\tau)\,\nabla \log p_\theta(\tau)\right] \\ &= \mathop{\mathrm{E}}_{\tau\sim p_\theta(\tau)}\left[\left(\sum_{k=0}^{T}\gamma^k r(x_k,u_k)\right)\left(\sum_{k=0}^{T}\nabla \log u_\theta(u_k \mid x_k)\right)\right].\end{aligned}$$

It is important to notice that this is an expectation over trajectories; whereas (8.11) is an expectation over states x sampled from the discounted state visitation frequency. The control u_k for both is sampled from the stochastic controller at each time-step k. The most important distinction is that (8.11) involves the expectation of the Q-function q^θ weighted by the gradient of the log-likelihood of picking each control action. There are numerous hacky ways of deriving (8.11) from (8.8) but remember that they are fundamentally different expressions of the *same policy gradient*.

This expression allows understanding of a number of properties of reinforcement learning.

1. While the algorithm collects the data, states that are unlikely under the distribution d^θ contribute little to (8.11). In other words, the policy gradient is insensitive to such states. The policy update will not consider these unlikely states that the system is prone to visit infrequently using the controller u_θ.

2. The opposite happens for states which are very likely. For two controls u_1, u_2 at the same state x, the policy increases the log-likelihood of taking the controls weighted by their values $q^\theta(x,u_1)$ and $q^\theta(x,u_2)$. This is sort of the "definition" of reinforcement learning. In the expression (8.8) the gradient was increasing the

likelihood of trajectories with high returns, here it deals with states and controls individually.

8.4.1 Implementing the new expression

Suppose we have a stochastic control that is a Gaussian

$$u_\theta(u \mid x) = \frac{1}{(2\pi\sigma^2)^{p/2}}\, e^{-\frac{\left\|u-\theta^\top x\right\|^2}{2\sigma^2}}$$

where $\theta \in \mathbb{R}^{d\times p}$ and $u \in \mathbb{R}^p$; the variance σ can be chosen by the user. We can easily compute $\log u_\theta(u \mid x)$ in (8.11). How should one compute $q^\theta(x,u)$ in (8.12)? We can again estimate it using sample trajectories from the system; each of these trajectories would have to start from a state x and the control at the first step would be u, with the controller u_θ being used thereafter. Note that we have one such trajectory, namely the remainder of the trajectory where we encountered (x,u) while sampling trajectories for the policy gradient in (8.11). In practice, we do not sample trajectories a second time, we simply take this trajectory, let us call it $\tau_{x,u}$ and set

$$q^\theta(x,u) = \sum_{k=0}^{T} \gamma^k r(x_k,u_k)$$

for some large time-horizon T where $(x_0,u_0) = (x,u)$ and the summation is evaluated for (x_k,u_k) that lie on the trajectory $\tau_{x,u}$. Effectively, we are evaluating (8.12) using one sample trajectory, a highly erroneous estimate of q^θ.

8.5 Actor-Critic methods

We can of course do more sophisticated things to evaluate the Q-function q^θ in our new expression of the policy gradient.

> Actor-Critic methods fit a Q-function to the data collected from the system using the current controller (policy evaluation step) and then use this fitted Q-function in the expression of the policy gradient (8.11) to update the controller. In this sense, Actor-Critic methods are conceptually similar to policy iteration.

In order to understand how to fit the Q-function, first recall that it should satisfy the Bellman equation. This means

$$q^\theta(x,u) = r(x,u) + \gamma \mathop{\mathrm{E}}_{u\sim u_\theta(\cdot\mid x'),x'\sim P(\cdot\mid x,u)} \left[q^\theta(x',u')\right]. \tag{8.13}$$

We do not know a model for the system so we cannot evaluate the expectation over $x' \sim \mathrm{P}(\cdot \mid x,u)$ like we used to in dynamic programming. But we do have the ability to get trajectories τ^i from the system.

Let's say (x_k^i, u_k^i) lie on τ^i at time-step k. We can then estimate the expectation over $\mathrm{P}(\cdot \mid x_k^i, u_k^i)$ using simply x_{k+1}^i and the expectation over the controls using simply u_{k+1}^i to write

$$q^\theta(x_k^i, u_k^i) \approx r(x_k^i, u_k^i) + \gamma\, q^\theta(x_{k+1}^i, u_{k+1}^i) \quad \text{for all } i \leq n, k \leq T.$$

This is a nice constraint on the Q-function. If this were a discrete-state, discrete-control MDP, it is a set of linear equations for the q-values. These constraints would be akin to our linear equations for evaluating a policy in dynamic programming except that instead of using the dynamics model (the transition matrix), we are using trajectories sampled from the system.

Parameterizing the Q-function using a neural network If we are dealing with a continuous state/control-space, we can think of parameterizing the q-function using parameters φ

$$q_\varphi^\theta(x, u) : X \times U \to \mathbb{R}.$$

The parameterization is similar to the parameterization of the controller, e.g., just like we would write a deterministic controller as

$$u_\theta(x) = \theta^\top x$$

we can think of a linear Q-function of the form

$$q_\varphi^\theta(x, u) = \varphi^\top \begin{bmatrix} x \\ u \end{bmatrix}, \quad \varphi \in \mathbb{R}^{m+d}$$

which is a linear function in the states and controls. You can also think of using something like

$$q_\varphi^\theta(x, u) = \begin{bmatrix} 1 & x & u \end{bmatrix} \varphi \begin{bmatrix} 1 \\ x \\ u \end{bmatrix} \quad \varphi \in \mathbb{R}^{(m+d+1)\times(m+d+1)}.$$

which is quadratic in the states and controls, or in general a deep network with weights φ as the Q-function.

Fitting the Q-function We can now "fit" the parameters of the Q-function by solving the problem

$$\widehat{\varphi} = \operatorname*{argmin}_\varphi \frac{1}{n(T+1)} \sum_{i=1}^{n} \sum_{k=0}^{T} \left\| q_\varphi^\theta(x_k^i, u_k^i) - r(x_k^i, u_k^i) - \gamma\, q_\varphi^\theta(x_{k+1}^i, u_{k+1}^i) \right\|^2. \tag{8.14}$$

which is nothing other than enforcing the Bellman equation in (8.13). If the Q-function is linear in $[x, u]$ this is a least squares problem, if it is quadratic the problem is a quadratic optimization problem which can also be solved efficiently, in general we would solve this problem using stochastic gradient descent if we are parameterizing the Q-function using a deep network. Such a Q-function is called the "critic" because it *evaluates*

the controller u_θ, or the "actor". This version of the policy gradient where one fits the parameters of both the controller and the Q-function are called Actor-Critic methods.

Actor-Critic Methods We fit a deep network with weights θ to parameterize a stochastic controller $u_\theta(\cdot \mid x)$ and another deep network with weights φ to parameterize the Q-function of this controller, $q^\theta_\varphi(x, u)$. Let the controller weights at the k^{th} iteration be $\theta^{(k)}$ and the Q-function weights be φ^k.

1. Sample n trajectories, each of T timesteps, using the current controller $u^{(k)}_\theta(\cdot \mid x)$.
2. Fit a Q-function $q^{\theta^{(k)}}_{\varphi^{k+1}}$ to this data using (8.14). using stochastic gradient descent. While performing this fitting (although it is not mathematically sound), it is common to use initialize the Q-function weights to their values from the previous iteration φ_κ.
3. Compute the policy gradient using the alternative expression in (8.11) and update parameters of the policy to $\theta^{(k+1)}$.

ℹ We will be pedantic and always write the q-function as q^θ_φ. The superscript θ denotes that this is the q-function corresponding to the θ-parameterized controller u_θ. The subscript denotes that the q-function is parameterized by parameters φ.

8.5.1 Advantage function

The new expression for the policy gradient in (8.11) also has a large variance; this should be no surprise, it is after all equal to the old expression. We can however perform variance reduction on this using the value function.

Our goal as before would be construct a relevant baseline to subtract from the Q-function. It turns out that any function that depends only upon the state x and is a potential, i.e., whose gradient has zero curl, is a valid baseline. This gives a powerful baseline for us to to use in policy gradients. We can use the value function as the baseline. The value function for controls taken by the controller $u_\theta(\cdot \mid x)$ (notice that this is not the optimal value function, it is simply the policy evaluation) is given by

$$v^\theta(x) = \mathop{\mathrm{E}}_{\tau \sim p_\theta(\tau)} \left[R(\tau) \mid x_0 = x \right]$$

where $u_k \sim u_\theta(\cdot \mid x_k)$ at each timestep. We also know that the value function is the expected value of the Q-function across different controls sampled by the controller

$$v^\theta(x) = \mathop{\mathrm{E}}_{u \sim u_\theta(\cdot \mid x)} \left[q^\theta(x, u) \right]. \tag{8.15}$$

The value function again satisfies the dynamic programming principle/-

Bellman equation

$$v^\theta(x) = \mathop{\mathrm{E}}_{u\sim u_\theta(\cdot|x)} \left[r(x,u) + \gamma \mathop{\mathrm{E}}_{x'\sim \mathrm{P}(\cdot|x,u)} \left[v^\theta(x') \right] \right].$$

We again parameterize the value function

$$v^\theta_\psi(x) : X \to \mathbb{R}$$

using parameters ψ and fit it to the data in the same way as (8.14) to get

$$\widehat{\psi} = \operatorname*{argmin}_\psi \frac{1}{n(T+1)} \sum_{i=1}^{n} \sum_{k=0}^{T} \left\| v^\theta_\psi(x^i_k) - r(x^i_k, u^i_k) - \gamma\, v^\theta_\psi(x^i_{k+1}) \right\|^2. \tag{8.16}$$

Using this baseline can modify the policy gradient to be

$$\nabla J(\theta) = \mathop{\mathrm{E}}_{x\sim d^\theta} \mathop{\mathrm{E}}_{u\sim u_\theta(\cdot|x)} \left[\left(\underbrace{q^\theta_\varphi(x,u) - v^\theta_\psi(x)}_{a^\theta_{\varphi,\psi}(x,u)} \right) \nabla_\theta \log u_\theta(u \mid x) \right]. \tag{8.17}$$

where each of the functions q^θ_φ and v^θ_ψ are themselves fitted using (8.14) and (8.16) respectively. The difference

$$\begin{aligned} a^\theta_{\varphi,\psi}(x,u) &= q^\theta_\varphi(x,u) - v^\theta_\psi(x) \\ &\approx q^\theta_\varphi(x,u) - \mathop{\mathrm{E}}_{u\sim u_\theta(\cdot|x)} \left[q^\theta_\varphi(x,u) \right] \end{aligned} \tag{8.18}$$

is called the advantage function. It measures how much better the particular control u is for a state x as compared to the average return of controls sampled from the controller at that state. The form (8.17) is the most commonly implemented form in research papers whenever they say "we use the policy gradient".

The advantage function is very useful while doing theoretical work on RL algorithms. But it is also extremely useful in practice. It imposes a constraint upon our estimate q^θ_φ and the estimate v^θ_ψ. If we are not solving (8.14) and (8.16) to completion, we may benefit by imposing this constraint on the advantage function. Can you think of a way?

8.5.2 Discussion

Policy gradients are, in general, a complicated suite of algorithms to implement. You will see some of this complexity when you implement the controller for something as simple as the simple pendulum. The key challenges with implementing policy gradients come from the following.

1. Need lots of data, each parameter update requires fresh data from the systems. Typical problems may need a million trajectories, most robots would break before one gets this much data from them if one implements these algorithms naively.

2. The log-likelihood ratio trick has a high variance due to $u_\theta(\cdot \mid x)$

being in the denominator of the expression, so we need to implement complex variance reduction techniques such as actor-critic methods.

3. Fitting the Q-function and the value function is not easy, each parameter update of the policy ideally requires you to solve the entire problems (8.14) and (8.16). In practice, we only perform a few steps of SGD to solve the two problems and reuse the solution of k^{th} iteration update as an initialization of the $k+1^{\text{th}}$ update. This is known as "warm start" in the optimization literature and reduces the computational cost of fitting the Q/value-functions from scratch each time.

4. The Q/value-function fitted in iteration k may be poor estimates of the Q/value at iteration $k+1$ for the new controller $u_{\theta^{(k+1)}}(\cdot \mid x)$. If the controller parameters change quickly, $\theta^{(k+1)}$ is very different from $\theta^{(k)}$, then so are $q^{\theta^{(k+1)}}$ and $v^{\theta^{(k+1)}}$. There is a very fine balance between training quickly and retaining the efficiency of warm start; and tuning this in practice is quite difficult. A large number of policy gradient algorithms like TRPO (https://arxiv.org/abs/1502.05477) and PPO (https://arxiv.org/abs/1707.06347) try to alleviate this with varying degrees of success.

5. The latter, PPO, is a good policy-gradient-based algorithm to try on a new problem. For instance, in a very impressive demonstration, it was used to build an RL agent to play StarCraft (https://openai.com/blog/openai-five). We will see better RL methods in the next chapter.

8.6 Pre-conditioning the gradient in reinforcement learning

⚠ Pre-conditioning is the second classical trick to accelerating optimization.

Newton's method for solving a minimizing problem $\operatorname{argmin} \ell(w)$ corresponds to weight updates

$$w^{(t+1)} = w^{(t)} - \left(\nabla^2 \ell(w^{(t)})\right)^{-1} \nabla \ell(w^{(t)}). \tag{8.19}$$

Newton's method converges quadratically, i.e., $\ell(w^{(t+1)}) - \ell(w^*) \leq c\left(\ell(w^{(t)}) - \ell(w^*)\right)^2$, the sub-optimality shrinks very quickly. It is however very difficult to implement Newton's method for large optimization problems, such as deep networks, because the Hessian $\nabla^2 \ell(w) \in \mathbb{R}^{p \times p}$ is a very large matrix. Inverting such large matrices at each step of descent is very expensive and furthermore, for many real problems the Hessian is extremely ill-conditioned $\lambda_{\text{max}}/\lambda_{\text{min}} \approx 10^8$; in fact for deep networks which have two times as many weights as the number of weights the Hessian always has zero eigenvalues. In general, we can use any other pre-conditioning matrix in place of the inverse Hessian so long as it is positive semi-definite

⚠ One can implement Newton's method for some small-scale neural networks using Hessian-vector products.

$$w^{(t+1)} = w^{(t)} - \eta G^{-1/2} \nabla \ell(w^{(t)}); \tag{8.20}$$

the square root is taken to emphasize the positive definiteness. We can approximate the Hessian using the so-called Gauss-Newton matrix:

$$\nabla^2 \ell(w) \approx \frac{1}{n} \sum_{i=1}^{n} \nabla \ell^i(w) \nabla \ell^i(w)^\top \tag{8.21}$$

if the gradients $\nabla \ell^i(w) \approx 0$. In other words, the Hessian, up to an approximation that holds towards the end of training, is equal to the average of the outer products of the individual sample gradients. This is still a very large matrix, so let's use only the diagonal and set the off-diagonal elements to zero:

$$\sqrt{G_{ii}} \equiv \left(\nabla^2 \ell(w)\right)_{ii} = \frac{1}{n} \sum_{i=1}^{n} \left(\frac{\mathrm{d}\ell^i(w)}{\mathrm{d}w_i} \right)^2,$$

i.e., the diagonal elements of the Hessian under this approximation are the averages of the squares of the gradients of the individual samples. We can now write the pre-conditioned descent direction as

$$\left(G^{-1/2} \nabla \ell(w^{(t)})\right)_i \approx \frac{\frac{\mathrm{d}\ell(w^{(t)})}{\mathrm{d}w_i}}{\sqrt{\frac{1}{n}\sum_{i=1}^{n} \left(\frac{\mathrm{d}\ell^i(w)}{\mathrm{d}w_i} \right)^2}}. \tag{8.22}$$

When implementing this descent direction using stochastic gradients, we can calculate the denominator using a run-time average of the square of the gradients

$$\frac{1}{t} \sum_{s=1}^{t} \nabla_i \ell^{\omega_s}(w^{(s)})$$

> Adam implements a pre-conditioned version of the gradient descent equation using a (square root of the) diagonal approximation of the Hessian as the pre-conditioning matrix. The new descent direction is (8.22) where again the denominator is calculated using exponential averaging instead of the Euclidean average. This is very useful to train deep networks in general because the gradients tend to ill-conditioned. This is particularly useful in reinforcement learning because the controller and the value function objectives have very different magnitudes, the former is equal to the average return while the latter is often near zero because it is the Bellman error.

8.6.1 The Levenberg-Marquardt algorithm

As we said, in many real problems when neural networks are being fitted, the Hessian tends to be very ill-conditioned. In such situations, it is useful to damp the iterations by taking a kind of average between gradient descent and (8.20) which uses the Gauss-Newton matrix. Consider an

optimization problem given by

$$w^* = \underset{w}{\text{argmin}}\, \ell(w) \doteq \sum_{i=1}^{N} \left(y^i - f(x^i, w)\right)^2 .$$

where $f(x^i, w)i$ is the output on the i^{th} datum. In RL this would be, say, one of the elements in the average over the Bellman error. From an iterate $w^{(t)}$, if we are making an update Δw to the weights, we should have

$$f(x^i, w^{(t)} + \Delta w) = f(x^i, w^{(t)}) + J^i(w^{(t)})\Delta w$$

using the Taylor series where the Jacobian $\mathbb{R}^p \ni J^i(w^{(t)}) = \frac{\partial f(x^i, w)}{\partial w} \mid_{w=w^{(t)}}$. In other words,

$$\begin{aligned}
\ell(w + \Delta w) &= \sum_{i=1}^{N} \left(y^i - f(x^i, w + \Delta w)\right)^2 \\
&= \left\| \vec{y} - \vec{f}(w) - J\Delta w \right\|^2 \\
&= \left(\vec{y} - \vec{f}(w)\right)^\top \left(\vec{y} - \vec{f}(w)\right) - 2\left(\vec{y} - \vec{f}(w)\right)^\top J\Delta w + \Delta w^\top J^\top J \Delta w.
\end{aligned}$$

> The Levenberg-Marquardt algorithm finds the best step Δw that leads to maximal reduction in the objective $\ell(w)$ using a locally quadratic approximation of the objective. By definition, such a step should always lead to improvement in the objective.

If we take the derivative of the right-hand side with respect to Δw to calculate the best step forward and minimize $\ell(w + \Delta w)$, we get

$$\left(J^\top J\right) \Delta w = J^\top (\vec{y} - \vec{f}(x)). \tag{8.23}$$

This is a linear equation for the step Δw that can be solved by inverting $J^\top J$, or using a damped version which looks like

$$\left(J^\top J + \lambda I\right) \Delta w = J^\top (\vec{y} - \vec{f}(x)).$$

The (non-negative) damping factor λ is adjusted at each iteration. If reduction of the loss is rapid, a smaller value can be used, bringing the algorithm closer to the Gauss–Newton algorithm, whereas if an iteration gives insufficient reduction in the residual, λ can be increased, giving a step closer to the gradient-descent direction.

> This is called the Levenberg-Marquardt algorithm and the region

defined by the damping parameter λ is called the trust-region. Effectively we trust our local approximation using the Taylor series within a ball of size λ. This is one of the most powerful optimization algorithms one can use, much better than SGD or Adam because it uses information about the curvature. It is easy to use when the Jacobian can be calculated, e.g., for small problems or problems with small neural networks. You can also use it from within scipy.

8.6.2 Trust region policy optimization (TRPO)

We will next discuss a trust-region-based algorithm for policy gradient. Consider two controllers u_θ and $u_{\theta'}$. Kakade and Langford proved in a 2002 paper titled "Approximately Optimal Approximate Reinforcement Learning" a very beautiful result. This is the basis of the TRPO algorithm and a later one called PPO which improves its practical performance. This paper showed that

$$v^{\theta}(x) = v^{\theta'}(x) + \underbrace{\mathop{\mathrm{E}}_{\tau \sim p_\theta}}_{\text{new controller}} \underbrace{\left[\sum_t \gamma^t a^{\theta'}(x_t, u_t) \right]}_{\text{old controller's advantage}} \tag{8.24}$$

for all trajectories τ that begin at state x. In words, the average return of one controller u_θ can be calculated using trajectories collected from this new controller as the discounted averaged of the the advantage of the old controller. We can rewrite this using the discounted state visitation frequency as

$$v^{\theta}(x) = v^{\theta'}(x) + \mathop{\mathrm{E}}_{x \sim d^\theta, u \sim u_\theta} \left[a^{\theta'}(x, u) \right].$$

This is quite nice because given an old controller $u_{\theta'}$, if we can ensure that the new one u_θ takes controls where the advantage of the old controller was positive, at each state,

$$\mathop{\mathrm{E}}_{u \sim u_\theta} \left[a^{\theta'}(x, u) \right] \geq 0,$$

(or at least on average over the states), then the new controller has a better average reward.

We could use this property to improve the current controller. You can think of this as the property satisfied by each iteration of policy iteration algorithm, the new policy's controller is better than the old policy's controller in at least one of the states of the MDP.

If we cannot find this better controller u_θ then we have converged to the optimal policy.

In practice, we estimate the advantage and the state visitation frequency using samples. Therefore, it is not easy to implement the idea above. To work around this, Kakade and Langford suggested a clever idea that is a local approximation of this objective (conceptually very similar to the Taylor series approximation we took above in Levenberg-Marquardt)

$$v^{\theta}(x) = v^{\theta'}(x) + \underbrace{\mathop{\mathrm{E}}_{x\sim d^{\theta'}, u\sim u_\theta}}_{\text{old visitation frequency}} \left[a^{\theta'}(x,u)\right]. \tag{8.25}$$

Notice the subtle change where the states are now sampled from the visitation frequency of the old controller. Certainly if $u_\theta \approx u_{\theta'}$ for all states, then the two are similar.

Just like we use a trust-region in Levenberg-Marquardt to control the changes in the weights using a local approximation, we use a trust region in TRPO because we do not know much of a step to take when we update this old controller using this local approximation of the objective. We used a ball in the weight space of size λ to control the step. In RL, we can use some notion of the discrepancy between the old controller and the updated controller. In TRPO, this is done using some divergence between the controls sampled from the two controllers at different states, e.g.,

$$\mathop{\mathrm{E}}_{x\sim d^{\theta'}} \left[\mathrm{KL}(u_\theta(\cdot \mid x), u_{\theta'}(\cdot \mid x))\right].$$

This gives us the TRPO objective. At each iteration with weights θ', it solves:

$$\begin{aligned} &\max_{\theta} \mathop{\mathrm{E}}_{x\sim d^{\theta'}} \mathop{\mathrm{E}}_{u\sim u_\theta} \left[a^{\theta'}(x,u)\right] \\ &\text{subject to } \mathop{\mathrm{E}}_{x\sim d^{\theta'}} \left[\mathrm{KL}(u_\theta(\cdot \mid x), u_{\theta'}(\cdot \mid x))\right] \leq \lambda. \end{aligned} \tag{8.26}$$

to obtain updated parameters.

> How would we implement this objective? We would first sample trajectories using $u_{\theta'}$, to calculate the state-visitation frequency $d^{\theta'}$. Then at each state we should calculate the controls taken by the new controller u_θ. You should immediately see a problem with this... The new controls will send the system to different states that those in our trajectories from $u_{\theta'}$. This would not be a problem if we were calculating the state-visitation frequency exactly, but since we only use a few trajectories, we would not know how likely the new state was under the old controller.

To work around this issue, we can use importance sampling which we have seen before in particle filtering. We rewrite the expectation using the

ℹ We can also implement TRPO in (8.26) using Levenberg-Marquardt to obtain the trust region instead of a constraint on the changes to the controller. We need to use a trust-region formed by a non-isotropic ball in the weight space of the controller instead of a simple λI because the output of the controller, namely the state-visitation frequency and the controller's probabilities have very different sensitivities to changes in different parameters of the neural network. Instead of worrying about what this non-isotropic ball is in the weight space, it is easy to control the step directly in the controller. The two are equivalent.

ratio of the probabilities of the two controllers:

$$\max_{\theta} \underbrace{\mathop{\mathrm{E}}_{x\sim p_{\theta'}} \mathop{\mathrm{E}}_{u\sim u_{\theta'}}}_{\text{old controller}} \left[\underbrace{\frac{u_\theta(u \mid x)}{u_{\theta'}(u \mid x)}}_{\text{importance ratio}} a^{\theta'}(x,u) \right]. \tag{8.27}$$

8.6.3 Proximal policy optimization (PPO)

TRPO was based on sound ideas and it also gives a proof that the value of the controller improves monotonically under certain conditions. The authors in the popular paper titled "Proximal Policy Optimization Algorithms" observed that this was not true in practice on some problems.

From our description of the algorithm, it should be clear when TRPO cannot make monotonic improvements upon the value. If the trust region parameter λ in (8.26) is too large, then we allow large changes to the controller. This can lead to very large differences in the state-visitation frequency $d^{\theta'}$ vs. d^{θ}. And all our mathematics of local approximations breaks down. If we use too small a parameter λ then the new controller is not allowed to be much different than the old one...and therefore the algorithm might appear slow in making progress. In Levenberg-Marquardt-type methods, the workaround to this issue is well-known. One should modify the parameter λ during the course of optimization adaptively—we cannot have it be fixed for the entire duration.

PPO implements two tricks to improve the practical performance of TRPO.

1. The first observation is that the importance ratio should be close to 1. If it is too large or too small, this indicates that the two controllers have very different probabilities for the controls taken by the old controller (and thereby the new controller's state-visitation frequency is very different...). We therefore use a parameter ϵ to clip the likelihood-ratio to satisfy

$$1 - \epsilon \leq \frac{u_\theta(u \mid x)}{u_{\theta'}(u \mid x)} \leq 1 + \epsilon.$$

2. The trust region parameter λ is changed during the course of optimization. If the change in the controller after the weight update

$$\Delta = \mathop{\mathrm{E}}_{x\sim d^{\theta^{(t)}}} \left[\mathrm{KL}(u_{\theta^{(t+1)}}(\cdot \mid x), u_{\theta^{(t)}}(\cdot \mid x)) \right]$$

is very small, say $\Delta < \Delta_{\text{user}}/2$, then we use 2λ for the next iteration, and if this change is too big, say $\Delta > 2\Delta_{\text{user}}$, then we decrease the trust region to set $\lambda/2$ for the next iteration. These parameters should be chosen carefully by watching the different parts of training objective. For example, scipy will implement a lot of careful choices to modify the Levenberg-Marquardt parameter and that is why it gives good answers for many nonlinear optimization algorithms.

The original PPO paper implements a few different things on top of this,

e.g., generalized advantage estimation, but many people have noticed that these tricks are not important.

PPO is a very good algorithm for reinforcement learning and should be the "first" thing you try on a new problem. As we have seen in this chapter, it implements some very classical ideas. Unfortunately, when it comes to policy gradient methods, we need all the techniques discussed in this chapter to get meaningful answers, e.g., actor-critic methods, calculating advantage to do variance reduction, using Adam to update the weights of the neural networks, and using trust-region methods like PPO to select step-sizes. There is a lot of good code available to implement these things, e.g., Open AI's Spinning Up https://github.com/openai/spinningup is a very good resource to study how to implement RL algorithms in < 500 lines of code. There are a lot of other fancier libraries but they do not work any better than this simple one.

Chapter 9

Q-Learning

Reading

1. Sutton & Barto, Chapter 6, 11
2. Human-level control through deep reinforcement learning https://www.nature.com/articles/nature14236
3. Deterministic Policy Gradient Algorithms, http://proceedings.mlr.press/v32/silver14.html
4. Addressing Function Approximation Error in Actor-Critic Methods https://arxiv.org/abs/1802.09477
5. An Application of Reinforcement Learning to Aerobatic Helicopter Flight, https://papers.nips.cc/paper/3151-an-application-of-reinforcement-learning-to-aerobatic-helicopter-flight

In the previous chapter, we looked at what are called "on-policy" methods, these are methods where the current controller $u_{\theta^{(k)}}$ is used to draw fresh data from the dynamical system and used to update to parameters $\theta^{(k)}$. *The key inefficiency in on-policy methods is that this data is thrown away in the next iteration.* We need to draw a fresh set of trajectories from the system for $u_{\theta^{(k+1)}}$. This lecture will discuss off-policy methods which are a way to reuse past data. These methods require much fewer data than on-policy methods (in practice, about 10–100$\times$ less).

9.1 Tabular Q-Learning

Recall the value iteration algorithm for discrete (and finite) state and control spaces; this is also called "tabular" Q-Learning in the RL literature because we can store the Q-function $q(x, u)$ as a large table with number of rows being the number of states and number of columns being the

number of controls, with each entry in this table being the value $q(x, u)$. Value iteration when written using the Q-function at the k^{th} iteration for the tabular setting looks like

$$\begin{aligned} q^{(k+1)}(x,u) &= \sum_{x' \in X} \mathrm{P}(x' \mid x, u) \left(r(x,u) + \gamma \max_{u'} q^{(k)}(x', u') \right) \\ &= \mathop{\mathrm{E}}_{x' \sim \mathrm{P}(\cdot \mid x, u)} \left[r(x,u) + \gamma \max_{u'} q^{(k)}(x', u') \right]. \end{aligned}$$

> In the simplest possible instantiation of Q-learning, the expectation in the value iteration above (which we can only compute if we know a model of the dynamics) is replaced by samples drawn from the environment.

We will imagine the robot as using an *arbitrary* controller

$$u_e(\cdot \mid x)$$

that has a fairly large degree of randomness in how it picks actions. We call such a controller an "exploratory controller". Conceptually, its goal is to lead the robot to diverse states in the state-space so that we get a faithful estimate of the expectation in value iteration. We maintain the value $q^{(k)}(x, u)$ for all states $x \in X$ and controls $u \in U$ and update these values to get $q^{(k+1)}$ after *each step* of the robot.

From the results on Bellman equation, we know that any Q-function that satisfies the above equation is the optimal Q-function; we would therefore like our Q-function to satisfy

$$q^*(x_k, u_k) \approx r(x_k, u_k) + \gamma \max_{u'} q^*(x_{k+1}, u').$$

over samples (x_k, u_k, x_{k+1}) collected as the robot explores the environment.

> **Tabular Q-Learning** Let us imagine the robot travels for n trajectories of T time-steps each. We can now solve for q^* by minimizing the objective
>
> $$\min_q \frac{1}{n(T+1)} \sum_{i=1}^{n} \sum_{k=0}^{T} \left(q(x_k^i, u_k^i) - r(x_k^i, u_k^i) - \gamma \max_{u' \in U} \; q(x_{k+1}^i, u') \right)^2. \tag{9.1}$$
>
> on the data collected by the robot. The variable of optimization here are all values $q^*(x, u)$ for $x \in X$ and $u \in U$.

Notice a few important things about the above optimization problem. First, the last term is a maximization over $u' \in U$, it is $\max_{u' \in U} q(x_{k+1}^i, u')$ and not $q(x_{k+1}^i, u_{k+1}^i)$. In practice, you should imagine a robot performing

Q-Learning in a grid-world setting where it seeks to find the optimal trajectory to go from a source location to a target location. If at each step, the robot has 4 controls to choose from, computing this last term involves taking the maximum of 4 different values (4 columns in the tabular Q-function).

Notice that for finite-horizon dynamic programming we initialized the Q-function at the terminal time to a known value (the terminal cost). Similarly, for infinite-horizon value iteration, we discussed how we can converge to the optimal Q-function with any initialization. In the above case, we do not impose any such constraint upon the Q-function, but there is an implicit constraint. All values $q(x, u)$ have to be consistent with each other and ideally, the residual

$$q(x_k^i, u_k^i) - r(x_k^i, u_k^i) - \gamma \max_{u' \in U} \; q(x_{k+1}^i, u') = 0$$

for all trajectories i and all timesteps T.

Solving tabular Q-Learning How should we solve the optimization problem in (9.1)? This is easy, every entry $q(x, u)$ for $x \in U$ and $u \in U$ is a variable of this objective and each $(\cdot)^2$ term in the objective simply represents a constraint that ties these different values of the Q-function together. We can solve for all $q(x, u)$ iteratively as

$$\begin{aligned} q(x,u) \; &\leftarrow \; q(x,u) - \eta \nabla_{q(x,u)} \ell(q) \\ &= (1-\eta)\, q(x,u) - \eta \left(r(x,u) + \gamma \max_{u'} q(x', u') \right) \end{aligned} \tag{9.2}$$

where $\ell(q)$ is the entire objective $\frac{1}{n(T+1)} \sum_i \sum_k \cdots$ above and $(x, u, x') \equiv (x_k^i, u_k^i, x_{k+1}^i)$ in the second equation. An important point to note here is that although the robot collects a finite number of data

$$D = \left\{ (x_k^i, u_k^i)_{k=0,1,\ldots,T} \right\}_{i=1}^n$$

we have an estimate for the value $q(x, u)$ at all states $x \in X$. As an intuition, tabular Q-learning looks at the returns obtained by the robot after starting from a state x (the reward-to-come $J(x)$) and patches the returns from nearby states x, x' using the constraints in the objective (9.1).

Terminal state One must be very careful about the terminal state in such implementations of Q-learning. Typically, most research papers imagine that they are solving an infinite horizon problem but use simulators that have an explicit terminal state, i.e., the simulator does not proceed to the next timestep after the robot reaches the goal. A workaround for using such simulators (essentially all simulators) is to modify (9.2) as

$$q(x,u) = (1-\eta)\, q(x,u) - \eta \left(r(x,u) + \gamma \left(1 - \mathbf{1}_{\{x' \text{ is terminal}\}}\right) \max_{u'} q(x', u') \right).$$

Effectively, we are setting $q(x', u) = 0$ for all $u \in U$ if x' is a terminal state of problem. This is a very important point to remember and Q-Learning will never work if you forget to include the term $\mathbf{1}_{\{x' \text{ is terminal}\}}$ in your expression.

What is the controller in tabular Q-Learning? The controller in tabular Q-Learning is easy to get after we solve (9.1). At test time, we use a deterministic controller given by

$$u^*(x) = \operatorname*{argmax}_{u'} q^*(x, u').$$

9.1.1 How to perform exploration in Q-Learning

The exploratory controller used by the robot $u_e(\cdot \mid x)$ is critical to perform Q-Learning well. If the exploratory controller does not explore much, we do not get states from all parts of the state-space. This is quite bad, because in this case the estimates of Q-function at *all states* will be bad, not just at the states that the robot did not visit. To make this intuitive, imagine if we cordoned off some nodes in the graph for the backward version of Dijkstra's algorithm and never used them to update the dist variable. We would never get to the optimal cost-to-go for *all* states in this case because there could be trajectories that go through these cordoned off states that lead to a smaller cost-to-go. So it is quite important to pick the right exploratory controller.

It turns out that a random exploratory controller, e.g., a controller $u_e(\cdot \mid x)$ that picks controls uniformly randomly is pretty good. We can show that our tabular Q-Learning will converge to the optimal Q-function $q^*(x, u)$ as the amount of data drawn from the random controller goes to infinity, even if we initialize the table to arbitrary values. In other words, if we are guaranteed that the robot visits each state in the finite MDP infinitely often, it is a classical result that updates of the form (9.2) for minimizing the objective in (9.1) converge to the optimal Q-function.

ℹ This is again the power of dynamic programming at work. The Bellman equation guarantees the convergence of value iteration provided we compute the expectation exactly. But if the robot does give us lots of data from the environment, then Q-Learning also inherits this property of convergence to the optimal Q-function from any initialization.

Epsilon-greedy exploration Instead of the robot using a arbitrary controller $u_e(\cdot \mid x)$ to gather data, we can use the current estimate of the Q-function with some added randomness to ensure that the robot visits all states in the state-space. This is a key idea in Q-Learning and is known as "epsilon-greedy" exploration. We set

$$u_e(u \mid x) = \begin{cases} \operatorname{argmax}_u q(x, u) & \text{with probability } 1 - \epsilon \\ \text{uniform}(U) & \text{with probability } \epsilon. \end{cases} \tag{9.3}$$

for some user-chosen value of ϵ. Effectively, the robot repeats the controls it took in the past with probability $1 - \epsilon$ and uniformly samples from the entire control space with probability ϵ. The former ensures that the robot moves towards the parts of the state-space where states have a high return-to-come (after all, that is the what the Q-function $q(x, u)$ indicates).

The latter ensures that even if the robot's estimate of the Q-function is bad, it is still visiting every state in the state-space infinitely often.

A different perspective on Q-Learning Conceptually, we can think of tabular Q-learning as happening in two stages. In the first stage, the robot gathers a large amount of data

$$D = \left\{(x_k^i, u_k^i)_{k=0,1,\ldots,T}\right\}_{i=1}^n$$

using the exploratory controller $u_e(\cdot \mid x)$; let us consider the case when we are using an arbitrary exploratory controller, not epsilon-greedy exploration. Using this data, the robot fits a model for the system, i.e., it learns the underlying MDP

$$\mathrm{P}(x' \mid x, u);$$

this is very similar to the step in the Baum-Welch algorithm that we saw for learning the Markov state transition matrix of the HMM in Chapter 2. We simply take frequency counts to estimate this probability

$$\mathrm{P}(x' \mid x, u) \approx \frac{1}{N} \sum_i \mathbf{1}_{\{x' \text{ was reached from } x \text{ using control } u\}}$$

where N is the number of the times the robot took control u at state x. Given this transition matrix, we can now perform value iteration on the MDP to learn the Q-function

$$q^{(k+1)}(x, u) = \mathop{\mathrm{E}}_{x' \sim \mathrm{P}(\cdot|x,u)} \left[r(x, u) + \gamma \max_{u'} q^{(k)}(x', u) \right].$$

The success of this two-stage approach depends upon how accurate our estimate of $\mathrm{P}(x' \mid x, u)$ is. This in turn depends on how much the robot explored the domain and the size of the dataset it collected, both of these need to be large. We can therefore think of Q-learning as interleaving these two stages in a single algorithm, it learns the dynamics of the system and the Q-function for that dynamics simultaneously. But the Q-Learning algorithm does not really maintain a representation of the dynamics, i.e., at the end of running Q-Learning, we do not know what $\mathrm{P}(x' \mid x, u)$ is.

9.2 Function approximation (Deep Q Networks)

Tabular methods are really nice but they do not scale to large problems. The grid-world in the homework problem on policy iteration had 100 states, a typical game of Tetris has about 10^{60} states. For comparison, the number of atoms in the known universe is about 10^{80}. The number of different states in a typical Atari game is more than 10^{300}. These are all problems with a discrete number of states and controls, for continuous state/control-space, the number of distinct states/controls is infinite. So it is essentially impossible to run the tabular Q-Learning method from

the previous section for most real-world problems. In this section, we will look at a powerful set of algorithms that parameterize the Q-function using a neural network to work around this problem.

We use the same idea from the previous chapter, that of parameterizing the Q-function using a deep network. We will denote

$$q_\varphi(x, u) : X \times U \mapsto \mathbb{R}$$

as the Q-function and our goal is to fit the deep network to obtain the weights $\hat{\varphi}$, instead of maintaining a very large table of size $|X| \times |U|$ for the Q-function. Fitting the Q-function is quite similar to the tabular case: given a dataset $D = \left\{(x_t^i, u_t^i)_{t=0,1,\ldots,T}\right\}_{i=1}^n$ from the system, we want to enforce

$$q_\varphi(x_t^i, u_t^i) = r(x_t^i, u_t^i) + \gamma \max_{u'} q_\varphi(x_{t+1}^i, u')$$

for all tuples $(x_t^i, u_t^i, x_{t+1}^i)$ in the dataset. Just like the previous section, we will solve

$$\begin{aligned}\hat{\varphi} &= \operatorname*{argmin}_\varphi \frac{1}{n(T+1)} \sum_{i=1}^n \sum_{t=1}^T \left(q_\varphi(x_t^i, u_t^i) - \underbrace{r(x_t^i, u_t^i) - \gamma \left(1 - \mathbf{1}_{\left\{x_{t+1}^i \text{ is terminal}\right\}}\right) \max_{u'} q_\varphi(x_{t+1}^i, u')}_{\text{target}(x';\varphi)} \right)^2 \\ &\equiv \operatorname*{argmin}_\varphi \frac{1}{n(T+1)} \sum_{i=1}^n \sum_{t=1}^T \left(q_\varphi(x_t^i, u_t^i) - \text{target}(x_{t+1}^i; \varphi)\right)^2\end{aligned} \tag{9.4}$$

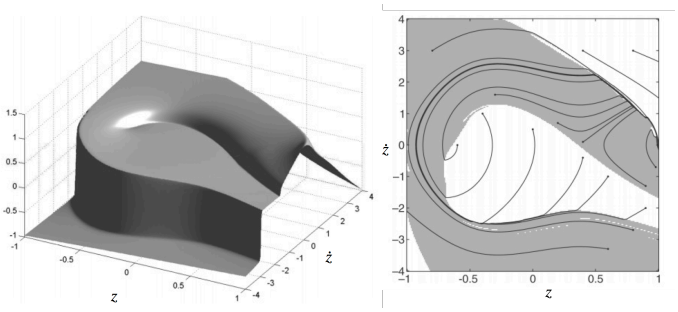
The last two terms in this expression above are together called the "target" because the problem is very similar to least squares regression, except that the targets also depend on the weights φ. This is what makes it challenging to solve.

As discussed above, Q-Learning with function approximation is known as "Fitted Q Iteration". Remember that very important point that the robot collects data using the exploratory controller $u_e(\cdot \mid x)$ but the Q-function that we fit is the *optimal* Q-function.

Fitted Q-Iteration with function approximation may not converge to the optimal Q-function It turns out that (9.4) has certain mathematical intricacies that prevent it from converging to the optimal Q-function. We will give the intuitive reason here. In the tabular Q-Learning setting, if we modify some entry $q(x, u)$ for an $x \in X$ and $u \in U$, the other entires (which are tied together using the Bellman equation) are all modified. This is akin to you changing the dist value of one node in Dijkstra's algorithm; the dist values of *all* other nodes will have to change to satisfy the Bellman equation. This is what (9.2) achieves if implemented with a decaying step-size η; see http://users.isr.ist.utl.pt/∼mtjspaan/readingGroup/ProofQlearning.pdf for the proof. This does not hold for (9.4). Even if the objective in (9.4) is zero on our collected dataset, i.e., the Q-function fits data collected by the robot perfectly, the Q-function may not be the *optimal* Q-function. An

ℹ The mathematical reason behind this is that the Bellman operator, i.e., the update to the Q/value-function is a contraction for the tabular setting, this is not the case for Fitted Q-Iteration unless the function approximation has some technical conditions imposed upon it.

intuitive way of understanding this problem is that even if the Bellman error is zero on samples in the dataset, the optimization objective says nothing about states that are not present in the dataset; the Bellman error on them is completely dependent upon the smoothness properties of the function expressed by the neural architecture. Contrast this comment with the solution of the HJB equation in Chapter 6 where the value function was quite non-smooth at some places. If our sampled dataset does not contain those places, there is no way the neural network can know the optimal form of the value function.



9.2.1 Embellishments to Q-Learning

We next discuss a few practical aspects of implementing Q-Learning. Each of the following points is extremely important to understand how to get RL to work on real-world problems, so you should internalize these.

Pick mini-batches from different trajectories in SGD . In practice, we fit the Q-function using stochastic gradient descent. At each iteration we sample a mini-batch of inputs $(x_t^i, u_t^i, x_{t+1}^i)$ from different trajectories $i \in \{1, \ldots, n\}$ and update the weights φ in the direction of the negative gradient.

$$\varphi^{k+1} = \varphi^k - \eta \nabla_\varphi \left(q_{p^k}(x, u) - \text{target}(x'; \varphi^k) \right)^2 .$$

The mini-batch is picked to have samples from different trajectories because samples from the same trajectory are correlated to each other (after all, the robot obtains the next tuple (x', u', x'') from the previous tuple (x, u, x')).

Replay buffer The dataset D is known as the replay buffer.

Off-policy learning The replay buffer is typically not fixed during training. Instead of drawing data from the exploratory controller u_e, we can think of the following algorithm. Initialize the Q-function weights to φ^0 and the dataset to $D = \emptyset$. At the k^{th} iteration,

- Draw a dataset D^k of n trajectories from the ϵ-greedy policy

$$u_e(u \mid x) = \begin{cases} \text{argmax}_u \, q^k(x, u) & \text{with probability } 1 - \epsilon \\ \text{uniform}(U) & \text{with probability } \epsilon. \end{cases}$$

- Add new trajectories to the dataset

$$D \leftarrow D \cup D^k.$$

- Update weights to q^{k+1} using all past data D using (9.4).

Compare this algorithm to policy-gradient-based methods which throw away the data from the previous iteration. Indeed, when we want to compute the gradient $\nabla_\theta \mathrm{E}_{\tau \sim p_{\theta^k}}[R(\tau)]$, we should sample trajectories from current weights θ^k, we cannot use trajectories from some old weights. In contrast, in Q-Learning, we maintain a cumulative dataset D that contains trajectories from all the past ϵ-greedy controllers and use it to find new weights of the Q-function. We can do so because of the powerful Bellman equation, Q-Iteration is learning the *optimal* value function and no matter what dataset (9.4) is evaluated upon, if the error is zero, we are guaranteed that Q-function learned is the optimal one. This is also the reason Q-Learning with a replay buffer is called "off-policy" learning because it learns the optimal controller even if the data that it uses comes from some other non-optimal controller (the exploratory controller or the ϵ-greedy controller).

Using off-policy learning is an old idea, the DQN paper which demonstrated very impressive results on Atari games using RL brought it back into prominence.

Setting a good value of ϵ for exploration is critical Towards the beginning of training, we want a large value for ϵ to gather diverse data from the environment. As training progresses, we want to reduce ϵ because presumably we have a few good control trajectories that result in good returns and can focus on searching the neighborhood of these trajectories.

Prioritized experience replay is an idea where instead of sampling from the replay buffer D uniformly randomly when we fit the Q-function in (9.4), we only sample data points (x_t^i, u_t^i) which have a high Bellman error

$$\left| q_\varphi(x_t^i, u_t^i) - r(x_t^i, u_t^i) - \gamma \left(1 - \mathbf{1}_{\{x_{t+1}^i \text{ is terminal}\}}\right) \max_{u'} q_\varphi(x_{t+1}^i, u') \right|$$

This is a reasonable idea but is not very useful in practice for two reasons. First, if we use deep networks for parameterizing the Q-function, the network *can* fit even very complex datasets so there is no reason to not use the data points with low Bellman error in (9.4); the gradient using them will be small anyway. Second, there are a lot of hyper-parameters that determine prioritized sampling, e.g., the threshold beyond which we consider the Bellman error to be high. These hyper-parameters are quite difficult to use in practice and therefore it is a good idea to not use prioritized experience replay at the beginning of development of your method on a new problem.

Using robust regression to fit the Q-function There may be states in the replay buffer with very high Bellman error, e.g., the kinks in the value function for the mountain car obtained from HJB above, if we happen to sample those. For instance, these are states where the controller "switches" and is discontinuous function of state x. In these cases, instead of these few states dominating the gradient for the entire dataset, we can use ideas in robust regression to reduce their effect on the gradient. A popular way to do so is to use a Huber-loss in place of the quadratic loss in (9.4)

$$\text{huber}_\delta(a) = \begin{cases} \frac{a^2}{2} & \text{for } |a| \leq \delta \\ \delta\left(|a| - \frac{\delta}{2}\right) & \text{otherwise.} \end{cases} \tag{9.5}$$

Delayed target Notice that the target also depends upon the weights φ:

$$\text{target}(x'; \varphi) := r(x, u) + \gamma \left(1 - \mathbf{1}_{\{x' \text{ is terminal}\}}\right) \max_{u'} q_\varphi(x', u').$$

This creates a very big problem when we fit the Q-function. Effectively, both the covariate and the target in (9.4) depend upon the weights of the Q-function. Minimizing the objective in (9.4) is akin to performing least squares regression where the targets keep changing every time you solve for the solution. This is the root cause of why Q-Learning is difficult to use in practice. A popular hack to get around this problem is to use some old weights to compute the target, i.e., use the loss

$$\frac{1}{n(T+1)} \sum_{i,t} \left(q_{\varphi^k}(x_t^i, u_t^i) - \text{target}(x_{t+1}^i;\ \varphi^{k'})\right)^2. \tag{9.6}$$

in place of (9.4). Here k' is an iterate much older than k, say $k' = k - 100$. This trick is called "delayed target".

Exponential averaging to update the target Notice that in order to implement delayed targets as discussed above we will have to save all weights $\varphi^k, \varphi^{k-1}, \ldots, \varphi^{k-100}$, which can be cumbersome. We can however do yet another clever hack and initialize two copies of the weights, one for the actual Q-function φ^k and another for the target, let us call it φ'^k. We set the target equal to the Q-function at initialization. The target copy is updated at each iteration to be

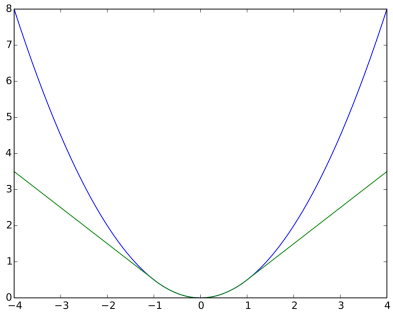
$$\varphi'^{k+1} = (1 - \alpha)\varphi'^k + \alpha\varphi^{k+1} \tag{9.7}$$

with some small value, say $\alpha = 0.05$. The target's weights are therefore an exponentially averaged version of the weights of the Q-function.

Why are delayed targets essential for Q-Learning to work? There are many explanations given why delayed targets are essential in practice but the correct one is not really known yet.

1. For example, one reason could be that since $q_{\varphi^k}(x, u)$ for a given state typically increases as we train for more iterations in Q-Learning,

Huber loss for $\delta = 1$ (green) compared to the squared error loss (blue).



the old weights inside a delayed target are an underestimate of the true target. This might lead to some stability in situations when the Q-function's weights φ^k change too quickly when we fit (9.4) or we do not have enough data in the replay buffer yet.

2. Another reason one could hypothesize is related to concepts like self-distillation. For example, we may write a new objective for Q-Learning that looks like

$$\left(q_{\varphi^k}(x_t^i, u_t^i) - \text{target}(x_{t+1}^i;\ \varphi^k)\right)^2 + \frac{1}{2\lambda}\left\|\varphi^k - \varphi^{k'}\right\|_2^2$$

where the second term is known as proximal term that prevents the weights φ^k from change too much from their old values $\varphi^{k'}$. Proximal objectives are more stable versions of the standard quadratic objective in (9.4) and help in cases when one is solving Q-Learning using SGD updates because they are stable to step-sizes.

Double Q-Learning Even a delayed target may not be sufficient to get Q-Learning to lead to good returns in practice. Focus on one state x. One problem arise from the max operator in (9.4). Suppose that the Q-function q_{φ^k} corresponds to a particularly bad controller, say a controller that picks a control

$$\operatorname*{argmax}_u q_{\varphi^k}(x, u)$$

that is very different from the optimal control

$$\operatorname*{argmax}_u q^*(x, u)$$

then, even the delayed target $q_{\varphi^{k'}}$ may be a similarly poor controller. The ideal target is of course the return-to-come, or the value of the optimal Q-function $\max_{u'} q^*(x', u')$, but we do not know it while fitting the Q-function. The same problem also occurs if our Q-function (or its delayed version, the target) is too optimistic about the values of certain control inputs, it will consistently pick those controls in the max operator. One hack to get around this problem is to pick the maximizing control input using the non-delayed Q-function but use the value of the delayed target

$$\text{target}_{\text{DDQN}}(x_{t+1}^i; \varphi'^k) = r(x, u) + \gamma \left(1 - \mathbf{1}_{\left\{x_{t+1}^i \text{ is terminal}\right\}}\right) q_{\varphi'^k}(x_{t+1}^i, u'). \tag{9.8}$$

where

$$u' = \underbrace{\operatorname*{argmax}_u q_{\varphi^k}(x_{t+1}^i, u)}_{\text{control chosen by the Q-function}}.$$

Training two Q-functions We can also train two copies of the Q-function simultaneously, each with its own delayed target and mix-and-match their targets. Let ${\varphi^{(1)}}^k$ and ${\varphi'^{(1)}}^k$ be one Q-function and target pair and ${\varphi^{(2)}}^k$ and ${\varphi'^{(2)}}^k$ be another pair. We update both of them using the following

objective.

$$\text{For } \varphi^{(1)}: \left(q^{(1)^k}(x,u) - r(x,u) - \gamma\left(1 - \mathbf{1}_{\{x' \text{ is terminal}\}}\right) \text{target}_{\text{DDQN}}(x', \varphi'^{(2)^k})\right)^2$$
$$\text{For } \varphi^{(2)}: \left(q^{(2)^k}(x,u) - r(x,u) - \gamma\left(1 - \mathbf{1}_{\{x' \text{ is terminal}\}}\right) \text{target}_{\text{DDQN}}(x', \varphi'^{(1)^k})\right)^2 \tag{9.9}$$

Sometimes we also use only one target that is the minimum of the two targets (this helps because it is more pessimistic estimate of the true target)

$$\text{target}(x') := \min\left\{\text{target}_{\text{DDQN}}(x', \varphi'^{(1)^k}), \text{target}_{\text{DDQN}}(x', \varphi'^{(2)^k})\right\}.$$

You will also see many papers train multiple Q-functions, many more than 2. In such cases, it is a good idea to pick the control for evaluation using all the Q-functions:

$$u^*(x) := \underset{u}{\text{argmax}} \sum_k q_{\varphi^{(k)}}(x,u).$$

rather than only one of them, as is often done in research papers.

A remark on the various tricks used to compute the target It may seem that a lot of these tricks are about being pessimistic while computing the target. This is our current understanding in RL and it is born out of the following observation: typically in practice, you will observe that the Q-function estimates can become very large. Even if the TD error is small, the values $q_\varphi(x,u)$ can be arbitrarily large; see Figure 1 in Continuous Doubly Constrained Batch Reinforcement Learning for an example in a slightly different setting. This occurs because we pick the control that maximizes the Q-value of a particular state x in (9.8). Effectively, if the Q-value $q_\varphi(x',u)$ of a particular control $u \in U$ is an over-estimate, the target will keep selecting this control as the maximizing control, which drives up the value of the Q-function at $q_\varphi(x,u)$ as well. This problem is a bit more drastic in the next section on continuous-valued controls. It is however unclear how to best address this issue and design mathematically sound methods that do not use arbitrary heuristics such as "pessimism".

ℹ Mathematically, the fundamental problem in function-approximation-based RL is actually clear: even if the Bellman operation is a contraction for tabular RL, it need not be a contraction when we are approximating the Q-function using a neural network. Therefore minimizing TD-error which works quite well for the tabular case need not work well in the function-approximation case. There may exist other, more robust, ways of computing the Bellman fixed point $q_\varphi(x,u) = r(x,u) + \max_{u'} \gamma\, q_\varphi(x',u')$ other than minimizing the the squared TD error but we do not have good candidates yet.

9.3 Q-Learning for continuous control spaces

All the methods we have looked at in this chapter are for discrete control spaces, i.e., the set of controls that the robot can take is a finite set. In this case we can easily compute the maximizing control of the Q-function.

$$u^*(x) = \underset{u}{\text{argmax}}\, q_\varphi(x,u).$$

Certainly a lot of real-world problems have continuous-valued controls and we therefore need Q-Learning-based methods to handle this.

Deterministic policy gradient A natural way, although non-rigorous, to think about this is to assume that we are given a Q-function $q_\varphi(x, u)$ (we will leave the controller for which this is the Q-function vague for now) and a dataset $D = \left\{(x_t^i, u_t^i)_{t=0}^T\right\}_{i=1}^n$. We can find a deterministic feedback controller that takes controls that lead to good values as

$$\theta^* = \max_\theta \frac{1}{n(T+1)} \sum_{i=1}^n \sum_{t=0}^T q_\varphi(x_t^i, u_\theta(x_t^i)). \tag{9.10}$$

Effectively we are fitting a feedback controller that takes controls $u_{\theta^*}(x)$ that are the maximizers of the Q-function. This is a natural analogue of the argmax over controls for discrete/finite control spaces. Again we should think of having a deep network that parametrizes the deterministic controller and fitting its parameters θ using stochastic gradient descent on (9.10)

$$\begin{aligned} \theta^{k+1} &= \theta^k + \eta \nabla_\theta q_\varphi(x^\omega, u_{\theta^k}(x^\omega)) \\ &= \theta^k + \eta \left(\nabla_u q_\varphi(x^\omega, u)\right) \left(\nabla_\theta u_{\theta^k}(x^\omega)\right) \end{aligned} \tag{9.11}$$

where ω is the index of the datum in the dataset D. The equality was obtained by applying the chain rule. This result is called the "deterministic policy gradient" and we should think of it as the limit of the policy gradient for a stochastic controller as the stochasticity goes to zero. Also notice that the term

$$\nabla_u q_\varphi(x^\omega, u)$$

is the gradient of the output of the Q-function $q_\varphi : X \times U \mapsto \mathbb{R}$ with respect to its second input u. Such gradients can also be easily computed using backpropagation in PyTorch. It is different than the gradient of the output with respect to its weights

$$\nabla_\varphi q_\varphi(x^\omega, u).$$

On-policy deterministic actor-critic Let us now construct an analogue of the policy gradient method for the case of a deterministic controller. The algorithm would proceed as follows. We initialize weights of a Q-function φ^0 and weights of the deterministic controller θ^0.

1. At the k^{th} iteration, we collect a dataset from the robot using the latest controller u_{θ^k}. Let this dataset be D^k that consists of tuples (x, u, x', u').

2. Fit a Q-function q^{θ^k} to this dataset by minimizing the temporal difference error

$$\varphi^{k+1} = \operatorname*{argmin}_\varphi \sum_{(x,u,x',u') \in D^k} \left(q_\varphi(x, u) - r(x, u) - \gamma \left(1 - \mathbf{1}_{\{x' \text{ is terminal}\}}\right) q_{\varphi'}(x', u')\right)^2. \tag{9.12}$$

   Notice an important difference in the expression above, instead of using $\max_u$ in the target, we are using the control that the current

controller, namely u_{θ^k} has taken. This is because we want to evaluate the controller u_{θ^k} and simply parameterize the Q-function using weights φ^{k+1}. More precisely, we hope that we have

$$q_{\varphi^{k+1}}(x, u_{\theta^k}(x)) \approx \max_u q^{\theta^k}(x, u).$$

3. We can now update the controller using this Q-function:

$$\theta^{k+1} = \theta^k + \eta \nabla_\theta q_{\varphi^{k+1}}(x^\omega, u_{\theta^k}(x^\omega)) \tag{9.13}$$

This algorithm is called "on-policy SARSA" because at each iteration we draw fresh data D^k from the environment; this is the direct analogue of actor-critic methods that we studied in the previous chapter for deterministic controllers.

ℹ SARSA is an old algorithm in RL that is the tabular version of what we did here. It stands for state-action-reward-state-action ...

Off-policy deterministic actor-critic methods We can also run the above algorithm using data from an exploratory controller. The only difference is that the we now do not throw away the data D^k from older iterations

$$D = D^1 \cup \cdots \cup D^k$$

and therefore have to change (9.12) to be

$$\varphi^{k+1} = \operatorname*{argmin}_\varphi \sum_{(x,u,x',u') \in D} \left(q_\varphi(x,u) - r(x,u) - \gamma \left(1 - \mathbf{1}_{\{x' \text{ is terminal}\}}\right) q_{\varphi'}(x', \underbrace{u_{\theta^k}(x')}_{\text{notice the difference}}) \right)^2. \tag{9.14}$$

Effectively, we are fitting the optimal Q-function using the data D but since we can no longer take the maximum over controls directly, we plug in the controller in the computation of the target. This is natural; we think of the controller as the one that maximizes the Q-function when we update (9.13). When used with deep networks, this is called the "deep deterministic policy gradient" algorithm, it is popular by the name DDPG.

Chapter 10

Model-based Reinforcement Learning

Reading

1. PILCO: A Model-Based and Data-Efficient Approach to Policy Search, http://mlg.eng.cam.ac.uk/pub/pdf/DeiRas11.pdf
2. Embed to Control: A Locally Linear Latent Dynamics Model for Control from Raw Images https://arxiv.org/abs/1506.07365
3. Deep Reinforcement Learning in a Handful of Trials using Probabilistic Dynamics Models https://arxiv.org/abs/1805.12114

We have seen a large number of methods which use a known model of the dynamical system to compute the control inputs, these include value/policy iteration, Linear Quadratic Regulator (LQR) and Model Predictive Control (MPC). We also saw a number of methods from the reinforcement learning literature that can work "model-free", i.e., having access to some data from the environment in lieu of a model. On one hand, model-based methods come with some obvious challenges, if we do not know the model of the system, the controller will not be optimal and worse, it may even be unsafe; think of driving on black ice that is a thin coat of ice which develops after repeated freezing and melting of snow on asphalt. On the other hand, model-free approaches are spectacularly inefficient: policy gradient-based methods require several thousands of trajectories to train a controller and even more efficient ones such as off-policy methods require prohibitive amounts of data; recall the example of an espresso bar in New York City that makes 50 shots a day, it takes more than a month to sample more than 1000 trajectories. This has limited the reach of model-free RL methods primarily to simulation, although there are examples where these policies were run (typically after training) in the real world

also; see another example at https://ai.googleblog.com/2020/05/agile-and-intelligent-locomotion-via.html. Very rarely you will see RL methods being used to train robots directly.

It makes sense to combine model-based and model-free methods if we want to reduce the number of data required from the system to learn a controller. Such methods are typically called model-based RL methods.

10.1 Learning a model of the dynamics

Imagine that we have a robot with dynamics

$$x_{k+1} = f(x_k, u_k)$$

and obtained some data from this robot using an exploratory controller $u_e(\cdot \mid x)$. Let us call this dataset $D = \left\{(x_t^i, u_t^i)_{t=0}^T\right\}_{i=1}^n$; it consists of n trajectories each of length T timesteps. We can fit a deep network to learn the dynamics. This involves parameterizing the unknown dynamics using a deep network with weights w

$$f_w : X \times U \mapsto \mathbb{R}$$

and minimizing a regression error of the form

$$w^* = \underset{w}{\text{argmin}} \frac{1}{n(T+1)} \sum_{i=1}^{n} \sum_{t=0}^{T} \left\| x_{t+1}^i - f_w(x_t^i, u_t^i) \right\|_2^2. \tag{10.1}$$

If the residual $\left\| x_{t+1}^i - f_w(x_t^i, u_t^i) \right\|^2$ is small on average over the dataset, then we know that given some new control $u' \neq u_t^i$, we can, for instance, estimate the new future state $x' = f_w(x, u')$. In principle, we can use this model now to execute algorithms like iterated LQR to find an optimal controller. We could also imagine using this as our own simulator for the robot, i.e., instead of drawing new trajectories in model-free RL from the actual robot, we use our learned model of the dynamics to obtain more data.

An inverse model of the dynamics We can also learn what is called the inverse model of the system that maps the current state x_t^i and the next state x_{t+1}^i to the control that takes the system from the former to the latter:

$$f_w^{\text{inv}} : X \times X \mapsto U.$$

The regression error for one sample in this case would be $\left\| u_t^i - f_w(x_t^i, x_{t+1}^i) \right\|^2$. This is often a more useful model to learn from the data, e.g., if we want to use this model in a Rapidly Exploring Random Tree (RRT), we can sample states in the configuration space of the robot and have the learned dynamics guess the control between two states. Also see the paper on contact-invariant optimization (https://homes.cs.washington.edu/ todorov/papers/-MordatchSIGGRAPH12.pdf) and a video at https://www.youtube.com/watch?v=mhr_jtQrhVA

for an impressive demonstration of using an inverse model.

Models can be wrong at parts of state-space where we have few data This is really the biggest concern with using models. We have seen in the chapter on deep learning that if we do not have data from some part of the state-space, there are few guarantees of the model f_w or f_w^{inv} working well for those states. A planning algorithm does not however *know* that the model is wrong for a given state. So the central question in learning a model is "how to estimate the uncertainty of the output of the model", i.e.,

$$\mathrm{P}(x_{k+1} \neq f_x(x_k, u_k))$$

where x_{k+1} is the true next state of the system and $f_x(x_k, u_k)$ is our prediction using the model. If we have a good estimate of such uncertainty, we can safely use the model only at parts of the state-space where this uncertainty is small.

Sequentially querying the environment for data We can use our ideas from DAgger and off-policy RL to improve our model iteratively by collecting more data using the controller that is being learned. Here is how it would work

1. Draw some data D from the system, fit a dynamics model $f_w^0(x, u)$ using (10.1). Learn a feedback controller $u^0(x)$ using any method we know so far (LQR, MPC, RL-based methods)
2. Run the learned controller $u^0(x)$ from the real system to collect more data D^1 and add it to the dataset

$$D \leftarrow D \cup D^1.$$

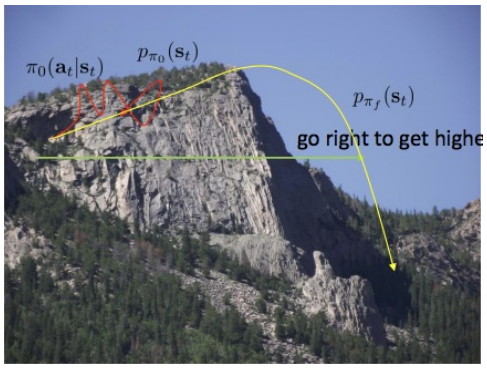
This is a simple mechanism that that ensures that we can collect more data from the system. If the controller goes to parts of the state-space that the model is incorrect at, we get samples from such regions and iteratively improve both the learned dynamics model $f_w^k(x, u)$ and the controller $u^k(x)$ using this model.

10.2 Some model-based methods

10.2.1 Bagging multiple models of the dynamics

Let us look at bagging, which is a method to estimate the uncertainty of a learning-based predictor. Bagging is short for *bootstrap aggregation*,

For a quick primer on planning using a model, see the notes at https://ocw.mit.edu/courses/aeronautics-and-astronautics/16-410-principles-of-autonomy-and-decision-making-fall-2010/lecture-notes/MIT16_410F10_lec15.pdf from Emilio Frazzoli (ETH/MIT).



and can be explained using a simple experiment. Suppose we wanted to estimate the average height μ of people in the world. We can measure the height of N individuals and obtain *one* estimate of the mean μ. This is of course unsatisfying because we know that our answer is unlikely to be the mean of the entire population. Bootstrapping computes multiple estimates of the mean μ_k over many *subsets* of the data N and reports the answer as

$$\mu := \text{mean}(\mu_k) + \text{stddev}(\mu_k).$$

Each subset of the data is created by sampling the original data with N samples *with replacement*. This is among the most influential ideas in statistics see "Bootstrap Methods: Another Look at the Jackknife" https://projecteuclid.org/journals/annals-of-statistics/volume-7/issue-1/Bootstrap-Methods-Another-Look-at-the-Jackknife/10.1214/aos/1176344552.full because it is a very simple and general procedure to obtain the uncertainty of the estimate. Also see a very famous paper "Bagging predictors" at https://link.springer.com/article/10.1007/BF00058655 that invented random forests based on this idea.

Training an ensemble We are going to train multiple dynamics models $f_{w^k}(x, u)$ for $k \in \{1, \ldots, M\}$, one each for bootstrapped versions of the training dataset $\left\{D^1, \ldots, D^M\right\}$. Each subset D^k is built by sampling a fraction, say 60% of the data uniformly randomly from our training dataset D. In other words, the bootstrapped versions of the data are not disjoint but their union is likely to be the entire training dataset. We are going to use this ensemble as follows. For each pair (x, u), we run all models and set

$$\hat{x'} = \frac{1}{M} \sum_{k=1}^{N} f_{w^k}(x, u);$$

i.e., the ensemble predicts the next state of the robot using the mean. The important benefit of using an ensemble is that we can also get an estimate of the error in these predictions

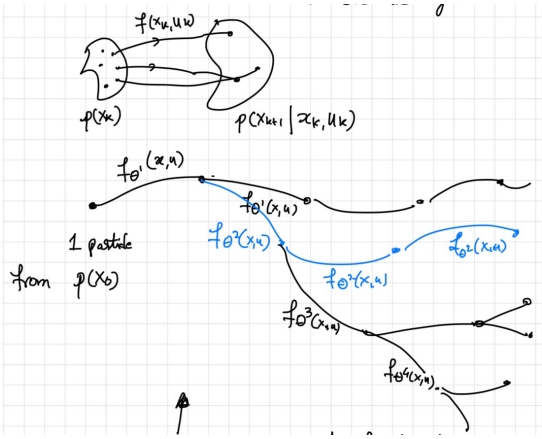
$$\text{error in } \hat{x'} = \left(\delta_k f_{w^k}(x, u)\right)^{1/2}.$$

Different members of the ensemble are training on different datasets and make different predictions as to what the next state could be. The mis-match between them is an indicator of the error in our dynamics model. This need not be an accurate estimate of the error (i.e., the difference between the predicted $\hat{x'}$ and the actual next state x' of the true dynamics) but is often a good proxy to use if we do bootstrapping.

Bagging is perhaps the most useful idea in machine learning (by far). It is always good to keep it in your mind. The winners of most high-profile machine learning competitions, e.g., the Netflix Prize (https://en.wikipedia.org/wiki/Netflix_Prize) or the ImageNet challenge, have been bagged classifiers created by fitting multiple architectures on the same dataset.

ℹ Typically, while fitting deep networks $f_{w^k}(x, u)$ using SGD, most RL papers initialize the training process at different weights and do not perform any bootstrapping. The rationale that is usually given is that since the training process is non-convex, two models initialized at different locations train to two different predictors even if they both work on the same data. Although doing this leads to *some* notion of uncertainty, it is not an entirely correct one and performing bootstrapping will always give better estimates.

How to use an ensemble for hallucinating new data The procedure is very similar to what we saw above for querying the model. The key difference is that we now have M models of the dynamics and can mix-and-match their predictions as we simulate the trajectory.



PILCO A powerful Gaussian Process-based algorithm to incorporate uncertainty in the predictions of a learned dynamics model is "PILCO: A Model-Based and Data-Efficient Approach to Policy Search" http://mlg.eng.cam.ac.uk/pub/pdf/DeiRas11.pdf. Instead of using bootstrapping of an ensemble to estimate the uncertainty, this algorithm explicitly models the uncertainty as

$$p(x_{k+1} \mid x_k, u_k) = N(x_k + \mathrm{E}[\Delta_k], \mathrm{Var}(\Delta_k))$$

where $\Delta_k = x_{k+1} - x_k$ in the training data, using a Gaussian Process.

See a preliminary but great tutorial on Gaussian Processes at https://distill.pub/2019/visual-exploration-gaussian-processes. PILCO is a complicated algorithm to implement but you can see the source code by the original authors at https://mloss.org/software/view/508. You can also look at the paper titled "BayesRace: Learning to race autonomously using prior experience" https://arxiv.org/abs/2005.04755 which uses model-based RL in

10.2.2 Model-based RL in the latent space

Chapter 11

Offline Reinforcement Learning

Reading

1. Offline Reinforcement Learning: Tutorial, Review, and Perspectives on Open Problems by Levine et al. (2020)
2. Continuous doubly constrained batch reinforcement learning by Fakoor et al. (2021)

So far, we have imagined that we have access to a model of a robot, a simulator for it, or access to the actual robot that allows us to obtain data from the robot. But there are many problems in which we cannot get any of these. For example, when Amazon sells merchandise to its customers, it is quite difficult for them to model or simulate each customer, or even a canonical one. It is possible to do rollouts using actual customer data but that would not be wide because an exploratory policy, by the time it learns, would lead to huge loses. This would also not be desirable for the customers. Amazon also needs reinforcement learning for this problem because there is not many other ways to explore what customers like and do not like. One could take another example from a very different domain. Suppose a hospital is trying to develop a new protocol to handle incoming patients. E.g., a patient comes into ER, there is a fixed set of checks that are performed quickly on them called as "triage". The number and kind of checks are performed directly affects patient care. If there are too many checks, then the patient loses precious time before they are treated. If there are too few, then there is a large bottleneck when these patients are referred to doctors. It is not easy to model or simulate this problem. It is possible to do rollouts to discover a policy but that would not be easy, or wise.

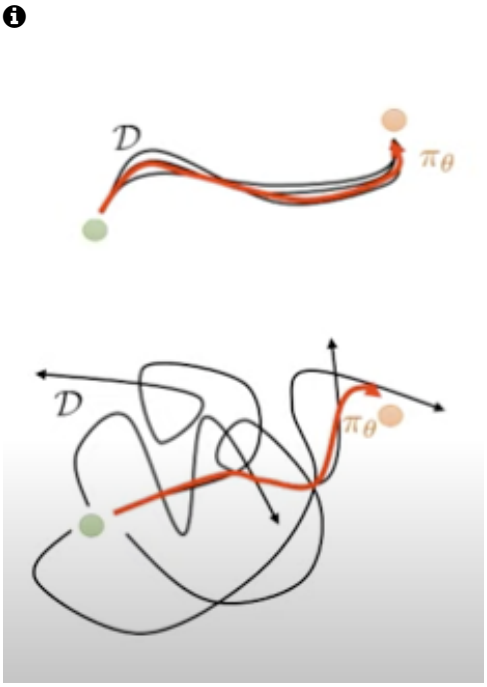
These problems have a few commonalities.

- We do not have models or simulators for such systems *and that is why* we need to use ideas from reinforcement learning to build controllers for them;
- There are both existing systems, i.e., there exists a controller that is currently deployed. This controller may be very complex, e.g., in Amazon's case it is the result of many scientists building this system over a decade (and thereby even if one could look at it, there is no way one would be able to model/simulate it). In the case of the hospital, the current triage policy was likely created by the doctors over experience and refined by the triage nurses by looking at actual cases. Because the system evolved over a long time, a lot of this knowledge is not accessible in a codified form.

> Offline reinforcement learning are a suite of techniques that allow us to learn the optimal value function from data that need not be coming from an optimal policy—without drawing any new data from the environment. Note that we do not just want to evaluate an existing policy, we want to learn the optimal value function, or at the very least improve upon the current policy.

There are many other problems of this kind: data is plentiful, just that we cannot get more.

Technically speaking, offline learning is a very clean problem in that we are close to the supervised learning setting (we do not know the true targets). A meaningful theoretical analysis of typical reinforcement learning algorithms is difficult because there are a lot of moving parts in the problem definition: exploratory controllers, the fact that we are adding correlated samples into our dataset as we draw more trajectories, function approximation properties of the neural network that does not allow Bellman iteration to remain a contraction etc. Some of these hurdles are absent in the analysis of offline learning.



A good intuition for offline reinforcement learning is given by this picture. In imitation learning, or behavior cloning, we simply want to learn a policy that mimics the expert (or recorded data). In offline reinforcement learning, we can "stitch together" multiple sub-optimal policies in different parts of the domain.

11.1 Why is offline reinforcement learning difficult?

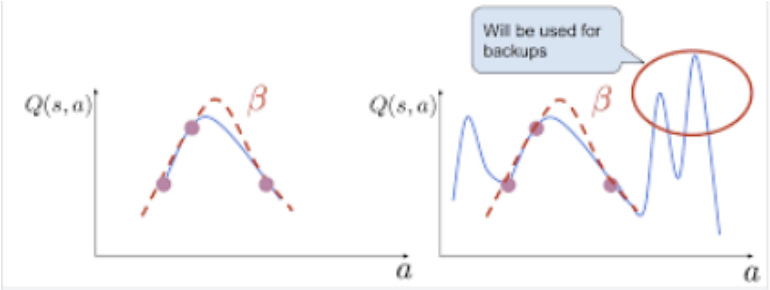
Suppose that we wanted to use behavior cloning for this problem. After all, we could build a state vector x and do behavior cloning with a neural network to learn a policy $u_\theta(x)$ from the existing dataset $D = \left\{(x_t^i, u_t^i)_{t=0,\ldots,T}\right\}_{i=1}^n$. In principle, we can also also learn the value function corresponding to this policy, $q_\varphi^\theta(x, u)$ where u is the control input. Note that this is not the optimal value function. We will call this the behavior cloning solution to offline reinforcement learning.

We know further that value iteration converges (for tabular MDPs) from any initial condition. In lieu of the actual model of the Markov Decision Process, we can imagine using the entire dataset D to calculate

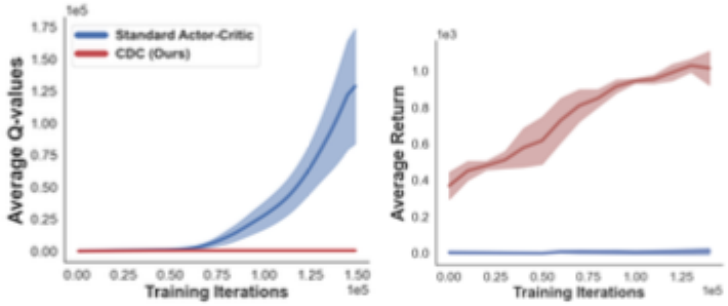
Bellman updates for a value function parameterized by a neural network:

$$\min_{\varphi} \sum_{i,t} \left(q_{\varphi}^{\theta\,(k+1)}(x_t^i, u_t^i) - r(x_t^i, u) - \gamma \max_{u' \in U} q_{\varphi}^{\theta\,(k)}(x_{t+1}^i, u') \right)^2 .$$

This approach is unlikely to work very well. Observe the following figure.



We do not really know whether the initial value function q_{φ}^{θ} assigns large returns to controls that are outside of the ones in the dataset. If it does, then these controls will be chosen in the maximization step while calculating the target. If there are states where the value function over different controls looks like this picture, then their targets will cause the value at all other states to grow unbounded during training. This is exactly what happens in practice. For example,



In offline reinforcement learning, this phenomenon is often called the "extrapolation error". It arises because we do not know a natural way to force the network to avoid predicting large values for controls that are not a part of the training dataset. It is instructive to ask why off-policy or on-policy reinforcement learning does not suffer from extrapolation error. Both of these algorithms explicitly draw more data from the simulator. If the value function were to over-estimate the value of certain control actions at a state, then the controller would take those controls during exploration and discover that the value was in fact an over-estimate. Methods that draw more data have this natural self-correcting behavior that we cannot get in offline learning.

There is a second problem associated with computing the maximum in value iteration. For problems with continuous controls, we do not know of computationally effective ways to compute the maximum $\max_{u \in U}$.

ℹ We have discussed how Bellman updates are a set of consistent constraints on the optimal value function. This entails that if we over-estimate the value at a given state, then the estimated return to come (i.e., the value) of *all* other states becomes incorrect.

Typical implementations of offline learning fit a controller that maximizes the value function

$$\theta = \text{argmax}\, \frac{1}{nT} \sum_{i,t} q^{\theta}_{\varphi}(x^i_t, u_\theta(x))$$

to make it easy to calculate this maximum. But this forces us to use another network (and it is not a given that the parameterized controller can calculate the correct maximum).

ℹ One can use a linear value function, e.g., $q^{\theta}_{\varphi}(x,u) = \langle [1,x,u], \varphi[1,x,u]\rangle$ or any other set of basis functions. But this is not enough to resolve the issue in theory. Bellman updates are not a contraction when the TD-error is projected onto a set of bases. In practice, this approach works reasonably well.

11.2 Regularized Bellman iteration

There are two broad class of techniques that are believed to give reasonable results for offline learning. These are both quite new and ad hoc and effective offline learning is essentially an open problem today. To wit, current offline learning methods not only fail to learn optimal value functions or policies from sub-optimal data, but they often do any better than behavior cloning.

11.2.1 Changing the fixed point of the Bellman iteration to be more conservative

Since extrapolation error is fundamentally caused by the value function taking large values, it is reasonable strategy to modify the Bellman updates to regularize the value function in some way. A basic version would look like a coupled system of updates

$$\begin{aligned}\varphi^* = \underset{\varphi}{\text{argmin}}\, \frac{1}{nT} \sum_{i,t} \left(q^{\theta\,(k+1)}_{\varphi}(x^i_t, u^i_t) - r(x^i_t, u) - \gamma q^{\theta\,(k)}_{\varphi}(x^i_{t+1}, u_\theta(x^i_{t+1}))\right)^2 \\ - \lambda\, \Omega(q^{\theta}_p) \\ \theta^* = \underset{\theta}{\text{argmax}}\, \frac{1}{nT} \sum_{i,t} q^{\theta}_{\varphi}(x^i_t, u_\theta(x^i_t)).\end{aligned} \tag{11.1}$$

We will use the regularizer

$$\Omega = \frac{1}{nT} \sum_{i,t} \left(q^{\theta}_{\varphi}(x^i_t, u_\theta(x^i_t)) - q^{\theta}_{\varphi}(x^i_t, u^i_t)\right)^2_+$$

The notation $(\cdot)_+$ denotes rectification, i.e., $(x)_+ = x$ if $x > 0$ and zero otherwise. Notice that second term in the objective for fitting the value function forces the value of the control $u_\theta(x^i_t)$ to be smaller than the value of the control u^i_t taken at the same state in the dataset. This is a conservative choice. While it does not fix the issue of extrapolation error, it forces the value network to predict smaller values and prevents it from blowing up.

A second, similar, strategy looks like

$$\Omega = \frac{1}{nT} \sum_{i,t} \max_{u' \sim u_\theta(x_t^i)} \left(q_\varphi^\theta(x_t^i, u') - q_\varphi^\theta(x_t^i, u_t^i) \right)_+^2$$

where the maximum of actions is computed over a large number of samples. A yet another strategy looks like

$$\Omega = \frac{1}{nT} \sum_{i,t} \log \int_u \exp\left(q_\varphi^\theta(x_t^i, u) \right) \mathrm{d}u \,.$$

These strategies have been found to be somewhat useful in the sense that they prevent the value function from taking large values. But it is also clear that the solution of the problems is not the optimal value function.

ℹ Practically speaking, it is reasonable to expect that even if we do not find the optimal value function, if we can obtain a better policy than the existing policy that is running on the system, then offline reinforcement learning is a viable approach.

11.2.2 Estimating the uncertainty of the value function

It is reasonable to ask the question whether initializing the value function

Chapter 12

Meta-Learning

Reading

1. Learning To Learn: Introduction (1996), http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.21.3140
2. Prototypical Networks for Few-shot Learning http://papers.nips.cc/paper/6996-prototypical-networks-for-few-shot-learning
3. Model-Agnostic Meta-Learning for Fast Adaptation of Deep Networks https://arxiv.org/abs/1703.03400
4. A Baseline for Few-Shot Image Classification https://arxiv.org/abs/1909.02729
5. Meta-Q-Learning https://arxiv.org/abs/1910.00125

The human visual system is proof that we do not need lots of images to learn to identify objects or lots of experiences to learn about concepts. Consider the mushrooms shown in the image below.

The one on the left is called *Muscaria* and you'd be able to identify the bright spots on this mushroom very easily after looking at one image. The differences between an edible one in the center (*Russala*) and the one on the right (*Phalloides*) may sometimes be subtle but a few samples from each of them are enough for humans to learn this important distinction.

There are also more everyday examples of this phenomenon. You touched a hot stove once as a child and have forever learnt not to do it. You learnt to ride a bike as a child and only need a few minutes on a completely new bike to be able to ride it these days. At the same time, you could start learning to juggle today and will be able to juggle 3 objects with a couple of days of practice.

The hallmark of human perception and control is the ability to generalize. This generalization comes in two forms. The first is the ability to do a task better if you see more samples from the same task; this is what machine learning calls generalization. The second is the ability to mix-and-match concepts from previously seen tasks to do well on new tasks; doing well means obtaining a lower error/higher reward as well as learning the new task quickly with few days. This second kind is the subject of what is called "learning to learn" or meta-learning.

$$\text{Standard machine learning} \Rightarrow \text{generalization across samples}$$

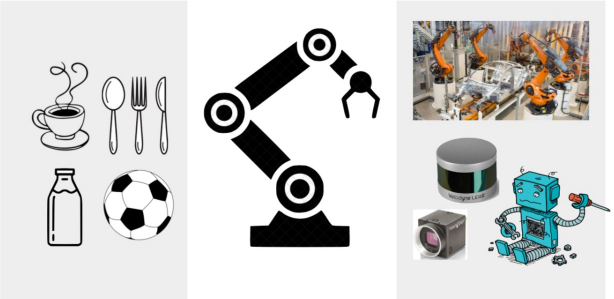
$$\text{Meta-Learning} \Rightarrow \text{generalization across tasks}$$

> What is a task? If we are going to formalize meta-learning, we better define what a task is. This is harder than it sounds. Say we are doing image classification, classifying cats vs. dogs could be considered Task 1; Task 2 could be classifying apples vs. oranges. It is reasonable to expect that learning low-level features such as texture, colors, shapes etc. while learning Task 1 could help us to do well on Task 2.

This is not always the case, two tasks can also fight each other. Say, you design a system to classify ethnicities of people using two kinds of features. Task 1 uses the shape of the nose to classify Caucasians (long nose) vs. Africans (wide nose). Task 2 uses the kind of hair to classify Caucasians (straight hair) vs. Africans (curly hair). An image of a Caucasian person with curly hair clearly results in two tasks fighting each other.

The difficulty in meta-learning begins with defining what a task is.

While understanding what a task is may seem cumbersome but doable for image classification, it is even harder for robotics systems.



We can think of two different kinds of tasks.

1. The first, on the left, is picking up different objects like a soccer ball, a cup of coffee, a bottle of water etc. using a robot manipulator. We may wish to learn how to pick up a soccer ball quickly given data about how to pick up the bottle.

2. The second kind of tasks is shown on the right. You can imagine that after building/training a model for the robot in your lab you want to install it in a factory. The factory robot might have 6 degrees-of-freedom whereas the one in your lab had only 5; your policy better adapt to this slightly different state-space. An autonomous car typically has about 10 cameras and 5 LIDARs, any learning system on the car better adapt itself to handle the situation when one of these sensors breaks down. The gears on a robot manipulator will degrade over the course of its life, our policies should adapt to this degrading robot.

Almost all the current meta-learning/meta-reinforcement learning literature focuses on developing methods to do the first set of tasks. The second suite of tasks are however more important in practice. In the remainder of this chapter, we will discuss two canonical algorithms to tackle these two kinds of tasks.

Meta-learning vs. multi-task learning
We have talked about adaptation as the way to handle new tasks in the previous remark. Consider the following situation: in standard machine learning, we know that larger the size of the training data we collect better the performance on the test data; a large number of images help capture lots of variability in the data, e.g., dogs of different shapes, sizes and colors. You can imagine then that in order to do well on lots of different tasks, i.e., meta-learn, we should simply collect data from lots of different tasks. Can you think as to why mere multi-task learning may not work well for meta-learning?

12.1 Problem formulation for image classification

The image classification formulation thinks of each class/category as a "task".

Consider a supervised learning problem with a dataset $D = \left\{(x^i, y^i)\right\}_{i=1,\ldots,N}$. The labels $y^i \in \{1, \ldots, C\}$ for some large K and there are $\frac{N}{C}$ samples in the dataset for each class. Think of this as a large dataset of cars, cats, dogs, airplanes etc. all objects that are very frequent in nature and for which we can get lots of images. This training set is called the meta-training set with C "base tasks".

Say, we were simply interested in obtaining a machine learning model to classify this data. Let us denote the parameters of this model by w. If this model predicts the probability of the input x belonging to each of these known classes we can think of maximizing the log-likelihood of the data under the model (or minimizing the cross-entropy loss)

$$\hat{w} = \underset{w}{\text{argmax}} \ \frac{1}{N} \sum_{i=1}^{N} \log p_w(y^i \mid x^i). \tag{12.1}$$

This is the standard multi-class image classification setup. Since we like to think of one task as one category, this is also the multi-task learning setup. The model

$$p_{\hat{w}}(\cdot \mid x)$$

after fitting on the training data will be good at classifying some new input image x as to whether it belongs to one of the C training classes. Note that we have written the model as providing the probability distribution $p_{\hat{w}}(\cdot \mid x)$ as the output, one real-valued scalar per candidate class $\{1, \ldots, C\}$.

Say, we are interested in classifying images from classes that are different than those in the training set. The model has only C outputs, effectively the universe is partitioned into C categories as far as the model is concerned and it does not know about any other classes. How should one formalize the problem of meta-learning then?

12.1.1 Fine-tuning

Let us now consider the following setup. In addition to our original dataset of the base tasks, we are given a "few-shot dataset" that has c *new* classes and s labeled samples per class, a total of $n = cs$ new samples

$$D' = \left\{(x^i, y^i)\right\}_{i=N+1,\ldots,N+n}; \quad y^i \in \{C+1, \ldots, C+c\}.$$

The words "few-shot" simply mean that s is small, in particular we are given much fewer images per class than the meta-training dataset,

$$\frac{n}{c} = s \ll \frac{N}{C}.$$

This models the situation where the model is forced to classify images from rare classes, e.g., the three kinds of strawberries grown on a farm in California after being trained on data of cars/cats/dogs/planes etc.

We would like to adapt the parameters $\hat{w}$ using this labeled few-shot data. Here is one solution, we simply train the model again on the new data. This looks like solving another optimization problem of the form

$$w^* = \operatorname*{argmin}_w -\frac{1}{n}\sum_{i=N+1}^{N+n} \log p_w(y^i \mid x^i) + \frac{\lambda}{2}\|w - \hat{w}\|_2^2. \tag{12.2}$$

The new parameters w can potentially do well on the new classes even if the shot s is small because training is initialized using the parameters $\hat{w}$. We write down this initialization using the second term

$$\frac{\lambda}{2}\|w - \hat{w}\|_2^2$$

which keeps the parameters being optimized w closed to their initialization using a quadratic spring-force controlled by the parameter λ. We can expect the new model p_{w^*} to perform well on the new classes if the initialization $\hat{w}$ was good, i.e., if the new tasks and the base tasks were close to each other. This method is called fine-tuning, it is the easiest trick to implement to handle new classes.

Think of a multi-layer neural network $\hat{w}$ that has K outputs. The new network should now produce m outputs, how should we modify this network?

12.1.2 Prototypical networks

The cross-entropy objective used in (12.1) to train the model $p_{\hat{w}}$ simply maximizes the log-likelihood of the labels given the data. It is reasonable to think that since the base classes are not going to show up as the few-shot classes, we should not be fitting to this objective.

The idea behind a prototypical loss is to train the model to be a good discriminator among different classes.

Let us imagine the features of the model, e.g., the activations of the last layer in the neural network,

$$z = \varphi_w(x)$$

for a particular image x. Note that the features z depend on the parameters w. During standard cross-entropy training, there is a linear layer on top of these features and our prediction probability for class y is

$$p_w(y \mid x) = \frac{e^{w_y^\top z}}{\sum_{y'} e^{w_{y'}^\top z}}$$

where $w_y \in \mathbb{R}^{\dim(z)}$. This is the softmax operation and the vectors w are the weights of the last layer of the network; when we wrote (12.1) we implicitly folded those parameters into the notation w.

Prototypical networks train the model to be a discriminator as follows.

1. Each mini-batch during training time consists of a few-shot task created out of the meta-training set by sub-sampling.

$$\begin{aligned} D^{\text{episode}} &= D^{\text{support}} \cup D^{\text{query}} \\ &= \left\{(x^i, y^i);\ y^i \in \{1, \ldots, C\}\right\}_{i \in \{1,\ldots,Cs\}}; \\ &\quad \cup \left\{(x^i, y^i);\ y^i \in \{1, \ldots, C\}\right\}_{i \in \{1,\ldots,Cq\}}. \end{aligned}$$

   with $|D^{\text{support}}| = Cs$ and $|D^{\text{query}}| = Cq$. This is called an "episode" by researchers in this literature. Each episode comes with some more data from the same classes called the "query-shot" in this literature. The query-shot is akin to the data from the new classes that the model is forced to predict during adaptation time. Let us have q query-shot per class in each episode.

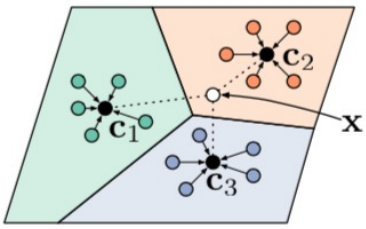
2. We know the labels of the $N = Cs$ labeled data and can compute the prototypes, which are simply centroids of the features,

$$\mu_y = \frac{1}{s} \sum_{(x^i, y^i) \in D^{\text{episode}}} \mathbf{1}_{\{y^i = y\}}\ \varphi_w(x^i).$$

3. You can now impose a clustering loss to force the query samples to be labeled correctly, i.e., maximize

$$p_{w,\mu_y}(y \mid x) = \frac{e^{-\|\varphi_w(x) - \mu_y\|_2}}{\sum_{y'}\ e^{-\left\|\varphi_w(x) - \mu_{y'}\right\|_2}}$$

   where $y = y^i$ and $x = x^i$ for each of the samples (x^i, y^i) in the query-set of the episode.



(a) Few-shot

4. The objective maximized at each mini-batch is

$$\frac{1}{Cq} \sum_{(x,y) \in D^{\text{query}})} \log p_{w,\mu_{y^i}}(y^i \mid x^i).$$

   Note that the gradient of the above expression is both on the weights w of the top layer and the weights w of the lower layers.

5. We can now use the trained model for classifying new classes by simply feeding the new images through the network, computing the prototypes using the few labeled data and computing the clustering loss on the unlabeled query data at test time to see which prototype the feature of a particular query datum is closest to.

Discussion Prototypical loss falls into the general category of metric-based approaches to few-shot learning. We make a few remarks next.

1. It is a very natural setting for learning representations of the data for classification that can be transferred easily. If the model is going to be used for new classes, it seems reasonable that the prototypes of the new classes should be far away from each other and the zs of the query samples should be clustered around their correct prototypes.
2. Prototypical networks perform well if you can estimate the prototypes accurately. In practice, this requires that you have about 10 labeled data per new class.
3. We used the ℓ_2 metric $\|\cdot\|_2$ in the z-space to compute the affinities of the query samples. This may not be a reasonable metric to use for some problems, so a large number of approaches try to devise/learn new metrics.
4. A key point of prototypical networks is that there is no gradient-based learning going on upon the new categories; we simply compute the prototypes and the affinities and use those to classify the new samples.

12.1.3 Model-agnostic meta-learning (MAML)

We will next look at a simple algorithm for gradient-based adaptation of the model on the new categories. The key idea is to update the model using the same objective in (12.1) but avoid overfitting the model on the meta-training data so that the model can be quickly adapted using the few-shot data via gradient-updates.

Here we consider an episode $D^{\text{episode}} = D^{\text{support}}$ and $D^{\text{query}} = \emptyset$, i.e., there are no query shots. Let us define

$$\ell(w; D^{\text{support}}) = \frac{1}{Cs} \sum_{(x^i, y^i) \in D^{\text{support}}} \log p_w(y^i \mid x^i);$$

this is the same objective as that in (12.1) so if we maximized the objective

$$\ell(w;\ D^{\text{support}})$$

we will perform standard cross-entropy training. At each mini-batch/episode, the MAML algorithm instead maximizes the objective

$$\ell_{\text{maml}}(w;\ D^{\text{support}}) = \ell\left(w + \alpha \nabla \ell(w; D^{\text{support}});\ D^{\text{support}}\right). \tag{12.3}$$

In other words, MAML uses a "look ahead" gradient: the gradient of $\ell(w;\ D^{\text{support}})$ is not in the steepest ascent direction of $\ell(w; D^{\text{support}})$ but in the steepest ascent direction after one update of the parameters $w + \alpha \nabla \ell(w;\ D^{\text{support}})$.

Adaptation on the few-shot data One we have a model trained using MAML

$$\hat{w} = \operatorname*{argmax}_w \ell_{\text{maml}}(w;\ D)$$

we can update it on new data simply by maximizing the standard cross-entropy objective again, i.e.,

$$w^* = \operatorname*{argmax}_w \frac{1}{n} \sum_{i=N+1}^{N+n} \log p_w(y^i \mid x^i) - \frac{1}{2\lambda} \|w - \hat{w}\|_2^2. \tag{12.4}$$

How does look-ahead in MAML help?

The adaptation phase is exactly the same as standard cross-entropy training.

MAML as an approximation of a second order optimization method MAML is not specific to few-shot learning. We can use the MAML objective for any other standard supervised learning problem, is this going to help? Indeed it will, gradient descent/stochastic gradient descent are myopic algorithms because they update parameters only in the direction of the steepest gradient, you can potentially do better by computing the lookahead gradient. The caveat is that is computationally difficult to compute the lookahead gradient. Observe that

$$\begin{aligned} \ell_{\text{maml}}(w) &= \ell(w + \alpha \nabla \ell(w)) \\ &\approx \ell(w) + \alpha \left(\nabla \ell(w)\right)^\top \nabla \ell(w) \\ \Rightarrow \nabla \ell_{\text{maml}}(w) &= \nabla \ell(w) + 2\alpha \nabla^2 \ell(w)\ \nabla \ell(w). \end{aligned}$$

So MAML is secretly a second-order optimization method, computing the gradient of the MAML objective requires having access to the Hessian of the objective $\nabla^2 \ell(w)$. For large models such as neural networks this is

very expensive to compute.

Remark 12.1. Let us consider a meta-training set with two mini-batches/episodes/tasks, $D = D^1 \cup D^2$. The MAML algorithm uses the gradient

$$\begin{aligned}\nabla \ell_{\text{maml}}(w; D) &= \nabla \frac{1}{2} \sum_{i=1}^{2} \ell_{\text{maml}}(w; D^i) \\ &= \frac{1}{2} \sum_{i=1}^{2} \nabla \ell(w; D^i) + 2\alpha \nabla^2 \ell(w; D^i) \, \nabla \ell(w; D^i)\end{aligned}$$

Observe now that if there exist parameters w that have $\nabla \ell(w; D^i)$ for all the episodes D^i then the MAML gradient is also zero. In other words, if there exist parameters w that work well for all tasks then MAML may find such parameters. However, in this case, the simple objective

$$\ell_{\text{multi-task}}(w; D) = \frac{1}{2} \sum_{i=1}^{2} \ell(w; D^i) \tag{12.5}$$

that sums up the losses of all the mini-batches/episodes/tasks will also find these parameters. This objective known as the multi-task learning objective is much simpler than MAML's because it requires only the first-order gradient.

❓ What happens if the two tasks are different (as is likely to be the case) in which case there don't exist parameters that work well for all the tasks?

12.2 Problem formulation for meta-RL

One mathematical formulation of meta-RL is as follows. Let k denote a task and there is an underlying (unknown) dynamics for this task given by

$$x_{t+1}^k = f^k(x_t^k, u_t^k, \xi_t)$$

We will assume that all the tasks have a shared state-space $x_t^k \in X$ and a shared control-space $u_t^k \in U$. The reward function of each task is different $r^k(x, u)$ but we are maximizing the same infinite-horizon discounted objective for each task. The q-function is then defined to be

$$q^{k,\theta^k}(x, u) = \mathop{\mathrm{E}}_{\xi(\cdot)} \left[\sum_{t=0}^{\infty} \gamma^t r^k(x_t, u_t) \;\middle|\; x_0 = x, u_0 = u, u_t = u_{\theta^k}(x_t) \right].$$

where $u_{\theta^k}(x_t)$ is a deterministic controller for task k. Given all these meta-training tasks, our objective is to learn a controller that can solve a new task $k \notin \{1, \ldots, K\}$ upon being presented a few trajectories from the new task. Think of you learning to pick up different objects during training time and then adapting to picking up a new object not in the training set.

Let us consider the off-policy Q-learning setting and learn separate controllers for all the tasks for now. As usual, we want the q-function

to satisfy the Bellman equation, i.e., if we are using parameters φ^k to approximate the q-function, we want to find parameters φ^k such that

$$\underset{\varphi^k}{\text{argmin}} \underset{(x,u,x')\in D^k}{\text{E}} \left[\left(q^{k,\theta^k}_{\varphi^k}(x,u) - r^k(x,u) - \gamma q^{k,\theta^k}_{\varphi^k}(x', u_{\theta^k}(x')) \right)^2 \right] \tag{12.6}$$

where the dataset D^k is created using some exploratory policy for the task k. The controllers u_{θ^k} are trained to behave like the greedy policy for the particular q-function $q^{k,\theta^k}_{\varphi^k}$

$$\underset{\theta^k}{\text{argmax}} \underset{(x,u)\in D^k}{\text{E}} \left[q^{k,\theta^k}_{\varphi^k}(x, u_{\theta^k}(x)) \right]. \tag{12.7}$$

The above development is standard off-policy Q-learning and we have seen it in earlier lectures. The different controllers u_{θ^k} do not learn anything from each other, they are trained independently on their own datasets. We can now construct a multi-task learning objective for meta-RL, in this we will learn a single q-function and a single controller for all tasks. We modify (12.6) and (12.7) to simply work on all the datasets together

$$\begin{aligned} &\underset{\varphi}{\text{argmin}} \underset{(x,u,x')\in D^1\cup D^2 \ldots}{\text{E}} \left[\left(q^{\theta}_{\varphi}(x,u) - r^k(x,u) - \gamma\, q^{w}_{\varphi}(x', u_w(x')) \right)^2 \right] \\ &\underset{w}{\text{argmax}} \underset{(x,u)\in D^1\cup D^2 \ldots}{\text{E}} \left[q^{w}_{\varphi}(x, u_w(x)) \right] \end{aligned} \tag{12.8}$$

This is the multi-task learning objective for RL. This is unlikely to work well because depending upon the task, the controllers for the different tasks will conflict with each other, it is unlikely that there is a single set of parameters for the controller and the q-function that works well for all tasks.

Imagine a planning task with multiple goals. The optimal trajectory that goes to one goal location will bifurcate from the optimal trajectory that goes to some other goal. We will never be able to learn a controller that goes to both goals using one neural network. How to fix this? You can use MAML certainly to under-fit the controller and the q-function to all the tasks and then adapt them using some data from the new task using gradient updates.

12.2.1 A context variable

Reinforcement Learning offers a very interesting way to solve the few-shot/meta-learning problem. We can append the state-space to include a context variable that is a representation of the particular task. Let us construct features for a trajectory using a set of basis functions

$$\{\phi_1(x,u,r), \phi_2(x,u,r), \ldots, \phi_m(x,u,r)\}$$

We will now construct a variable $\mu^k(\tau)$ for a trajectory $\tau_{0:t} = (x_0, u_0, \ldots, x_t, u_t, \ldots)$ from task k as

$$\mu(\tau_{0:t}) = \sum_{s=0}^{t} \sum_{i=1}^{m} \gamma^t\, \alpha_i\, \phi_i(x^k_s, u^k_s, r^k(x^k_s, u^k_s)).$$

We will call this variable a "context" because we can use it to guess which task a particular trajectory is coming from. **It is important to note that the mixing coefficients α_i are shared across all the tasks**. We would

like to think of this feature vector $\mu(\tau)$ as a kind of indicator of whether a trajectory τ belongs to the task k or not. We now learn a q-function and controller that also depend on $\mu(\tau)$

$$\begin{aligned} &q_\varphi^\theta(x_t, u_t, \mu(\tau_{0:t})) \\ &u_\theta(x_t, \mu(\tau_{0:t})). \end{aligned} \tag{12.9}$$

Including a context variable like $\mu(\tau)$ allows the q-function to detect the particular task that it is being executed for using the past t time-steps of the trajectory $\tau_{0:t}$. This is similar to learning independent q-functions $q_{\varphi_k}^{k,\theta^k}$ and controllers u_{θ^k} but there is parameter sharing going on in (12.9). We will still solve the multi-task learning problem like (12.8) but also optimize the parameters α_is that combine the basis functions.

$$\begin{aligned} &\underset{\varphi,\alpha_i}{\operatorname{argmin}} \sum_{i=1}^{K} \underset{(x,u,x')\in D^k}{\mathrm{E}} \left[\left(q_\varphi^\theta(x,u,\mu(\tau)) - r^k(x,u) - \gamma\, q_\varphi^\theta(x', u_\theta(x', \mu(\tau))) \right)^2 \right] \\ &\underset{\theta,\alpha_i}{\operatorname{argmax}} \sum_{i=1}^{K} \underset{(x,u)\in D^k}{\mathrm{E}} \left[q_\varphi^\theta(x, u_\theta(x, \mu(\tau)), \mu(\tau)) \right]. \end{aligned} \tag{12.10}$$

The parameters α_i of the context join the q-functions and the controllers of the different tasks together but also allow the controller the freedom to take different controls depending on which task it is being trained for.

Adapting the meta-learnt controller to a new task Suppose we trained on K tasks using the above setup and have the parameters $\hat{\theta}, \widehat{\varphi}, \{\widehat{\alpha_i}\}_{i=1,\ldots,m}$ in our hands. How should we adapt to a new task? This is easy, we can run an exploration policy on the new task (the current policy u_θ will work just fine if the control space U is the same) to collect some data and update our off-policy Q-learning parameters $\hat{\theta}, \widehat{\varphi}$ on this data using (12.6) and (12.7) while keeping the results close to our meta-trained parameters using penalties like

$$\frac{1}{2\lambda}\|w - \hat{w}\|_2^2 \quad \text{and} \quad \frac{1}{2\lambda}\|\varphi - \widehat{\varphi}\|_2^2.$$

❓ Does adaptation always improve performance on the new task?

We don't update the context parameters α_is during such adaptation.

12.2.2 Discussion

This brings an end to the chapter on meta-learning and Module 4. We focused on adapting learning-based models for robotics to new tasks. This adaptation can take the form of learning a reward (inverse RL), learning the dynamics (model-based RL) or learning to adapt (meta-learning). Adaptation to new data/tasks with few samples is a very pertinent problem because we want learning-based methods to generalize a variety of different tasks than the ones they have been trained for. Such adaptation also comes with certain caveats, adaptation may not always improve the performance

on new tasks; understanding when one can/cannot adapt forms the bulk of the research on meta-learning.

Bibliography

Censi, A. (2016). A class of co-design problems with cyclic constraints and their solution. *IEEE Robotics and Automation Letters*, 2(1):96–103.

Fakoor, R., Mueller, J. W., Asadi, K., Chaudhari, P., and Smola, A. J. (2021). Continuous doubly constrained batch reinforcement learning. *Advances in Neural Information Processing Systems*, 34:11260–11273.

Levine, S., Kumar, A., Tucker, G., and Fu, J. (2020). Offline reinforcement learning: Tutorial, review, and perspectives on open problems. *arXiv preprint arXiv:2005.01643*.

Turing, A. M. (2009). Computing machinery and intelligence. In *Parsing the Turing Test*, pages 23–65. Springer.