DRAFT: Do Not Circulate

Artificial Intelligence

CS 343H – Fall 2026

Volkan Isler

Department of Computer Science
The University of Texas at Austin

Notes and Manuscript Draft

Released: August 23, 2026

Last modified: August 23, 2026

Preface

I wrote these notes in the summer of 2026 for CS 343H Artificial Intelligence (HON) at UT Austin. Over the last decade, the "AI class" as part of the CS curriculum went through some major changes. It used to be a bit of an eclectic course focused on programming, game-playing and logic-based planning[1]. Artificial Intelligence (AI) is now a mainstream technology and one of the main drivers of the economy. While logic-based methods retain their importance (and gain even more as part of increasing focus on verification), the techniques used in current AI systems draw heavily from probability theory, linear algebra, optimization and machine learning. Perhaps most importantly, hand-designed state representations have mostly been replaced with learned (often neural network based) representations whose outputs are interpreted as probabilities.

AI classes and textbooks address this shift by appending the new material after the traditional material. This approach leads to bloated textbooks and creates a gap in student expectations, preparation and learning in classes: A typical AI course starts out with basic search techniques that the students are already familiar and comfortable with from their programming classes. Then it goes through a, perhaps sudden, shift as the material on Monte Carlo Search, Markov Decision Processes is introduced. Later on, a similar shift happens when neural networks, in particular, modern architectures like convolutional neural networks and transformers are introduced. Overall, the students finish the semester with a fragmented view of the field.

I wrote these notes in the hope that a geometry-focused introduction to some of the main concepts such as dynamic programming, gradients, neural nets as function representations and regression as a method to learn representation parameters will help smooth these sudden shifts and help prepare for the upcoming material. In my experience, the *"tell them what you'll tell them, tell them, tell them what you told them"*approach to good technical writing is effective for (human) learning in general. This is the "tell them what you'll tell" them part. I tried to keep the main text as accessible as I can, and put some of the details in appendices for students who would like to learn more about the formalizations and fancy terminology.

To my students: I hope you will find this approach helpful and enjoyable, and that it

[1] Mine was entirely in Prolog - a logic programming language!

gives you guidance for your learning both in this class, and in your AI journey beyond. As you will see, it is work-in-progress. Please let me know if you see any errors or confusing parts.

— Volkan Isler, August 2026, Austin, TX

A note on my workflow

I am not a heavy user of generative AI tools. I wanted to leverage this project (of writing these notes) to gain more experience and to better understand the state of the art in GenAI. During this project, I primarily used CoPilot and occasionally Claude available through UT's IT infrastructure. I tried a few variations but my final workflow ended up as follows: First, I setup a latex project. Next, I wrote almost all of the chapters by hand (yes, with a pencil on a notebook!) and drew sketches of the figures. Then I scanned my notes, and asked the AI assistant to translate into latex and generate the figures without changing the text. Once I have the latex file ready based on what I originally wrote, GenAI tools were extremely helpful for tasks like making notation consistent and adding missing steps in derivations.

In some ways, this process is similar to how we will optimize a path from point a to point b as shown in Figure 6.3 using gradient descent in the path space. In this approach, the handwritten draft is the initial path. GenAI is used for gradual improvements over the whole structure. This approach allowed me to focus on the content I want to convey without adding additional complexity or "fluff", and to retain my voice as a teacher and writer.

Contents

Part I

Charting AI from a Geometric Vantage Point

Chapter 1

Introduction

Many problems in Artificial Intelligence (AI) can be formulated as variations of the following planning problem:

Compute a sequence of actions to take an agent from an initial state to a goal state.

Sometimes, simply reaching the goal (such as winning a board-game) is enough. Other times, we also care about how we reach the goal. Often, there is a running cost, and the objective is, for example, to minimize the total number of actions taken, or their total cost, until the goal is reached.

A key player in the formulation above is the state representation. Informally, the **state** is a description of all relevant parameters of a task. Ideally the state description should include all relevant parameters and only relevant parameters (i.e. nothing more).

In this part of the course, we will build the mathematical language for representing states as vectors. Next, we will turn our attention to cost functions and look at methods for solving the planning problem above. The goal is not a comprehensive treatment. It is to gain the geometric vantage from which the main territories of AI come into view, and to establish a way of seeing that will carry us through the rest of the course. Later parts of the course, we will go beyond geometry and look at symbolically defined state spaces, and stochastic cost functions. These topics, in their classical form, may seem like they do not rely on geometry at all. What is striking is that their modern, high-performing versions increasingly do: rather than being hand-designed, they learn high-dimensional representations that capture the underlying structure, or manifold, of the space they operate in. Generative models are a clear case as they learn the manifold on which realistic data lives, and then generate new samples from it. From this point-of-view, the geometric view is not just a convenient entry point but a solid foundation for you to build on in your AI journey.

1.1 Running Example: Planar Navigation

Consider an agent in a two-dimensional world. The task is navigation. For this problem, a natural representation for the state of the agent is the agent's position. We will refer to this example throughout the chapter. Let's take a closer look at the position vector.

To represent position vectors, we associate them with coordinates $[a, b]^T$ where a and b are real numbers in $\mathbb{R}$. We will adopt the convention of representing positions as column vectors. This is why you see the transpose ^{T}. We will use this informal, coordinate-level view throughout the chapter and defer the formal definition of a vector space to Appendix B. There, we make precise the sense in which these position vectors are elements of the vector space $\mathbb{R}^2$ over the field $\mathbb{R}$.

Coordinates allow us to reason about distances and geometry in an abstract way. But if someone gives us the vector $[a, b]^T$, how do we know which position (point) on the plane it corresponds to?

If our 2D world is this page, where is the point $[-2, 5.1]^T$?

Chapter 2

Coordinate Frames and Vectors

To figure out the specific point $\vec{P} = [a, b]^T$, we need to know which coordinate frame it is expressed in[1].

> Recall that for two vectors $\vec{x}_1 = [a_1, b_1]^T$ and $\vec{x}_2 = [a_2, b_2]^T$, the Euclidean norm is $\|\vec{x}_1\| = \sqrt{a_1^2 + b_1^2}$, and the dot product is $\vec{x}_1 \cdot \vec{x}_2 = a_1 a_2 + b_1 b_2$.

A coordinate frame (in 2D) consists of:

- An origin $\vec{O}$
- Two axes $\vec{x}$ and $\vec{y}$

The axis vectors have some special properties:

- Unity: $\|\vec{x}\| = \|\vec{y}\| = 1$
- Orthogonality: $\vec{x} \cdot \vec{y} = 0$

In fancy words, they are orthonormal. The important point is that knowing the coordinate frame lets us agree on the origin and the axes, which in turn anchors the representation to a specific geometric space. We use the notation ${}^A\vec{P} = [a, b]^T$ to express the specific coordinate frame P is expressed in. Now that we have a specific coordinate frame $\{A\}$, the point ${}^A P = [a, b]^T$ can be located from its coordinates by starting at the origin of A and moving:

- a units along the x-axis
- b units along the y-axis

[1] I use the arrow $\vec{P}$ to emphasize that P is a vector. If this is clear from the context, the arrow is omitted to keep the notation simple.

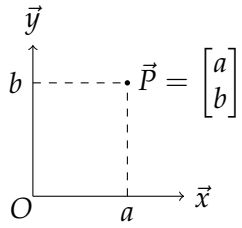


Figure 2.1: A coordinate frame: given by its origin O and two axes $\vec{x}$ and $\vec{y}$. Once we know the coordinate frame and the coordinate values, we can exactly pinpoint $P = [a, b]^T$.

Equivalently,

$$\vec{P} = \vec{O} + a\vec{x} + b\vec{y}$$

In Figure 2.1, given $\vec{O}$, $\vec{x}$ and $\vec{y}$ as shown, we get to the specific point P (on the page or screen!) shown in the figure.

2.1 Examples of Vector Representations

Our 2D example provides us a useful starting point. Vector representations in high dimensions are essential for representing more complex states. Let's look at some examples.

From Coordinates to Paths For our navigation problem, we can represent a path as a sequence of n waypoints where each waypoint is a position vector. Therefore, a path can be represented as a $2n$ dimensional vector. We will study this example further in Section 6.

Audio signals Standard audio sampling at 44.1 kHz results in a 44,100-dimensional vector representing a recording of duration one second.

Images A 10-megapixel image corresponds to a 3×10^7-dimensional vector (three values per pixel: R, G, B).

Functions The signals above can also be represented as functions:

- $u(t)$ for an audio signal
- $f(u, v)$ for image intensity

In practice, these functions are measured at discrete points. For audio, sampling at 44.1 kHz means recording the value of $u(t)$ at 44,100 equally spaced time points per second —

exactly the 44,100-dimensional vector we saw above. For an image, recording $f(u, v)$ at each of the 10^7 pixel locations gives the 3×10^7-dimensional vector (three color channels per pixel). In general, a function sampled at n points can be represented as a vector in $\mathbb{R}^n$.

Chessboard Evaluation We can design a function to evaluate a chessboard using features such as:

- Remaining pieces
- Control of important regions (e.g., the center)
- Mobility of pieces
- Pawn structure

The function takes as input a board state representation and returns a real number indicating the strength of the board. It can, for example, output a linear combination of the feature values above. In this case, each feature can be interpreted as a basis vector.

Chessboard State as a Vector The state of the chessboard during a game can be encoded as a vector of dimension $64 = 8 \times 8$, where each axis corresponds to a board square. A natural first attempt is to store a piece code on each square: 0 for empty, 1 for a white pawn, 2 for a white rook, and so on. This records the board faithfully, but it is a poor representation if we want distances to be meaningful. The codes are merely *names* for the pieces, yet writing them as numbers artificially imposes an order and spacing the pieces do not have. With pawn $= 1$ and rook $= 2$, swapping a pawn for a rook changes the vector by 1, while swapping it for a queen (code 5, say) changes it by 4 — as though a queen swap were four times "farther." Renumber the pieces and all these distances change, so the geometry is an accident of labeling and tells us nothing.

A fix is to stop treating piece identity as a number. Instead of one coordinate per square holding a code, we use one coordinate per (square, piece type) pair. There are 12 piece types (6 white, 6 black), so each square gets a block of 12 coordinates: a 1 in the slot for the piece occupying that square, 0 everywhere else, and all zeros for an empty square. The board becomes a vector of dimension $64 \times 12 = 768$. This is a *one-hot* encoding, and it is essentially how chess-playing neural networks represent the board. Now no piece is numerically larger than another, and the squared Euclidean distance between two boards simply counts how many squares hold different contents: positions differing on one square are close, those differing on many are far. In this case, there are other metrics such as Hamming distance that can be used. See Appendix B for a more thorough discussion.

One-hot representation removes the false geometry, but it is not a rich one: the distance is just a count of differing squares, so every legal move looks about equally "small,"

whether brilliant or a blunder. This is the limitation of *hand-designed* encodings in general. We decide in advance what each coordinate means, and the resulting geometry only reflects the bookkeeping we imposed, not the structure of the problem. What we would really like is for nearby vectors to correspond to similar *situations*: for example, boards with similar winning chances should be within the vicinity of each other. This is easier said than done because changing the location of a single piece can alter the game drastically. This is the motivation for *learned* representations, where the encoding itself is discovered from data (from game logs or self-play) rather than fixed up front, so that distance in the vector space captures something we actually care about. Neural networks are the dominant tool for learning such representations.

Word Embeddings Learned word embeddings (vectors) are widely used in modern language processing systems. Similar to the chess-board representation discussion, rather than fixing the coordinates by hand, models such as Word2Vec *learn* a 200–300 dimensional vector for each word from large amounts of text. Because the vectors are shaped by how words are actually used, distance in the embedding space reflects meaning: words that appear in similar contexts end up close together[2].

In short, it's fair to say that vector representations are a cornerstone of modern AI systems.

Explore Further

Use your GenAI software to learn more about the state representations mentioned above or for your favorite AI problem. For example, ask about the full list of features used for chess games, or look up Word2Vec.

[2]This example comes with apologies to UTCS Professor Ray Mooney who famously said this: `https://www.cs.utexas.edu/~mooney/cramming.html`

Chapter 3

Norms, Neighborhoods and Shortest Paths

Let us return to our planar navigation problem. Suppose we have agreed on a coordinate frame. The agent is currently at state (point) x_0 and wants to reach point x_f.

At first glance, the answer to the question *How should the agent move?* seems "straightforward" (ha ha). However, it is not yet well defined. There are two issues we must clarify:

1. What actions are available to the agent?
2. What makes one solution better than another?

Both questions rely fundamentally on the notion of *distance* in the underlying state space.

3.1 Actions and Local Motion

To make progress, we first need to define how the agent can move. In general, actions could be very broad (for example, "teleport directly to x_f"). If we had such a function to reach any point, the problem would be trivial. On the other hand, if the actions are completely arbitrary, the problem could easily become intractable. In most realistic settings, actions are *local*: the agent can only make small moves from its current position. For now, let's focus on geometrically defined neighborhoods; we will see more general action sets later in the course.

In our planar setting, there are two ways to make this precise. We can assume that:

- the agent has bounded speed, or
- it moves in steps and at each step, it can move to any point within *unit distance* of its current location.

Next, we need a precise way to measure distance.

3.2 Vector Norms

We have already seen an example in the previous section: The L_2 norm for measuring the length of vectors in the 2D case. The notion of "unit distance" requires a function that measures the size of the displacement vector $\vec{b} - \vec{a}$. This is precisely what a *vector norm* provides.

> **Definition.** A norm $\|\cdot\|$ assigns a non-negative length to each vector. It is zero only for the zero vector, scales as $\|c\,\vec{v}\| = |c|\,\|\vec{v}\|$, and satisfies the *triangle inequality* $\|\vec{u} + \vec{v}\| \leq \|\vec{u}\| + \|\vec{v}\|$.

The triangle inequality is critical for imposing structure on the state space. We will see it in various forms throughout the course.

Using a norm, the feasibility constraint becomes

$$\|\vec{x}_{\text{next}} - \vec{x}_{\text{current}}\| \leq 1,$$

that is, each step moves the agent by at most unit length (or some other threshold). This gives a precise definition of the set of available actions.

Armed with the notion of a norm, we can:

- define what moves are allowed (the neighborhood of a point), and
- define the cost of moving.

Different choices of norm lead to different notions of distance, different feasible actions, and ultimately different solutions.

3.3 The Euclidean Norm and Optimal Paths

In a flat, continuous plane, the most natural[1] notion of distance comes from the *Euclidean* (or L_2) norm. For a vector $\vec{v} = [a, b]^T$,

$$\|\vec{v}\|_2 = \sqrt{a^2 + b^2}.$$

This matches our geometric intuition of distance in open spaces. The distance between two points $\vec{p}_1$ and $\vec{p}_2$ is then $\|\vec{p}_2 - \vec{p}_1\|_2$.

[1] literally, because in Nature, many signals like sound or light we needed to deal with throughout our evolution propagate this way.

Going back to our path planning problem, a natural objective is to reach $\vec{x}_f$ from $\vec{x}_0$ while minimizing the total distance traveled. Under the Euclidean norm, a fundamental result holds: *the shortest path between two points in the plane is a straight line.*

To see why, think of any path from x_0 to x_f as a sequence of waypoints: $\vec{x}_0, \vec{x}_1, \vec{x}_2, \ldots, \vec{x}_k = \vec{x}_f$. Consider the displacement vectors between consecutive waypoints $\vec{e}_i = \vec{x}_{i+1} - \vec{x}_i$. As the waypoints grow denser, the sum of their lengths $\sum_i \|\vec{e}_i\|$ converges to the length of the path.

By the triangle inequality (applied repeatedly), the sum of the lengths is at least the length of the sum:

$$\sum_i \|\vec{e}_i\| \;\geq\; \left\|\sum_i \vec{e}_i\right\| = \|\vec{x}_f - \vec{x}_0\|.$$

The equality on the right holds because we have a telescoping sum. The relationship becomes an equality when every displacement points in the same direction — that is, when the path is the straight segment from $\vec{x}_0$ to $\vec{x}_f$. Any deviation makes at least one displacement point elsewhere and strictly increases the total[2]. So the straight line is the unique shortest path. We will revisit this argument when we talk about gradients in Section 6 (see also Figures 6.2 and 6.3).

Explore Further

To reinforce the concepts above,

(i) Generalize the triangle inequality to more than two vectors: $\sum_i \|\vec{e}_i\| \;\geq\; \left\|\sum_i \vec{e}_i\right\|$

(ii) Verify that $\left\|\sum_i \vec{e}_i\right\| = \|\vec{x}_f - \vec{x}_0\|$.

So far, we deliberately mixed up the notion of norm (the length of a vector) and distance (the length of the shortest path) because these two concepts coincide in the Euclidean space. The vector from $\vec{x}_f - \vec{x}_0$ is also the shortest path.

From the point of neighborhoo definition, in the Euclidean space, the local and global are identical and therefore the *neighborhood structure* (what movements are allowed) and the *objective* (what paths are optimal) coincide: think of a small neighborhood, a ball of radius r, centered around x_i. Pick any point x on the boundary of this ball. The shortest path from x_i to x is a line. We can keep growing the ball by increasing r until it includes x_f, the neighborhoods remain disks and the shortest path structure does not change.

As we will see in the next section, this is not true in general. In more general spaces, the situation is more complex. If you view the sphere as a three dimensional object, the line connecting two points does not stay on the surface of the sphere. So what is the

[2]Since this inequality holds for any finite set of waypoints, and both sides converge as the waypoints grow denser, it holds in the limit as well.

shortest path? Can there be more than one? What do the neighborhoods look like? These questions motivate the next section.

Explore Further

Learn about other norms (e.g., L_1, L_∞). What do the local neighborhoods look like? What kind of actions/costs lead to such neighborhoods?

Chapter 4

Curvature, Non-Flat Spaces, and Geodesics

So far, we have assumed that the agent operates in a flat, Euclidean plane. In such environments, neighborhoods are simple (disc-shaped), and the shortest path between two points is a straight line. However, many real-world environments are not flat. In these settings, both the notion of a neighborhood and the notion of optimal paths become more subtle.

Let us consider two examples.

4.1 Example 1: Navigation on a Terrain

Consider an agent moving on the UT Austin campus. While we often draw maps of the campus on a flat sheet of paper, the actual environment is not flat. There are slopes, hills, stairs, and varying elevations. A terrain, also known as a height-map, which assigns a height value to every point on the plane is a more accurate representation. Figure 5.2 shows a hypothetical terrain.

Even though the agent's position can still be described using coordinates, the geometry of the space is no longer identical to the flat plane. In particular:

- Moving "straight" in the coordinate sense may require going uphill or downhill,
- Distances are affected by elevation changes,
- The effort required to traverse a path depends on the steepness of the terrain along the way.

As a result, the local neighborhood of a point is no longer perfectly captured by a flat disc. Instead, it is distorted by the underlying surface. If we zoom in very closely, the terrain looks approximately flat, but globally it exhibits *curvature*.

4.2 Example 2: Navigation on the Surface of the Earth

A second example arises when we consider navigation on the surface of the Earth. For simplicity, we model the Earth as a sphere.

Suppose an agent wants to travel between two cities. If we draw the Earth on a flat map, it may seem that the shortest path is a straight line on that map. However, airplanes do not follow straight lines on flat maps. Instead, they follow curved routes. This happens because the surface of the Earth is curved and distances must be measured *along the surface*, not through it.

On a sphere, the shortest path between two points lies along a *great circle*. To obtain the great circle, form the plane through your two points and the center of the sphere, and intersect the sphere with this plane. Two points that are not antipodal determine a unique great circle, and they split it into two arcs; the shorter arc is the shortest path (Antipodal points are diametrically opposite from each other. The line connecting them passes through the center. Why did we exclude antipodal points from this statement?)

4.3 Curvature and Neighborhoods

These examples illustrate that the underlying space itself can be *curved*.
In a flat (Euclidean) space:

- Neighborhoods are disc-like,
- Geometry is uniform everywhere,
- Straight lines minimize distance.

In a curved space:

- Neighborhoods are only approximately disc-like at very small scales,
- The geometry can change from point to point,
- The notion of a "straight path" must be redefined.

Explore Further

This deviation from flatness is captured by the concept of *curvature*, which measures how much a space bends compared to the flat plane. For our purposes in this course, an intuitive understanding of curvature suffices. If you want to refresh your calculus, you can review the formal definition for planar curves (remember a circular arc with radius r has curvature $1/r$) and explore further in a differential geometry course to learn about curvature in higher dimensions.

4.4 Geodesics: Generalizing Straight Lines

In curved spaces, we replace the idea of a straight line with the notion of a *geodesic*. A geodesic is a path that *locally* minimizes distance: if you perturb it slightly while keeping its endpoints fixed, you get a path that is at least as long.

You can think of a geodesic as a shortest path but there is a subtlety you should be aware of: Every shortest path is a geodesic, but a geodesic need not be globally shortest. The situation on the sphere is a good example. Two non-antipodal points split their great circle into two arcs. Both arcs are geodesics — each one is locally length-minimizing — yet only the shorter arc is the shortest path between the points. So *geodesic* is the local notion and *shortest path* is the global one. They coincide exactly when a geodesic is the shortest among all geodesics joining its endpoints.

As we have seen, there might be multiple geodesics between two points. Even if we limit our attention to the shortest one, there might be multiple (in fact, infinitely many) shortest paths. What's a good example for the sphere?

The main takeaway message is that geodesics generalize straight lines:

- In Euclidean space, geodesics are straight lines,
- On a sphere, geodesics are arcs of great circles,
- On a terrain, geodesics are paths that locally minimize travel distance (or effort, depending on the model). See Figure 5.4.

When the underlying space is curved, both neighborhoods and optimal paths change. Norms alone are no longer sufficient to describe geometry; we must account for the structure of the space itself. Understanding this transition from flat to curved spaces is essential in many areas of AI and robotics, where agents operate in complex environments that cannot be accurately modeled using simple Euclidean geometry. Two remarks before we move on to computing shortest paths.

Remark 1. Even in more general spaces, it is reasonable to assume that local neighborhoods are flat. You can readily see this for the sphere example. Here is a more subtle example: if you take a picture of a static scene, move the camera a little bit and take another picture, we can still use L_2 to decide whether two images are similar. But as the camera moves farther, L_2 ceases to accurately represent distances in a meaningful way. This suggests that the space of all images is a curved space but can be approximated as locally flat.

Remark 2. So far, we defined neighborhoods using a distance threshold. For agents operating in more general spaces, we will introduce the notion of an *action* to describe neighborhoods. The neighborhood will become set of all states the agent can arrive at by taking a single action.

Explore Further

Learn about curvature, and how geodesics are computed on the sphere. See Section A for a starting point.

Explore Further

The state space can be curved even in the case of an agent operating on the plane! Can you think of any examples? Hint: instead of a point robot, what if your agent was car-like, so that we needed to keep track of both its position and orientation? Then the state would be (x, y, θ). Can you visualize the underlying state space?

Chapter 5

Level Sets and Dynamic Programming

The previous section introduced the notion of a shortest path, but we did not say much about how they are actually computed. In this section, we will develop such a computational method by studying the distance function and the sets it induces. The goal here is to give you a geometric intuition. We will dedicate much of the course to computing shortest paths in more general settings!

Definition 1 (distance function)**.** A distance function, or *metric*, is a function $d(x, y)$ that assigns a real number to each pair of points x and y. Intuitively, $d(x, y)$ measures the length of the shortest path from x to y. A metric must satisfy the following properties:

- $d(x, y) \geq 0$, with $d(x, y) = 0$ if and only if $x = y$;
- $d(x, y) = d(y, x)$ for all x, y (symmetry); and
- $d(x, z) \leq d(x, y) + d(y, z)$ for all x, y, z (the triangle inequality).

These properties can be thought of as generalizations of the properties of vector norms. The first property says distances are non-negative and that the only point at distance 0 from x is x itself. The last property, the **triangle inequality**, says that going from x to z directly can never be longer than taking a detour through some intermediate point y. This simple fact is the key to building shortest paths incrementally, from shorter shortest paths.

Verify these statements

Please take some time to verify the following statements:

- If $f(a, b)$ returns the length of a shortest path from a to b, then f is a metric.
- If $\pi(a, b) = (a = a_0, a_1, \ldots, a_n = b)$ is a shortest path from a to b, then every subpath
$$\pi_{ij} = (a_i, a_{i+1}, \ldots, a_j)$$
is a shortest path from a_i to a_j.

The second statement is especially important. It says that shortest paths have an *optimal substructure*: once a path is optimal, every piece of it is optimal as well. This is the property that *dynamic programming* exploits, since it lets us assemble a shortest path out of shorter shortest-paths. All shortest-path algorithms rely on this property. For unit-cost steps, the wavefront procedure developed below is essentially *breadth-first search*. Once steps can carry different (positive) costs, the same idea generalizes to *Dijkstra's algorithm*[1], also called *uniform-cost search*. We study the general, weighted version later in the course.

Distance from a fixed starting point Fix a starting point x_0 (this is usually the initial state of the agent). Instead of thinking about distances between arbitrary pairs of points, we now focus on the function
$$d(x) : x \mapsto d(x_0, x),$$
which assigns to each point x its distance from the start x_0 taken as a fixed value. A good mental picture for $d()$ is the waves formed when a pebble dropped into water at x_0.

In the flat world, think of all locations the agent can arrive at in i steps: initially at $i = 0$, this is just the point x_0. Then in one step, it can reach a circle of radius one centered at x_0, then a circle of radius two, three, and so on. These are the ripples in a pond mentioned above. They look like concentric circles because the environment is flat. Now we generalize this idea.

The function $d(x_0, x)$ gives us two related but different ways to organize the state space. First, the **level set**
$$L_i = \{x : d(x) = d(x_0, x) = i\}$$
contains all points whose distance from x_0 is *exactly* i.

Second, the **sublevel set**
$$B_i = \{x : d(x) = d(x_0, x) \leq i\}$$
contains all points whose distance from x_0 is *at most* i.

[1] named after former UT Austin Professor Edsger W. Dijkstra.

You can think of the sets B_i as growing balls centered at x_0, and the sets L_i as their boundaries, or *wavefronts* (sometimes also called "shells"). See Figure 5.3. These wavefronts are the sets that we will compute one layer at a time. The two families are related by

$$B_i = \bigcup_{j=0}^{i} L_j, \qquad \text{and} \qquad L_i = B_i \setminus B_{i-1} \quad (i \geq 1).$$

So B_i is cumulative, collecting everything reached up to distance i, while L_i isolates only the new frontier at distance i.

Here is the key point: if you pick any point x on the boundary L_i, there may be many paths from x_0 to x, including multiple optimal ones. But we don't need to track all of them. By optimal substructure, any shortest path to a point *beyond* x passes through x optimally: once we know $d(x_0, x)$, we can extend the path without reconsidering how we arrived at x.

5.1 A discrete-time model

Dealing with a completely general state space is difficult, so we will make two simplifying assumptions.

First, we assume time is discrete. At time t the agent is in state x_t, it takes an action, and at time $t+1$ it moves to a new state x_{t+1}. Thus we can think of motion as proceeding in unit steps (or in the case of multiple players, turns).

> Second, we assume that distances can be computed locally. In many spaces the geometry becomes simple when we zoom in: a small neighborhood of any point looks like a patch of flat Euclidean space, so we can measure short displacements with the ordinary Euclidean norm. Spaces that carry a notion of local distance in this way are called *Riemannian manifolds* (we will just say *manifold*). This local structure is what lets us build up global distances one small step at a time.

Unit-cost steps and wavefronts In the discrete-time setting, let us additionally assume that each step has cost 1. In other words, the objective is to minimize the number of steps or total time. Under this assumption, the distance from x_0 to a state is exactly the minimum number of steps needed to reach it optimally:

$$d(x_0, x) = \text{minimum number of optimal steps from } x_0 \text{ to } x.$$

Because each step has unit cost, the wavefronts acquire a direct algorithmic meaning:

$$L_i = \{x : \text{the shortest path from } x_0 \text{ to } x \text{ uses exactly } i \text{ steps}\},$$

and B_i is the set of states reachable in at most i steps. This is the point at which geometry and dynamic programming meet.

Dynamic programming as wavefront expansion The structure we developed gives a natural algorithm. We begin with

$$L_0 = \{x_0\}.$$

Then, having computed the current frontier L_i, we examine all states that can be reached from points in L_i in one step. Any such state that has not already appeared in $B_i = \bigcup_{j=0}^{i} L_j$ belongs to L_{i+1}.

The computation expands outward from the start in successive wavefronts: $L_0, L_1, L_2, \ldots$ where each wavefront contains the states exactly one unit farther away than the previous one (Figure 5.3). This is the *wavefront expansion*. Recall the example of dropping a pebble into a pond: the ripples move outward from the source, crossing points in order of increasing distance. The algorithm does the same thing in state space.

5.2 Why the triangle inequality matters

Why does this layer-by-layer procedure work?

Suppose a shortest path from x_0 to some point x passes through an intermediate point y. Then the portion of the path from x_0 to y must itself be shortest; otherwise we could replace it with a shorter one and obtain a shorter path to x. This is exactly the optimal substructure property discussed earlier.

Equivalently, if y is the predecessor of x along a shortest path, then

$$d(x_0, x) = d(x_0, y) + d(y, x).$$

Under the unit-cost assumption, each step contributes cost 1, so this becomes

$$d(x_0, x) = d(x_0, y) + 1.$$

Thus, once we know all points in L_i, we can discover candidates for L_{i+1} by taking one more optimal step. This is the essence of dynamic programming: compute a complex quantity by building it from previously computed simpler ones.

5.3 From wavefronts to shortest paths

Once the wavefronts reach x_f, we can compute the shortest path by tracing it backwards. If $x_f \in L_k$, then a shortest path can be reconstructed by moving backward through the shells: find a predecessor $x_{k-1} \in L_{k-1}$ that connects optimally to x_f, then a predecessor $x_{k-2} \in L_{k-2}$ that connects optimally to x_{k-1}, and so on, until we reach x_0.

So the computational strategy has two phases:

1. Compute the wavefronts $L_0, L_1, L_2, \ldots$ outward from the start.
2. Recover *a* shortest path by tracing backward from the target through decreasing distance.

The first phase tells us *how far* every point is from the start. The second phase tells us *how to get there* along a shortest path.

This is already a lot of ideas. Don't worry, we will revisit them many times throughout the course. Next, a few remarks with an eye toward upcoming topics.

Remark 1. There is another useful consequence of organizing the state space by distance. Each point belongs to exactly one wavefront L_i: the one corresponding to its shortest-path distance from x_0. It cannot belong to two different wavefronts, because a point cannot be at two different shortest distances from the start. This is why the wavefront algorithm does not need to revisit a point once it has been added to some L_i: at that moment we have already found its shortest possible distance, and any later path to it is at least as long. *We will revisit these ideas when we discuss tree-search versus graph-search.*

Remark 2. We can also run the same algorithm in reverse. Instead of starting from x_0 and expanding outward, we can start from the goal and grow reverse wavefronts backward through the state space. Then each point is labeled not by its distance *from the start*, but by its distance *to the goal*. These are called *cost-to-go* values. We will build on this viewpoint: in heuristic search, a cost-to-go estimate tells us which states seem closer to the goal and therefore more promising to explore; in Markov decision processes, a closely related idea appears as a *value function*, which assigns to each state a number measuring how good it is to be there. Either way, whether we propagate the wavefront forward from the start or backward from the goal, we use the same geometry of shortest-path distance to organize the state space.

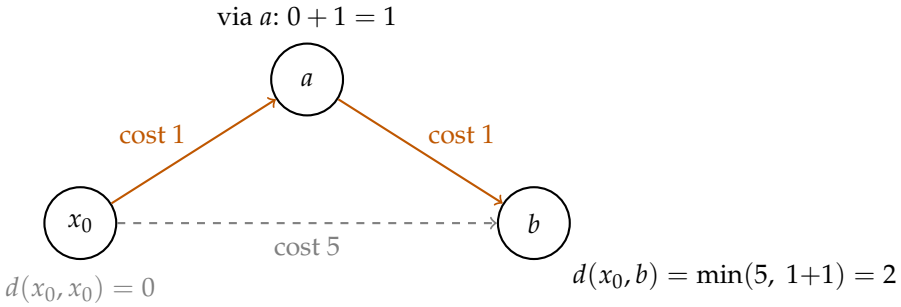


Figure 5.1: When step costs vary, "first discovered" is no longer "closest." The direct edge $x_0 \to b$ (gray, dashed) is found in a single step but costs 5. The two-step detour $x_0 \to a \to b$ (orange) costs only $1 + 1 = 2$, so it is the true shortest path. We therefore cannot finalize b the moment we first reach it; we must wait until no cheaper route remains.

Remark 3. So far each step has had unit cost. Under unit cost, "first discovered" and

"closest" are the same: a point reached in fewer steps is automatically nearer, which is why we could finalize each shell as soon as we reached it. With varying cost, this is no longer true. A point reached in one expensive step can be farther than a point reached in two cheap ones. Concretely, suppose one edge goes from x_0 directly to b with cost 5, while another route goes $x_0 \to a$ (cost 1) $\to b$ (cost 1), as in Figure 5.1. Discovering b first does not mean we have found its shortest distance; the direct edge gives 5, but the detour through a gives 2. So we can no longer finalize a point the moment we first see it. *In heuristic search, the same idea will come up as when we learn about admissibility.*

The fix is a single change to the order of expansion: always finalize the *nearest* unfinished point on the frontier, rather than the earliest discovered one. We keep the frontier sorted by distance-so-far and always remove the minimum. Unit cost is the special case where "nearest" and "earliest" coincide, which is why a plain queue suffices there and we recover breadth-first search. Sorting the frontier by distance instead gives *Dijkstra's algorithm* (uniform-cost search).

The reason this is correct has not changed. Write $c(y, b)$ for the cost of the step from y to b. The optimal-substructure relation from Remark 1 now reads

$$d(x_0, b) = \min_y \left[d(x_0, y) + c(y, b) \right],$$

where the minimum is taken over the points y from which b can be reached in one step. In words: the shortest distance to b is achieved by some predecessor y, and equals the shortest distance to y plus the cost of the final step. The unit-cost version $d(x_0, b) = d(x_0, y) + 1$ is the special case $c(y, b) = 1$. This is why finalizing the nearest frontier point is safe: once $d(x_0, y)$ is known for every candidate predecessor closer than b, the minimum above is determined, and no later (longer) path can improve it — provided costs are nonnegative, which they are, since they are distances. We will meet this equation again as the *Bellman equation* when we study value functions and Markov decision processes. So the general case has the same geometry and the same correctness argument, with one change to the order in which we expand the frontier.

5.4 Summary

This chapter introduced a couple of funadamental and powerful concepts:

- The function $d(x) : x \mapsto d(x_0, x)$ organizes the state space by distance from the starting point.
- The sublevel sets $B_i = \{x : d(x_0, x) \leq i\}$ collect everything reachable within distance i; the level sets $L_i = \{x : d(x_0, x) = i\}$ pick out the frontier exactly at distance i.
- The optimal substructure property of shortest paths allows us to build shortest paths incrementally: every subpath of a shortest path is itself shortest, so we can assemble long shortest paths from shorter ones already computed.

- In the unit-cost model these become "reachable in at most i steps" and "first reached in exactly i steps," respectively.
- The wavefront expansion computes shortest distances by growing these wavefronts one layer at a time. A shortest path is recovered by following the distance function backward from the target.

In short, dynamic programming computes geodesics by exploring the geometry of distance one layer at a time.

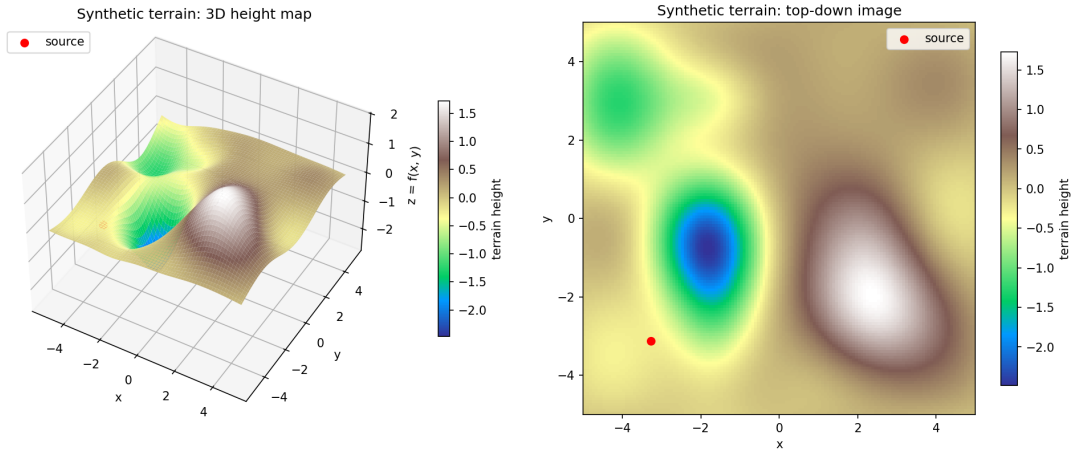


Figure 5.2: A terrain $z = f(x, y)$, built as a sum of Gaussian bumps and pits plus a gentle sinusoidal ripple. The left panel shows the terrain as a 3D height map, and the right panel shows the same terrain in a top-down view. The red marker denotes the source point x_0 used in the shortest-path computations.

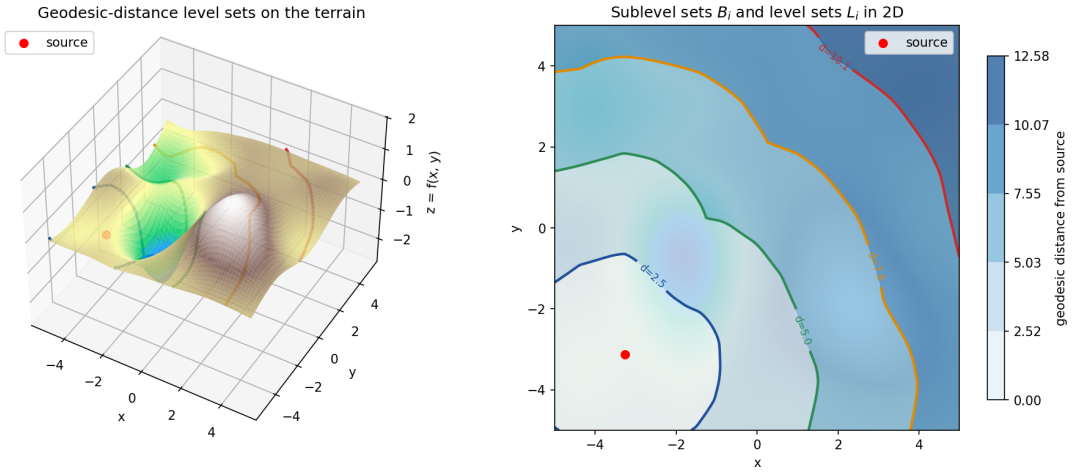


Figure 5.3: Geodesic-distance level sets and sublevel sets from the source x_0. Distances are measured along the surface: each step costs its 3D length $\sqrt{(\Delta x)^2 + (\Delta y)^2 + (\Delta z)^2}$, and the geodesic distance $d(x_0, x)$ is computed by Dijkstra's algorithm on the grid. In the left panel, curves of constant geodesic distance are drawn on the terrain surface. In the right panel, the shaded regions illustrate the sublevel sets $B_i = \{x : d(x_0, x) \leq i\}$ and the contour lines illustrate the level sets $L_i = \{x : d(x_0, x) = i\}$. The contours stay nearly circular near x_0 but bulge outward over the high terrain, where climbing adds surface length.

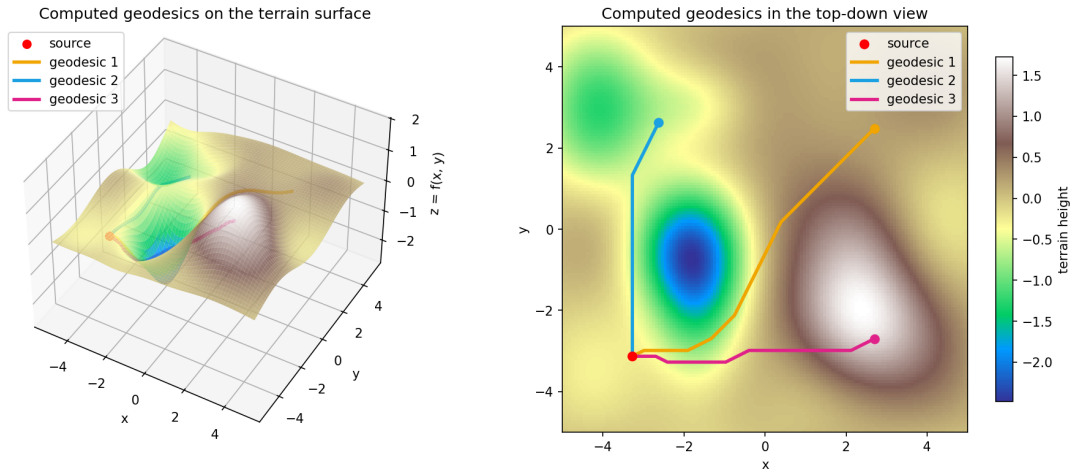


Figure 5.4: Examples of geodesics from the source to three goal points. The left panel shows the shortest paths on the 3D terrain surface; the right panel shows the same paths from a top-down view. Each path is recovered by tracing the Dijkstra predecessors backward from its goal. Notice that the paths bend around the deep pit and the tall peak rather than crossing them, since climbing costs extra surface length. Because the search runs on a grid graph, the recovered paths show slight axis-aligned staircasing; this is a discretization artifact, and finer grids (or fast-marching methods) converge to the true smooth geodesics.

Chapter 6

Gradients (and Geodesics)

In the previous chapter, we saw how we can compute all shortest paths from an origin using Dynamic Programming (DP). While DP is a very powerful and general technique, it has one major drawback: as it grows the wavefronts, it somehow needs to store which states belong to the wavefronts. Unless there is a compact description (such as an algebraic equation describing a circle or a sphere, or a trained neural-network), the size of the wavefront grows proportionally with the size of the state space. As we have already seen, state spaces can be very large!

In this section, we will learn about an alternative approach for computing geodesics. Along the way, we will introduce the important concept of a gradient.

Recall, from calculus, that the derivative of a function is another function that gives us the rate of change at an input location. For example, if $f(x) = x^2$, $f'(x) = df/dx = 2x$, which tells us that at $x = 1.3$, if we perturb x by a small amount δx, $f(x)$ will change by $2x\delta x = (2)(1.3)(\delta x)$. In other words, we get a first order approximation to f around x:

$$f(x + \delta x) \approx f(x) + f'(x)\delta x$$

(Can you recover the definition of derivative from this equation?)

Explore Further

Good opportunity to review Taylor series approximations and function approximations using them. You already derived the first order approximation!

The gradient is a vector which extends this concept to multivariate functions.

Definition. Let $f(\vec{x}) = f(x_1, x_2, \ldots, x_n)$ be a scalar function of n variables. The *partial derivative* $\partial f/\partial x_i$ is the rate of change of f as we vary the single coordinate x_i while holding

all the others fixed. The *gradient* of f is the *vector* that collects these partial derivatives:

$$\nabla f(\vec{x}) = \left[\frac{\partial f}{\partial x_1}, \frac{\partial f}{\partial x_2}, \ldots, \frac{\partial f}{\partial x_n} \right]^T.$$

So while f outputs a single number at each point, its gradient outputs a *vector* at each point — one component per coordinate direction.

As an example, take the two-variable analog of $f(x) = x^2$, namely $f(x, y) = x^2 + y^2$. Its partial derivatives are $\partial f/\partial x = 2x$ and $\partial f/\partial y = 2y$, so

$$\nabla f(x, y) = [\, 2x,\ 2y \,]^T.$$

At the point $(1.3, 0)$ the gradient is $[\, 2.6,\ 0 \,]^T$, and at $(1, 1)$ it is $[\, 2,\ 2 \,]^T$.

The gradient generalizes the derivative in a precise sense. In one dimension, perturbing x by a small δx changes f by approximately $f'(x)\,\delta x$ (the $2x\,\delta x$ from above). In several dimensions, perturbing $\vec{x}$ by a small vector $\delta\vec{x}$ changes f by approximately the *dot product* of the gradient with that perturbation:

$$f(\vec{x} + \delta\vec{x}) \;\approx\; f(\vec{x}) + \nabla f(\vec{x}) \cdot \delta\vec{x}.$$

The single number $f'(x)$ has become the vector $\nabla f(\vec{x})$, and ordinary multiplication has become the dot product.

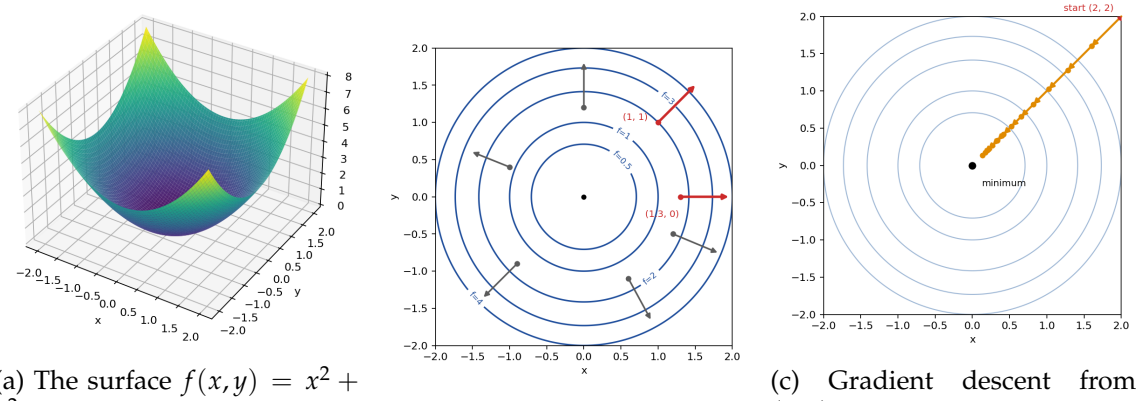
This dot product is largest when $\delta\vec{x}$ points in the same direction as ∇f. So the gradient has a clean geometric meaning: *it points in the direction of steepest increase of f*, and its length $\|\nabla f\|$ is the rate of increase in that direction. The negative gradient $-\nabla f$ points in the direction of steepest decrease. This last fact is the one we will build on: to move toward a goal, we will repeatedly take a small step in the direction $-\nabla f$.

6.1 Gradient Descent

The function $f(x, y) = x^2 + y^2$ is the (squared) cost-to-go to the origin for the Euclidean case. We now have a new method for our agent. If we have access to the cost-to-go function, we can compute paths to the goal by following the gradient. Since $-\nabla f$ points in the direction of steepest decrease, a simple way to move toward a minimum of f is to take repeated small steps in that direction. Starting from a point $\vec{x}_0$, we iterate

$$\vec{x}_{k+1} = \vec{x}_k - \eta\, \nabla f(\vec{x}_k),$$

where $\eta > 0$ is a small number called the *step size* (also called the *learning rate*). Each step nudges us a little downhill. This procedure is called *gradient descent*, and it is one of the workhorses of modern AI.



(a) The surface $f(x, y) = x^2 + y^2$.

(b) Level sets and gradients.

(c) Gradient descent from $(2, 2)$.

Figure 6.1: The function $f(x, y) = x^2 + y^2$ and its gradient. **(a)** The surface is a bowl with its minimum at the origin. **(b)** The level sets are concentric circles, and the gradient $\nabla f = [2x, 2y]^T$ (arrows, drawn at a common scale) points radially outward at every point, perpendicular to the level set through its base; the two red arrows are the worked examples $\nabla f(1.3, 0) = [2.6, 0]^T$ and $\nabla f(1, 1) = [2, 2]^T$. **(c)** Gradient descent, $x_{k+1} = x_k - \eta \nabla f(x_k)$ with step size $\eta = 0.1$, started at $(2, 2)$: it walks straight down to the minimum, taking large steps far out and progressively smaller ones as it approaches. The shrinking steps reflect the magnitude of the gradient, $\|\nabla f\| = 2\sqrt{x^2 + y^2}$, which grows linearly with distance from the minimum. This growth is exactly the upward curvature of the bowl: the walls steepen as you climb (the second derivative is constant, $\partial^2 f / \partial x^2 = 2$, so the slope increases at a constant rate), while near the bottom the surface is almost flat and the gradient nearly vanishes.

Let us apply it to our example $f(x, y) = x^2 + y^2$, starting at $(2, 2)$. Because $\nabla f = [2x,\ 2y]^T$, the update simply multiplies each coordinate by $(1 - 2\eta)$:

$$\vec{x}_{k+1} = \vec{x}_k - \eta\,[2x_k,\ 2y_k]^T = (1 - 2\eta)\,\vec{x}_k.$$

With $\eta = 0.1$, each coordinate shrinks by a factor of 0.8 at every step, so the iterates march steadily toward the minimum of our cost-to-go function: the goal at $(0, 0)$. Figure 6.1c shows the trajectory.

Two things are worth noting. First, the path is a straight line to the minimum. This is special to the symmetric bowl: the gradient is radial, so $-\nabla f$ points directly at the bottom from every point. For less symmetric functions, $-\nabla f$ does not aim straight at the minimum, and the descent path bends (See Figure 6.4). Second, the steps shrink as we approach the origin. The gradient magnitude $\|\nabla f\| = 2\sqrt{x^2 + y^2}$ is large far from the minimum and small near it, so gradient descent naturally takes big steps when it is far away and slows down as it closes in.

The step size η has to be chosen with care. If it is too small, progress is slow and we need many steps. If it is too large, we overshoot the minimum and can even diverge: for our example, the factor $(1 - 2\eta)$ has magnitude less than one only when $0 < \eta < 1$, and a larger η would send the iterates flying outward instead of inward. Much of the practical art of gradient descent lies in choosing η well [1].

Finally, let's go back to geodesics. Recall from the previous section the *cost-to-go* function, which assigns to each state its distance to the goal. If we descend that function — repeatedly stepping along its negative gradient — we trace a path that heads toward the goal as directly as the landscape allows. This gives an alternative to wavefront expansion: instead of growing and storing wavefronts over the whole state space, we follow the gradient from wherever we happen to be. We will return to this idea, and to its limitations, later in the course when we talk about heuristics. *You can think of heuristics as guesses about the cost-to-go function.* In Markov Decision Processes and RL, the cost-to-go is associated with the *value function whose gradient is associated with optimal actions.*

6.2 Example: Straightening a Path by Gradient Descent

Back in Chapter 2, we noted that a path can itself be written as a vector: a sequence of n waypoints is just a $2n$-dimensional vector of coordinates. At the time this was only an observation about representation. Now we can put it to work. If a path is a point in a high-dimensional space, and its length is a function of that point, then we can *descend* on length — and watch the path improve itself.

Fix the two endpoints $\vec{p}_0 = a$ and $\vec{p}_n = b$. The interior waypoints $\vec{p}_1, \ldots, \vec{p}_{n-1}$ are free, with $\vec{p}_i = (x_i, y_i)$. The path is the polyline $a \to \vec{p}_1 \to \cdots \to \vec{p}_{n-1} \to b$, and our decision

[1] Modern variants like Adam try to control the step size to improve convergence.

variable is the stack of free coordinates,

$$\vec{w} = (x_1, y_1, \ldots, x_{n-1}, y_{n-1}) \in \mathbb{R}^{2(n-1)}.$$

Writing $\vec{e}_i = \vec{p}_i - \vec{p}_{i-1}$ for the i-th segment, the length of the path is the sum of the segment lengths,

$$L(\vec{w}) = \sum_{i=1}^{n} \|\vec{e}_i\| = \sum_{i=1}^{n} \sqrt{(x_i - x_{i-1})^2 + (y_i - y_{i-1})^2}.$$

This is a scalar function of $\vec{w}$, exactly the kind of object we can run gradient descent on.

The gradient. To descend, we need ∇L. Consider one interior waypoint $\vec{p}_i$. It appears in exactly two terms of the sum: the segment $\vec{e}_i = \vec{p}_i - \vec{p}_{i-1}$ arriving at it, and the segment $\vec{e}_{i+1} = \vec{p}_{i+1} - \vec{p}_i$ leaving it. Every other term is constant in $\vec{p}_i$, so only these two matter.

The one fact we need is the gradient of a length. For a nonzero vector $\vec{u}$,

$$\nabla_{\vec{u}} \|\vec{u}\| = \frac{\vec{u}}{\|\vec{u}\|} = \hat{\vec{u}},$$

the unit vector pointing along $\vec{u}$. (In components, $\frac{\partial}{\partial u_x}\sqrt{u_x^2 + u_y^2} = u_x/\|\vec{u}\|$, and likewise for u_y.) Since $\vec{p}_i$ enters $\vec{e}_i$ with a $+$ sign and $\vec{e}_{i+1}$ with a $-$ sign, the chain rule gives

$$\nabla_{\vec{p}_i} L = \hat{\vec{e}}_i - \hat{\vec{e}}_{i+1} = \frac{\vec{p}_i - \vec{p}_{i-1}}{\|\vec{p}_i - \vec{p}_{i-1}\|} - \frac{\vec{p}_{i+1} - \vec{p}_i}{\|\vec{p}_{i+1} - \vec{p}_i\|}.$$

The full gradient stacks these 2-vectors for $i = 1, \ldots, n-1$; the fixed endpoints contribute nothing.

What the gradient says. The gradient at $\vec{p}_i$ is the difference of two unit vectors: the direction *into* the waypoint minus the direction *out of* it. This has a clean reading. It is zero exactly when $\hat{\vec{e}}_i = \hat{\vec{e}}_{i+1}$, that is, when the incoming and outgoing segments point the same way and the path passes *straight through* $\vec{p}_i$ with no kink. So a polyline is a critical point of length if and only if it has no corners — which, with the endpoints pinned, means it is the straight segment from a to b.

The descent step $-\nabla_{\vec{p}_i} L = \hat{\vec{e}}_{i+1} - \hat{\vec{e}}_i$ points along the bisector of the angle at the kink, pulling the waypoint inward to flatten the corner (Figure 6.2). A useful mental picture: imagine each segment as a string under unit tension; the net pull on the bead at $\vec{p}_i$ is exactly $\hat{\vec{e}}_{i+1} - \hat{\vec{e}}_i$, and gradient descent is the chain of strings relaxing until it is taut. Figure 6.3 shows this in action: starting from a deliberately wiggly path, each step pulls slack out of the corners, and the polyline relaxes toward the straight line.

This is the discrete version of a statement from Section 4: *a geodesic locally minimizes length*. Here "locally" is literal — the gradient only ever looks at a waypoint and its two

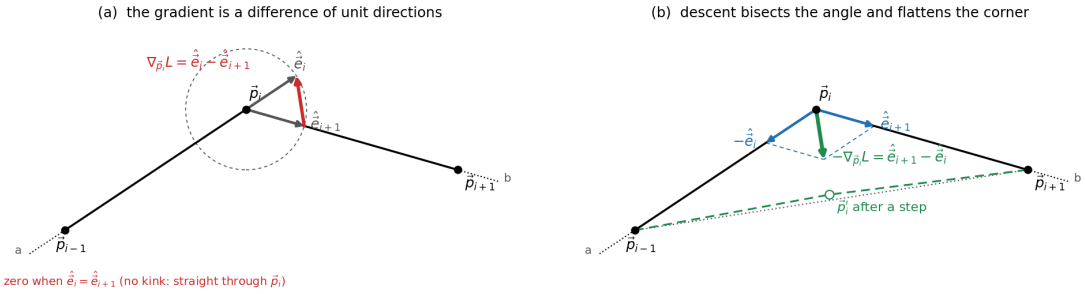


Figure 6.2: Geometry of the path-length gradient at a single interior waypoint $\vec{p}_i$. **(a)** Placing the two forward unit segment vectors $\hat{\vec{e}}_i$ (into $\vec{p}_i$) and $\hat{\vec{e}}_{i+1}$ (out of $\vec{p}_i$) tail-to-tail, their tip-to-tip connector is the gradient $\nabla_{\vec{p}_i} L = \hat{\vec{e}}_i - \hat{\vec{e}}_{i+1}$ (the dashed circle marks unit length). The gradient is zero exactly when the two directions coincide, i.e. the path runs straight through $\vec{p}_i$ with no kink. **(b)** The descent step $-\nabla_{\vec{p}_i} L = \hat{\vec{e}}_{i+1} - \hat{\vec{e}}_i$ is the resultant of two unit "tensions" pulling $\vec{p}_i$ toward its neighbours, $-\hat{\vec{e}}_i$ and $\hat{\vec{e}}_{i+1}$. It bisects the interior angle and points toward the chord $\vec{p}_{i-1}\vec{p}_{i+1}$; one step moves the waypoint to $\vec{p}_i'$, flattening the corner (dashed). Repeating this at every waypoint relaxes the polyline toward the straight segment.

neighbors, yet driving it to zero everywhere yields the globally straight path. If we replace the Euclidean segment length $\|\vec{e}_i\|$ with a length measured on a terrain or through a cost field, the very same construction applies: descent on the new length bends the path *around* expensive regions instead of straightening it, tracing a geodesic of the curved space. The straight line is simply what this procedure produces when the space is flat. Let's take a look at an example on a curved surface.

6.3 Local versus Global: The Same Descent on a Terrain

The flat-plane example was reassuring: descent on length pulled every path to the one global optimum, the straight line. But recall that a geodesic is only *locally* shortest. On a curved space, descending length can leave us with a geodesic that is not the globally shortest path based on the initialization. Let us see this happen.

We return to the terrain of Figures 5.2–5.4, and use exactly the construction from the previous example, with one change: the length of a path is now measured *along the surface*, so each little segment costs $\sqrt{(\Delta x)^2 + (\Delta y)^2 + (\Delta z)^2}$ instead of $\sqrt{(\Delta x)^2 + (\Delta y)^2}$. Everything else is the same — a polyline of waypoints, and gradient descent on its length. Climbing now costs extra length, so descent will try to keep the path low and short at the same time.

We fix a start and a goal on opposite sides of the tall peak and run the descent twice, from two different initial paths (Figure 6.4, left). One initialization is nudged to the north

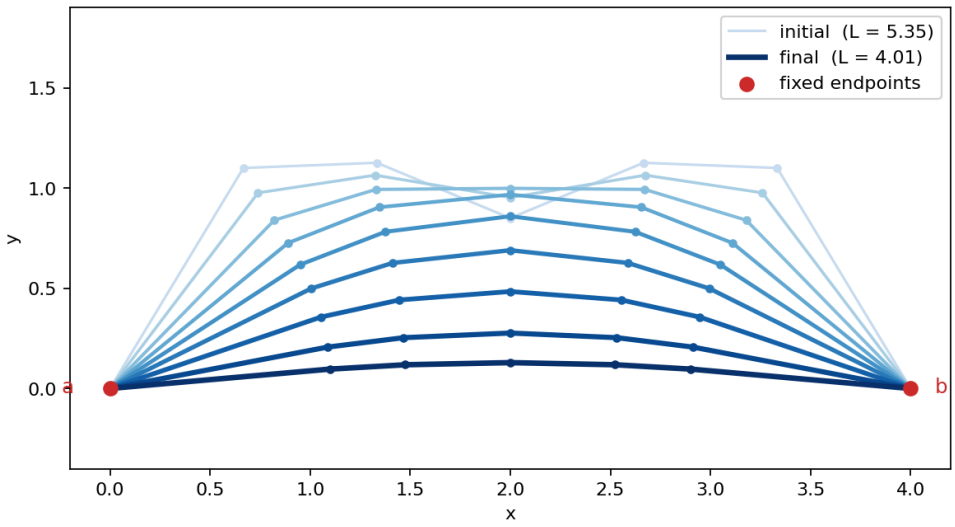


Figure 6.3: Gradient descent on the length of a piecewise-linear path with five free interior waypoints and fixed endpoints a and b (red). Starting from a wiggly initial path (lightest shade), each gradient step moves every waypoint along $-\nabla_{\vec{p}_i} L = \hat{\vec{e}}_{i+1} - \hat{\vec{e}}_i$, the bisector of the angle at that corner, pulling slack out of the path. Successive paths are drawn in darkening shades; the polyline relaxes toward the straight segment, the unique length-minimizing path with these endpoints (length $5.35 \to 4.01$, against a straight-line distance of 4.00). The corners flatten because the gradient at a waypoint vanishes only when the path passes straight through it — the discrete expression of local length-minimization.

of the peak; it relaxes to a short path that skims the peak's northern flank. The other is nudged to the south; it relaxes to a path that detours around the southern side. Both are genuine local minima: at each one, the length-gradient is zero, so every nearby path is at least as long. Yet they are clearly different paths, and one (the southern detour) is about 20% longer than the other.

Why does the southern path not simply slide over to join the shorter northern one? Because to get there it would have to drag itself *across the peak*, and going up and over the peak makes the path longer before it could ever get shorter. The descent only feels the length in the immediate neighborhood of the current path; it has no way to know that a better route exists on the far side of the mountain. Each path is trapped in its own valley — not a valley of the terrain, but a valley of the *length function* over the space of paths. This is the price of a local method: the answer it returns depends on where it started.

Contrast this with the method of Section 5. There we computed the cost-to-go function $d(x, x_f)$ — the geodesic distance from every point to the goal — by growing wavefronts over the whole terrain, and we can recover the shortest path by following $-\nabla d$ from the start (steepest descent on cost-to-go). Because that computation explores the entire space rather than the neighborhood of a single path, it finds the *globally* shortest route. In Figure 6.4 (right), this global optimum coincides with the shorter, northern path; the southern path that gradient descent happily returned is longer.

Both methods compute geodesics. The trapped southern path *is* a geodesic — it is locally length-minimizing, with no kinks — but it is not the shortest path. Cost-to-go descent returns the geodesic that is also globally shortest. The two approaches sit at opposite ends of a trade-off we will meet repeatedly: descending the length of a path is cheap and local, working with one candidate at a time, but it depends on a good initialization and can settle for a suboptimal answer; the dynamic-programming approach is global and guaranteed, but it pays for that guarantee by exploring and storing information about the entire state space. In the rest of this course, we will talk more about navigating between these two extremes.

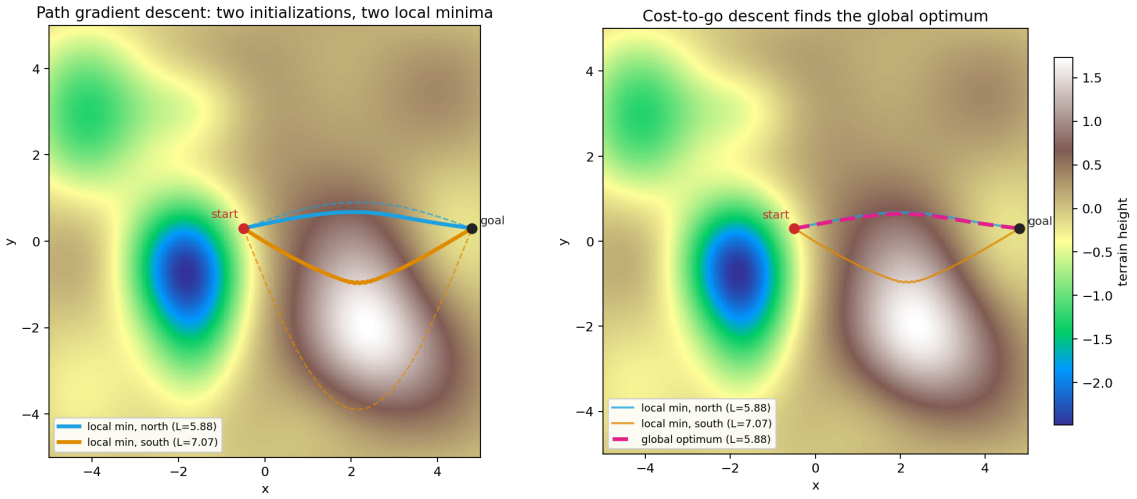


Figure 6.4: Local versus global shortest paths on the terrain, between a fixed start and goal placed on opposite sides of the tall peak. **Left:** gradient descent on *surface* path length, run from two initializations (dashed). The north initialization relaxes to a short path skimming the peak's northern flank ($L = 5.88$); the south initialization relaxes to a path that detours around the southern side ($L = 7.07$). Both are local minima of length — each has zero length-gradient, so every nearby path is at least as long — but they are different, and the southern one is about 20% longer. Neither can reach the other without first climbing over the peak, which would increase its length, so each is trapped in its own basin. **Right:** the global optimum (magenta), obtained by steepest descent on the cost-to-go function $d(x, x_f)$ computed by Dijkstra's algorithm over the whole terrain (Section 5). It coincides with the shorter northern path. The southern path is a perfectly valid *geodesic* — locally length-minimizing — but not the globally shortest route, illustrating the distinction of Section 4.

Chapter 7

Transformations

In this section, we study transformations between representations (more precisely, vector spaces). Such transformations appear throughout AI: Rather than working directly in a complicated space, we often map the data to a simpler space where the geometry is easier to understand or the computations are easier to perform. For example, a curved surface such as the surface of Earth may be mapped to a flat coordinate chart for visualization. Another common use is in multi-agent settings: Two sensing agents (e.g., a car and a drone performing habitat surveying) may need to map their measurements into a common coordinate frame before they can exchange information. Later, we will also view the layers of a neural network as transformations that act on vector representations.

7.1 The Canonical Frame

As we emphasized throughout the preceding chapters, a coordinate vector only becomes "physically localized" after we specify the frame in which it is expressed.

One thing you may have noticed is that, even though we developed the notation for specifying coordinate frames at the beginning of this chapter, we haven't used it much so far. This is because, when there is only one coordinate frame in play, we often rely on an implicit *canonical frame* to simplify the setting. In two dimensions, the canonical frame has origin

$$\vec{O} = \begin{bmatrix} 0 \\ 0 \end{bmatrix},$$

and basis vectors

$$\vec{x}_1 = \begin{bmatrix} 1 \\ 0 \end{bmatrix}, \qquad \vec{x}_2 = \begin{bmatrix} 0 \\ 1 \end{bmatrix}.$$

Therefore, a point with coordinates $\begin{bmatrix} c_1 \\ c_2 \end{bmatrix}$ can be written as

$$\vec{P} = \vec{O} + c_1\vec{x}_1 + c_2\vec{x}_2.$$

Equivalently,

$$\vec{P} = \begin{bmatrix} \vec{x}_1 & \vec{x}_2 \end{bmatrix} \begin{bmatrix} c_1 \\ c_2 \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} c_1 \\ c_2 \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \end{bmatrix} = \begin{bmatrix} c_1 \\ c_2 \end{bmatrix}.$$

Note that, in order to facilitate generalization, we will start indexing the axes rather than giving them names: $\vec{x}_1$ and $\vec{x}_2$ rather than by $\vec{x}$ and $\vec{y}$. This way, in n-dimensions, we can refer to the axes using indices e.g., $\vec{x}_1, \ldots, \vec{x}_n$. This is a useful place to notice the general form

$$f(x) = A\vec{x} + \vec{b},$$

where A is a transformation matrix and $\vec{b}$ is a translation (shift).

The convenience of the canonical frame is fragile. The moment we want to refer to an actual physical location or we need to coordinate across multiple frames, we need to know the coordinate frame precisely, including the units. A self-driving car has many sensors. Each one obtains measurements in its own coordinate frame. Imagine what would happen, if we used the same canonical frame for all of them when merging measurements! A famous cautionary tale is the 1999 loss of the Mars Climate Orbiter, which was caused by a mismatch of units across different components of the software. This motivates our first simple example.

Example: changing units. Suppose a position vector is currently expressed in meters and we would like to express the same physical displacement in millimeters. If we keep the origin fixed, the conversion is simply

$${}^{B}\vec{P} = \begin{bmatrix} 1000 & 0 \\ 0 & 1000 \end{bmatrix} {}^{A}\vec{P}.$$

Thus, if

$${}^{A}\vec{P} = \begin{bmatrix} c_1 \\ c_2 \end{bmatrix},$$

then

$${}^{B}\vec{P} = \begin{bmatrix} 1000c_1 \\ 1000c_2 \end{bmatrix}.$$

In this example, only the scale changed; the origin stayed at $\begin{bmatrix} 0 \\ 0 \end{bmatrix}$. We will keep that simplifying assumption in the next example as well.

7.2 Rotated Coordinate Frames

Now imagine two cameras observing the same event, but with image axes that are rotated relative to one another. We assume the two frames share the same origin, and only their axes differ. Let frame $\{A\}$ be the blue frame and frame $\{B\}$ be the orange frame in Figure 7.1. Camera B observes the point P and reports the coordinates

$$ {}^{B}\vec{P} = \begin{bmatrix} c_1 \\ c_2 \end{bmatrix}. $$

Our task is to express the same geometric vector in frame $\{A\}$.

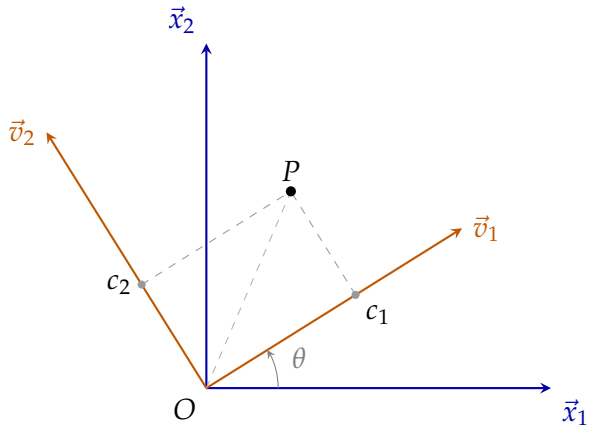


Figure 7.1: Two coordinate frames with a common origin. The blue axes define frame $\{A\}$ and the orange axes define frame $\{B\}$. The origin for both frames is $\vec{0}$. The point P is decomposed as $P = c_1\vec{v}_1 + c_2\vec{v}_2$, and the marked points on the rotated axes show the projections onto $\vec{v}_1$ and $\vec{v}_2$.

To build the transformation, let us first express the basis vectors of frame $\{B\}$ in frame $\{A\}$. In frame $\{B\}$, the basis vectors are, by definition,

$$ {}^{B}\vec{v}_1 = \begin{bmatrix} 1 \\ 0 \end{bmatrix}, \qquad {}^{B}\vec{v}_2 = \begin{bmatrix} 0 \\ 1 \end{bmatrix}. $$

Because frame $\{B\}$ is obtained by rotating frame $\{A\}$ by an angle θ, those same vectors have the coordinates

$$ {}^{A}\vec{v}_1 = \begin{bmatrix} \cos\theta \\ \sin\theta \end{bmatrix}, \qquad {}^{A}\vec{v}_2 = \begin{bmatrix} -\sin\theta \\ \cos\theta \end{bmatrix} $$

in frame $\{A\}$.

Now we have everything we need. Since ${}^B\vec{P} = \begin{bmatrix} c_1 \\ c_2 \end{bmatrix}$ means that

$$\vec{P} = c_1\vec{v}_1 + c_2\vec{v}_2,$$

we can express the same vector in frame $\{A\}$ by substituting the A-coordinates of the basis vectors:

$${}^A\vec{P} = c_1\,{}^A\vec{v}_1 + c_2\,{}^A\vec{v}_2 = \begin{bmatrix} {}^A\vec{v}_1 & {}^A\vec{v}_2 \end{bmatrix} \begin{bmatrix} c_1 \\ c_2 \end{bmatrix}.$$

Therefore,

$${}^A\vec{P} = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \begin{bmatrix} c_1 \\ c_2 \end{bmatrix} = {}^AR(\theta)_B\,{}^B\vec{P}.$$

We have just computed the 2×2 rotation matrix R. Let's make a few observations with an eye toward generalization.

- The mapping from ${}^B\vec{P}$ to ${}^A\vec{P}$ is a matrix multiplication. The matrix $R(\theta)$ is a two-dimensional rotation matrix parameterized by the rotation angle θ. Therefore, even though the matrix has $2 \times 2 = 4$ parameters (or variables), it has only one degree of freedom. Another way to see this is, there are 4 variables but three constraints (two for unity and one for orthogonality of the columns) bind their values.

- The additional notation ${}^AR_B$ makes the mapping direction clear:

  $${}^AR_B\,{}^B\vec{P} = {}^AR_{\not{B}}\,{}^{\not{B}}\vec{P} = {}^A\vec{P}.$$

  This way, we can keep track of coordinate frames when we chain them:

  $$({}^AR_B)\,({}^BR_C)\,{}^C\vec{P} = {}^AR_{\not{B}}\,{}^{\not{B}}R_{\not{C}}\,{}^{\not{C}}\vec{P} = {}^A\vec{P}.$$

- The columns of the rotation matrix are the basis vectors of frame $\{B\}$ expressed in frame $\{A\}$.

- Because those basis vectors are orthonormal (unit length and perpendicular to each other), the columns of the matrix are orthonormal as well.

- More generally, the jth column of R is the jth axis of frame $\{B\}$ expressed in frame $\{A\}$. Equivalently, the (i, j)th entry of the matrix is the projection of the jth axis of frame $\{B\}$ onto the ith axis of frame $\{A\}$:

  $$R_{ij} = \vec{v}_j \cdot \vec{x}_i$$

Explore Further

We derived the rotation matrix for the purpose of expressing $\vec{P}$ in two different coordinate frames. Under this interpretation, $\vec{P}$ does not physically move. But you can use exactly the same math and interpret it as an actual movement that moves P according to rotation action. Standard robotics texts such as Craig's book (where the ${}^{A}R_{B}$ notation is from) provide great starting points. Computer Vision or Graphics textbooks also cover this material in depth.

7.3 Beyond Rotations: General Linear Transformations

So far, we have focused on coordinate transformations between frames. More generally, a transformation is any rule that maps one vector to another,

$$T : \mathbb{R}^n \to \mathbb{R}^m.$$

When the transformation is linear, it can be written as a matrix multiplication:

$$\vec{y} = A\vec{x}.$$

Thus, linear transformations give us a general language for describing how one vector representation is converted into another.

Rotations are a particularly special kind of linear transformation. A rotation changes the direction of a vector without changing its length. Equivalently, it preserves distances, angles, and dot products. In matrix form, a rotation matrix has orthonormal columns and satisfies

$$R^T R = I.$$

So rotations preserve geometry: they re-express vectors without stretching or collapsing space.

Other linear transformations behave differently. A scaling stretches or shrinks vectors along certain directions. A projection maps vectors onto a lower-dimensional subspace, for example by dropping one coordinate or projecting a point onto a line. Reflections flip space across an axis or plane, and shears slide one part of the space relative to another. These transformations do not all preserve lengths or angles, but they are still described by matrices.

A powerful fact from linear algebra is that every matrix can be decomposed using the Singular Value Decomposition (SVD):

$$A = U\Sigma V^T.$$

Here, U and V are orthogonal matrices, so they act like rotations (or rotations combined with reflections), while Σ is diagonal and scales along special orthogonal directions. Geometrically, this means that every linear transformation can be viewed as three simpler steps:

1. rotate the input coordinates,
2. scale along perpendicular directions,
3. rotate again.

This point of view is captured in Figure 7.2.

This decomposition is conceptually very useful. It tells us that even a complicated linear map is built from a small collection of geometric primitives.

Despite their mathematical elegance, linear transformations by themselves are often limited (try finding a linear transformation to map the globe to a flat chart). As we saw, transformations involving rotations can be represented as a matrix multiplications and hence they are linear transformations. But we avoided translations which shift the origin. The general form of these transformations is $y = Ax + \vec{b}$. Even for the slightly general case when $\vec{b} \neq \vec{0}$, we need a more general form of transformations called *affine* transforms. See Appendix B for a formal definition and additional details.

Linear transfors are still useful in non-linear settings. They can efficiently represent "local" transformations (a small portion of the surface of the sphere can be mapped to a flat surface using a linear transform). They also provide building blocks for more complex transformations. Next we will look at a widely used representation – neural networks.

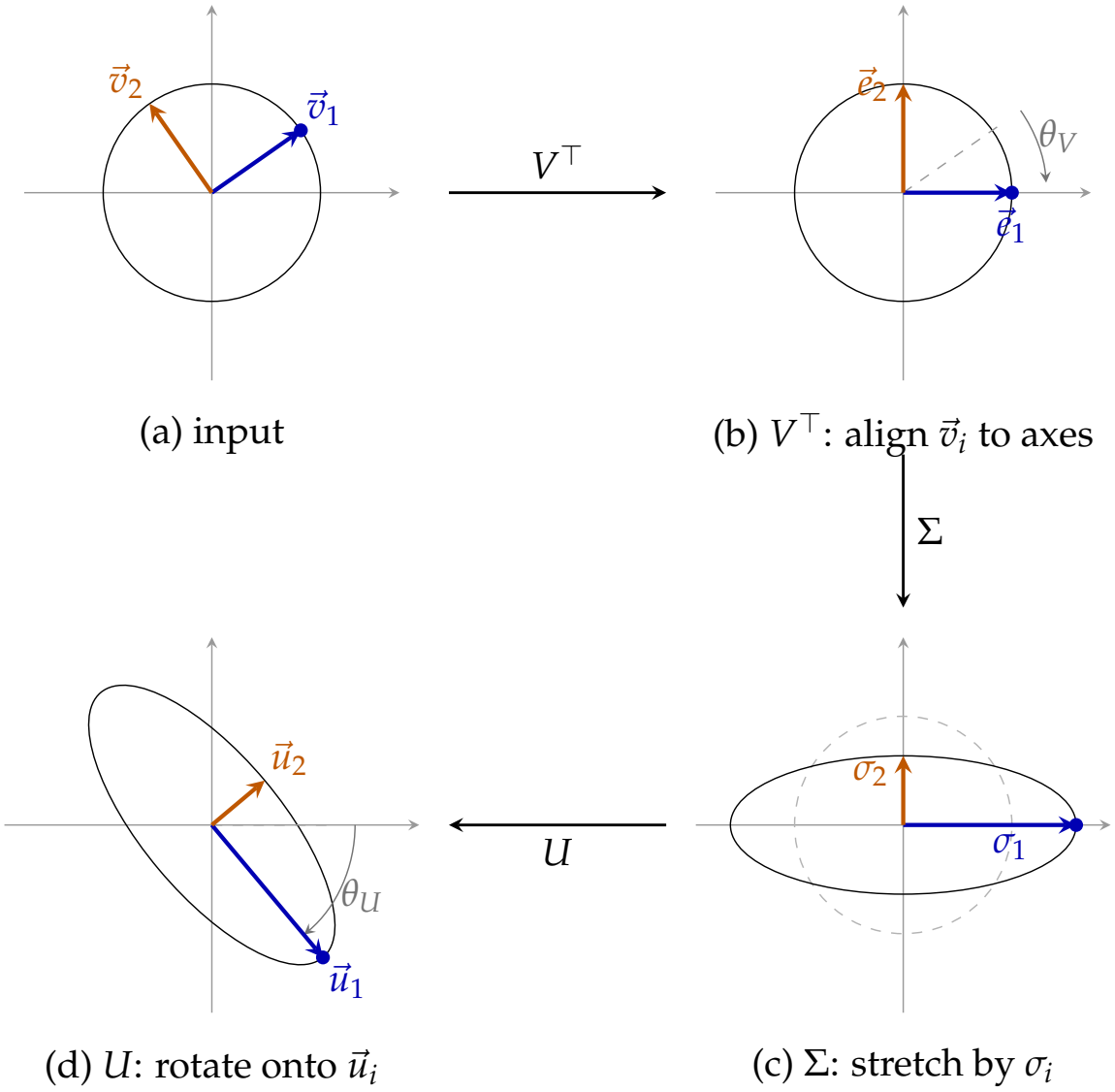


Figure 7.2: Geometric view of the SVD $A = U\Sigma V^\top$, obtained by tracking an orthonormal frame rather than the shape itself. **(a)** In the input space the right singular vectors $\vec{v}_1, \vec{v}_2$ (the columns of V) sit on the unit circle at some orientation. **(b)** $V^\top$ rotates this frame onto the coordinate axes, $\vec{v}_i \mapsto \vec{e}_i$, through the angle θ_V. **(c)** Σ stretches the axes by the singular values, $\vec{e}_i \mapsto \sigma_i \vec{e}_i$, turning the circle into an axis-aligned ellipse. **(d)** U rotates the stretched frame onto the left singular vectors, $\sigma_i \vec{e}_i \mapsto \sigma_i \vec{u}_i$, through a *different* angle θ_U. Following the blue direction across the four panels traces $\vec{v}_1 \mapsto \vec{e}_1 \mapsto \sigma_1 \vec{e}_1 \mapsto \sigma_1 \vec{u}_1 = A\vec{v}_1$. The two rotations are independent: $\theta_U \neq \theta_V$ in general, which is exactly why U and V are distinct orthogonal matrices and why the input frame $\{\vec{v}_i\}$ and the output frame $\{\vec{u}_i\}$ point in different directions.

Chapter 8

Neural Networks

Neural networks build transformations out of very simple ones such as linear maps. This section develops that idea from the ground up, starting with a single linear unit and a rectifier, and ending with the general matrix form of one layer. We will use the absolute value $|x|$ function as a running example. It is the simplest function that a purely linear (or affine) model cannot represent.

Do this:

Before we proceed, try representing $|x|$ as an linear function of the form $y = wx + b$. Note that we are switching our notation from $ax + b$ to be consistent with neural net terminology: w for weights and b for bias. Also note that as we discussed earlier and in Appendix B, y is actually an affine function but calling these units as linear units have become standard over the years.

Why $|x|$ needs more than a linear map: The absolute value is defined by

$$|x| = \begin{cases} x & x \geq 0 \\ -x & x < 0, \end{cases}$$

which is precisely a conditional — an "if" statement: *if x is non-negative return x, otherwise return $-x$*. Its graph is a V, with a kink at the origin and a nonzero, increasing slope on **both** sides.

A single affine map $y = wx + b$ produces only a straight line and can never bend, so it cannot represent this V. What we need is a mechanism that switches behavior depending on the sign of its input.

The rectifier. One of the simplest ways to introduce nonlinearities is to pass a linear function $y = ax + b$ through a *rectified linear unit* (ReLU),

$$\text{ReLU}(y) = \max(0, y) = \begin{cases} y & y \geq 0 \\ 0 & y < 0. \end{cases}$$

Applied to $y = x$, it gives $\text{ReLU}(x) = \max(0, x)$: negative inputs are set to zero, and non-negative inputs pass through unchanged (Figure 8.1). The result is piecewise linear with a single "kink" at the origin, flat on the left and rising with slope one on the right. The ReLU supplies exactly this capability: it turns a signal on or off according to whether the input is positive or negative, which is the arithmetic equivalent of the branch in the if statement above.

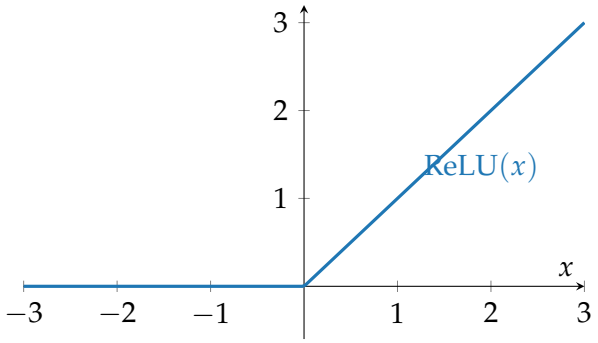


Figure 8.1: The rectifier $\text{ReLU}(x) = \max(0, x)$: zero for negative inputs, the identity for non-negative inputs, with a single kink at the origin.

$|x|$ as a sum of two ReLUs A single ReLU only handles one branch — it is active on one side of its kink and flat on the other. To cover both arms of the V we use two units, one for each branch, and add them:

$$|x| = \text{ReLU}(x) + \text{ReLU}(-x).$$

The two cases confirm this. When $x \geq 0$ we have $\text{ReLU}(x) = x$ and $\text{ReLU}(-x) = 0$, so the sum is x. When $x < 0$ we have $\text{ReLU}(x) = 0$ and $\text{ReLU}(-x) = -x$, so the sum is $-x$. In both regions the sum equals $|x|$ (Figure 8.2).

Each term is a *linear + ReLU unit* of the form $\text{ReLU}(wx + b)$: the first uses $(w, b) = (1, 0)$, the second uses $(w, b) = (-1, 0)$. The first unit reproduces the positive branch and the second, with its negated weight, reproduces the negative branch. Together the two units implement the two-way conditional that a single linear map could not.

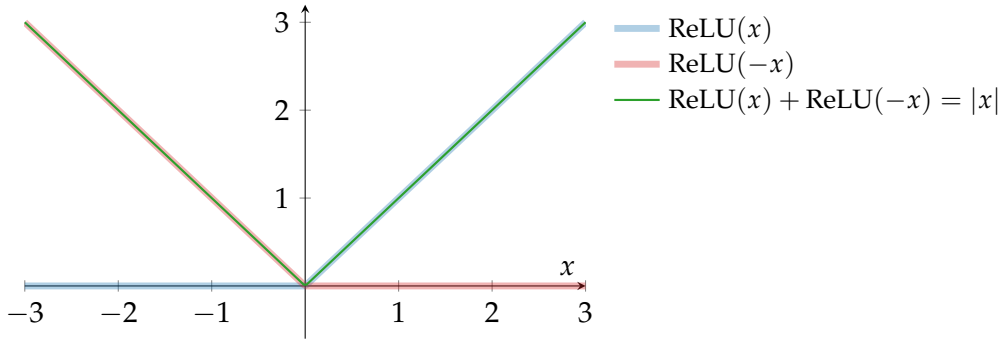


Figure 8.2: Two rectifiers with opposite weights, $\mathrm{ReLU}(x)$ and $\mathrm{ReLU}(-x)$ (translucent bands), each supplying one arm of the V. Their sum (solid line) is exactly $|x|$.

Shifting and scaling the kink The same construction generalizes once we allow the weight and bias to vary. The general one-dimensional linear unit is

$$\mathrm{ReLU}(wx + b),$$

with a slope w and a bias b. It is convenient to read the affine argument in the factored form

$$wx + b = w(x - t), \qquad t = -\frac{b}{w}$$

because t is exactly the location of the kink — the point where the argument vanishes and the unit switches on. This gives the two controls a clear meaning:

- the **bias** b shifts the kink along the axis (with $w = 1$, the unit $\mathrm{ReLU}(x + b)$ turns on at $x = -b$, sliding the corner left or right); You can view this as the translation step we omitted in the previous section.
- the **slope** w sets how steeply the active arm rises.

The shifted and scaled V Combining the two ideas gives a V of arbitrary position and steepness. Applying the two-unit construction to the general affine argument $wx + b$,

$$|wx + b| = \mathrm{ReLU}(wx + b) + \mathrm{ReLU}\big(-(wx + b)\big),$$

produces a V whose kink lies at $x = -b/w$ and whose arms have slope $|w|$. Checking the branches: where $wx + b \geq 0$ the first term equals $wx + b$ and the second is zero; where $wx + b < 0$ the first term is zero and the second equals $-(wx + b)$. In both cases the sum is $|wx + b|$.

Figure 8.3 shows a representative example, $w = 2$ and $b = -2$, giving $|2x - 2|$: a V with its kink shifted to $x = 1$ and arms of slope 2. The bias positions the corner and the

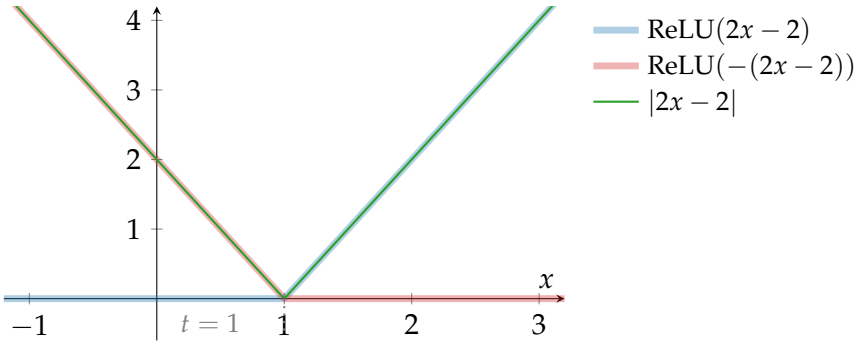


Figure 8.3: The shifted and scaled V with $a = 2$, $b = -2$. The two rectifiers (translucent bands) sum to $|2x - 2| = 2\,|x - 1|$, a V with kink at $x = t = -b/a = 1$ and arm slope $|a| = 2$.

weight scales the slope, so the two units can place a kink of any steepness anywhere on the axis.

This is the key to representing complex functions: each unit contributes one kink, and by summing many shifted and scaled ReLUs one can assemble any continuous piecewise-linear shape, approximating any continuous function on a bounded domain to any desired accuracy.

General matrix form of a linear + ReLU layer Stacking many such units and moving to vector-valued inputs gives one full layer of a network. For an input $\vec{x} \in \mathbb{R}^n$, a layer of m units computes

$$\vec{h} = \mathrm{ReLU}(W\vec{x} + \vec{b}),$$

where $W \in \mathbb{R}^{m \times n}$ is the weight matrix, $\vec{b} \in \mathbb{R}^m$ is the bias vector, and $\mathrm{ReLU}(\cdot)$ is applied elementwise. Row i of W together with entry b_i defines the single unit $\mathrm{ReLU}(\vec{w}_i^{\top}\vec{x} + b_i)$, the multivariate version of $\mathrm{ReLU}(ax + b)$ from above.

A linear output layer then combines these hidden units into the final value,

$$y = \vec{v}^{\top}\vec{h} + d = \sum_{i=1}^{m} v_i\ \mathrm{ReLU}(\vec{w}_i^{\top}\vec{x} + b_i) + d,$$

which is exactly a weighted sum of linear + ReLU units — the general form of the constructions above.

The absolute value is the smallest nontrivial instance of this template: with $n = 1$, $m = 2$,

$$W = \begin{bmatrix} 1 \\ -1 \end{bmatrix}, \quad \vec{b} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}, \quad \vec{v} = \begin{bmatrix} 1 \\ 1 \end{bmatrix}, \quad d = 0,$$

the layer reproduces $|x| = \mathrm{ReLU}(x) + \mathrm{ReLU}(-x)$. Widening the layer (larger m) adds more kinks, and stacking layers composes these piecewise-linear maps, which is how networks scale from a single corner to richly structured functions.

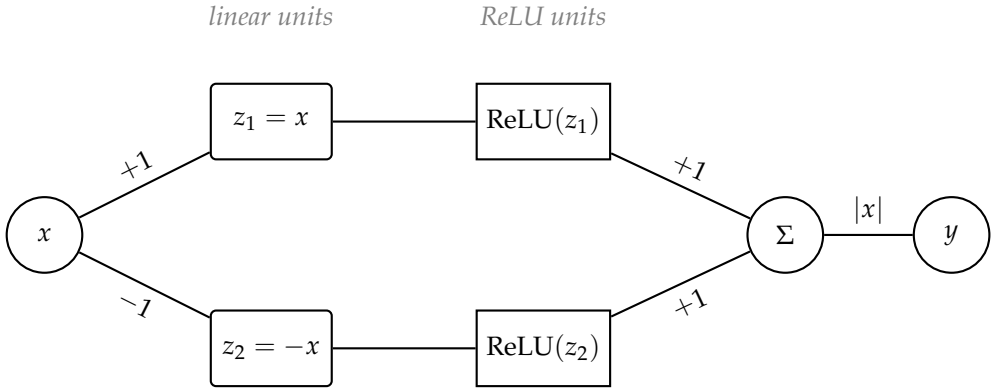


Figure 8.4: Network realizing $|x| = \mathrm{ReLU}(x) + \mathrm{ReLU}(-x)$. The input x is fed to two linear units with weights $+1$ and -1 (both biases zero), giving $z_1 = x$ and $z_2 = -x$. Each is passed through a ReLU, and the two activations are summed with output weights $+1$ to produce $y = |x|$.

Here is a code snippet that implements the network in Figure 8.4 in PyTorch.

```
import torch
import torch.nn as nn

class AbsValueNet(nn.Module):
    def __init__(self):
        super().__init__()
        self.hidden = nn.Linear(1, 2)    # z = W x + b
        self.relu = nn.ReLU()
        self.output = nn.Linear(2, 1)    # y = v^T h + d
        self._set_weights()

    def _set_weights(self):
        with torch.no_grad():
            self.hidden.weight.copy_(torch.tensor([[1.0], [-1.0]]))
            self.hidden.bias.copy_(torch.tensor([0.0, 0.0]))
            self.output.weight.copy_(torch.tensor([[1.0, 1.0]]))
            self.output.bias.copy_(torch.tensor([0.0]))

    def forward(self, x):
        return self.output(self.relu(self.hidden(x)))  # y = |x|
```

In the code above, we set the weights for the neural network by hand. The neural networks used in modern AI systems have far more parameters than we could ever set manually. A standard image-classification network such as ResNet-50 has about 25 million weights, and large language models are bigger by several orders of magnitude[1]. Setting even a tiny fraction of these weights by hand would be hopeless.

This where the true power of neural networks as representations lies: rather than being programmed with hand-set weights, they *learn* their weights from examples or from experience. We supply data, and an optimization procedure adjusts the parameters until the network produces the behavior we want. In fact, we have already seen the essential tool for this: gradient descent (Section 6), which here descends not the length of a path but a measure of how wrong the network's outputs are.

Next, we will go into what it means to "learn" the weights, and the basic methods for doing so.

[1]Historical note: I asked Claude to give me the number of parameters for the specific model it is running. It "thought" for a while (while I was having interesting thoughts about self-awarenes) and said: "I can't give a real parameter count for the specific model I'm running (Claude Opus 4.8). Anthropic doesn't publicly disclose parameter counts for Claude models, which is now standard practice among frontier labs." In fact, there is more – See **??**.

Chapter 9

Estimation, Model Fitting and Probabilities

In this final stop of our tour, we will take a look at learning representations. Along the way we will introduce basics of probabilistic representations and estimation.

Let us start with a simple example. Suppose we would like to estimate the temperature in a room. To do so, we take a measurement from a thermometer, and say it reads 72°F. What is the temperature?

We all know that the reading is not the actual temperature. In fact, what "the temperature in the room" even means is not well defined. It varies from point to point and from moment to moment. We will therefore carefully distinguish between two quantities:

- the *measurement* z, the number our sensor reports, and
- the *state* x, the true quantity we are trying to estimate.

The measurement and the state are related to each other by a *model*. In this case we have a very simple model,

$$z = x, \tag{9.1}$$

which says that the thermometer reports the true temperature exactly. In general, the model could be an arbitrary function of x.

Given this model and the single reading of 72°F, it is reasonable to estimate the temperature in the room as

$$\hat{x} = 72°\text{F},$$

where the hat in $\hat{x}$ emphasizes that this is an *estimate* of x rather than the true value.

Now suppose we have a second thermometer, take another measurement with it, and it reads 68°F. How should this second reading impact our estimate?

9.1 Least Squares Error

To formalize, we keep the model $z = x$ from Equation (9.1) and now have two measurements z_1 and z_2 (in general n measurements $z_1, \ldots, z_n$). We would like to come up with a single estimate $\hat{x}$ that is "good" in some precise sense.

Residuals. A reasonable thing to ask of our estimate is that it not be too far from the measurements. For each measurement z_i we define the *residual*

$$r_i(x) = z_i - x, \tag{9.2}$$

the discrepancy between the i-th measurement and a candidate estimate x.

We might first try to choose x so as to minimize the sum of the residuals, $\sum_{i=1}^{n} r_i(x)$. However, because residuals can be negative, a direct sum is not meaningful: large residuals of opposite sign can cancel each other out and still yield a small total, even when the individual errors are large. An algebraic trick that fixes this is to *square* the residuals before summing, which gives us the *least squares error* (LSE) formulation:

$$\mathrm{LSE}(x; z) \;=\; J(x) \;=\; \frac{1}{2}\sum_{i=1}^{n}\big(z_i - x\big)^2. \tag{9.3}$$

Squaring makes every term non-negative (so cancellation is impossible) and penalizes large deviations more heavily than small ones. The factor $\frac{1}{2}$ is included only for convenience: it cancels the 2 that appears when we differentiate, and it does not change the minimizer.

Solving for the estimate. For this simple example we can solve for the best estimate directly, by setting the derivative of the error function to zero:

$$\frac{dJ}{dx} = 0 \;\;\Longrightarrow\;\; \frac{1}{2}\cdot 2\sum_{i=1}^{n}\big(z_i - x\big)\cdot(-1) = 0.$$

Dropping the constant factor -1 and the $\frac{1}{2}\cdot 2 = 1$, this is

$$\sum_{i=1}^{n}\big(z_i - x\big) = 0 \;\;\Longrightarrow\;\; \sum_{i=1}^{n} z_i - \sum_{i=1}^{n} x = 0 \;\;\Longrightarrow\;\; \sum_{i=1}^{n} z_i - n\,x = 0,$$

where we used $\sum_{i=1}^{n} x = nx$ because x does not depend on i. Solving for x gives

$$n\,x = \sum_{i=1}^{n} z_i \;\;\Longrightarrow\;\; \hat{x} = \frac{1}{n}\sum_{i=1}^{n} z_i. \tag{9.4}$$

We have just derived that the *sample mean* is the estimate that minimizes the least squares error. For our two thermometer readings,

$$\hat{x} = \frac{72 + 68}{2} = 70^{\circ}\mathrm{F}.$$

9.2 Batch versus Recursive Estimation

The least squares estimate of the previous section was a *batch* estimate: we used all available measurements at once to compute the sample mean in Equation (9.4). Alternatively, we could keep a *running average* that we update as each new measurement arrives. Let μ_k denote the average of the first k measurements,

$$\mu_k = \frac{1}{k} \sum_{i=1}^{k} z_i. \tag{9.5}$$

When the $(k+1)$-th measurement z_{k+1} arrives, we would like to obtain μ_{k+1} without recomputing the whole sum from scratch. Starting from the definition and splitting off the newest term,

$$\mu_{k+1} = \frac{1}{k+1} \sum_{i=1}^{k+1} z_i = \frac{1}{k+1} \left(\sum_{i=1}^{k} z_i + z_{k+1} \right) = \frac{1}{k+1} \big(k\, \mu_k + z_{k+1} \big),$$

where in the last step we used $\sum_{i=1}^{k} z_i = k\, \mu_k$ from Equation (9.5). Distributing the $\frac{1}{k+1}$ gives the recursive update

$$\mu_{k+1} = \frac{k}{k+1}\, \mu_k + \frac{1}{k+1}\, z_{k+1}. \tag{9.6}$$

We can interpret Equation (9.6) as a weighted average of the old estimate μ_k and the new measurement z_{k+1}. As the number of measurements k grows, the weight $\frac{k}{k+1}$ on the running average approaches 1, so we trust the accumulated average more and more and each individual new reading less and less.

To gain further insight, rewrite the coefficient of μ_k using $\frac{k}{k+1} = 1 - \frac{1}{k+1}$ (equivalently, add and subtract $\frac{1}{k+1}\mu_k$ on the right-hand side of Equation (9.6)):

$$\mu_{k+1} = \mu_k - \frac{1}{k+1}\, \mu_k + \frac{1}{k+1}\, z_{k+1} = \mu_k + \frac{1}{k+1} \big(z_{k+1} - \mu_k \big),$$

which we write as

$$\mu_{k+1} = \mu_k - \frac{1}{k+1} \big(\mu_k - z_{k+1} \big). \tag{9.7}$$

Compare Equation (9.7) with the gradient descent update from Chapter 6,

$$x_{k+1} = x_k - \eta\, f'(x_k).$$

The two have exactly the same shape. The step size (learning rate) is $\eta_k = \frac{1}{k+1}$, and the term $\mu_k - z_{k+1}$ plays the role of the gradient $f'(x_k)$: indeed, it is the derivative with respect to μ_k of the single-measurement squared error $\frac{1}{2}(\mu_k - z_{k+1})^2$. We see that our recursive

estimator is essentially performing gradient descent to minimize the least squares error, one measurement at a time. It gradually reduces the learning rate ($\frac{1}{k+1} \to 0$), which is exactly what ensures that, in the end, all samples are weighted equally — recovering the same sample mean as the batch estimate.

When we study reinforcement learning, we will see a variation on this formula that keeps the learning rate from shrinking to zero, thereby putting more emphasis on more recent samples.

9.3 Enter Probabilities

So far, our models have been deterministic. In this last section of the chapter, we will look at *probabilistic* formulations, which allow us to model uncertainty in the environment (for example, a noisy sensor) as well as our lack of complete information in certain scenarios. I am assuming familiarity with basic probability theory. For a refresher, the basic definitions of random variables, densities, the Gaussian distribution, and Bayes' formula are available in Appendix C.

Recall our model $z = x$. We can explicitly model *why* a measurement differs from the true temperature by adding a noise term to the formulation:

$$z = x^* + \varepsilon, \qquad \varepsilon \sim \mathcal{N}(0, \sigma^2), \tag{9.8}$$

where x^* is the (fixed but unknown) true temperature and ε is a random variable drawn from the normal (Gaussian) distribution with mean 0 and variance σ^2; the parameter σ captures how noisy the sensor is. With this formulation the measurement z also becomes a random variable, centered at x^*.

The likelihood of a measurement. Using the Gaussian density (Appendix C), we can now write the *likelihood* of observing a particular reading z, for a given true value x:

$$\Pr[z \mid x] = \frac{1}{\sigma\sqrt{2\pi}} \exp\left(-\frac{(z-x)^2}{2\sigma^2}\right). \tag{9.9}$$

For multiple measurements $z_1, \ldots, z_k$, and assuming the noise values are *independent*, the joint likelihood factorizes into a product:

$$\begin{aligned}\Pr[z_1, \ldots, z_k \mid x] &= \Pr[z_1 \mid x]\ \Pr[z_2 \mid x] \cdots \Pr[z_k \mid x] \\ &= \prod_{i=1}^{k} \frac{1}{\sigma\sqrt{2\pi}} \exp\left(-\frac{(z_i-x)^2}{2\sigma^2}\right) = C \exp\left(-\frac{1}{2\sigma^2}\sum_{i=1}^{k}(z_i-x)^2\right),\end{aligned} \tag{9.10}$$

where $C = \left(\sigma\sqrt{2\pi}\right)^{-k}$ collects all the constant factors that do not depend on x, and we used the fact that a product of exponentials is the exponential of the sum of the exponents.

Maximizing the likelihood. Our goal is to find the value of x that maximizes this probability, also known as the *likelihood* of having received the particular measurements $z_1, \ldots, z_k$. A standard trick is to work with the *log-likelihood* instead of the likelihood itself. Because the logarithm is an increasing function, the value of x that maximizes the likelihood also maximizes its logarithm — taking the log does not change the location of the optimum. Taking the log of Equation (9.10) gives

$$\log\left(C\, e^{-\frac{1}{2\sigma^2}\sum_i (z_i - x)^2}\right) = \log C \; - \; \frac{1}{2\sigma^2}\sum_{i=1}^{k}(z_i - x)^2. \tag{9.11}$$

Here we realize that we have arrived back at the same least squares formulation of Equation (9.3). When we maximize Equation (9.11) over x, the constant $\log C$ plays no role (its derivative vanishes), and the positive factor $\frac{1}{2\sigma^2}$ is just a scale that does not affect the location of the optimum. So maximizing the log-likelihood is the same as *minimizing* $\sum_i (z_i - x)^2$, which is exactly the least squares error. The maximizing estimate is therefore once again the sample mean of Equation (9.4).

> Our derivations yielded a key observation: the least squares estimate we obtained from a purely algebraic argument turns out to be the *maximum likelihood estimate* under the assumption of independent Gaussian measurement noise. Always keep this in mind: when you are taking averages, or "minimizing L2 Loss", you are assuming independent Gaussian noise model.

Explore Further

We assumed the noise on every measurement had the *same* variance σ^2. What changes if some sensors are more reliable than others, i.e. each measurement z_i has its own variance σ_i^2? Redo the log-likelihood argument and see how the estimate becomes a *weighted* average. Which measurements get more weight, and does that match your intuition?

9.4 The vector-valued linear case

The scalar model $z = x$ is a useful first step, but in many applications both the state and the measurement are vectors. Let $\vec{x} \in \mathbb{R}^p$ denote the unknown state, let $\vec{z} \in \mathbb{R}^m$ denote the measurement vector, and let $H \in \mathbb{R}^{m \times p}$ be a known matrix. The model is

$$\vec{z} = H\vec{x}. \tag{9.12}$$

When the measurements are noisy or when the system is overdetermined, there may be no vector $\vec{x}$ that satisfies this equation exactly. As in the scalar case, we therefore choose the estimate that minimizes a squared residual.

Define the residual vector

$$\vec{r}(\vec{x}) = \vec{z} - H\vec{x}.$$

The least-squares objective is

$$J(\vec{x}) = \frac{1}{2}(\vec{z} - H\vec{x})^T(\vec{z} - H\vec{x}). \tag{9.13}$$

Expanding the quadratic gives

$$\begin{aligned} J(\vec{x}) &= \frac{1}{2}\Big(\vec{z}^T\vec{z} - \vec{z}^T H\vec{x} - \vec{x}^T H^T\vec{z} + \vec{x}^T H^T H\vec{x}\Big) \\ &= \frac{1}{2}\Big(\vec{z}^T\vec{z} - 2\vec{x}^T H^T\vec{z} + \vec{x}^T H^T H\vec{x}\Big), \end{aligned} \tag{9.14}$$

where we used the fact that the scalar $\vec{z}^T H\vec{x}$ equals its transpose $\vec{x}^T H^T\vec{z}$.

Now differentiate with respect to $\vec{x}$. The constant term $\vec{z}^T\vec{z}$ disappears, the derivative of $-\vec{x}^T H^T\vec{z}$ is $-H^T\vec{z}$, and because $H^T H$ is symmetric, the derivative of $\frac{1}{2}\vec{x}^T H^T H\vec{x}$ is $H^T H\vec{x}$. Therefore

$$\nabla_{\vec{x}} J(\vec{x}) = H^T H\vec{x} - H^T\vec{z}. \tag{9.15}$$

Setting the gradient to zero gives the *normal equations*

$$H^T H\hat{\vec{x}} = H^T\vec{z}. \tag{9.16}$$

If H has full column rank, then $H^T H$ is invertible and the least-squares estimate is

$$\hat{\vec{x}} = (H^T H)^{-1} H^T\vec{z}. \tag{9.17}$$

In the general case, the same idea is written compactly using the pseudoinverse:

$$\hat{\vec{x}} = H^{\dagger}\vec{z}. \tag{9.18}$$

Explore Further

We presented a basic introduction to estimation theory. Appendix C makes the notions of likelihood, maximum likelihood, and the closely related maximum a posteriori estimate precise and gives you a starting point for further exploration.

9.5 Geometry of the solution: projection and the pseudoinverse

The least-squares solution has a very clean geometric meaning. The set of all vectors of the form $H\vec{x}$ is the *range* (or column space) of H, denoted $\mathcal{R}(H)$. This is a subspace of the measurement space $\mathbb{R}^m$. When the system $\vec{z} = H\vec{x}$ cannot be solved exactly, we look for the point in $\mathcal{R}(H)$ that lies closest to the observed measurement vector $\vec{z}$.

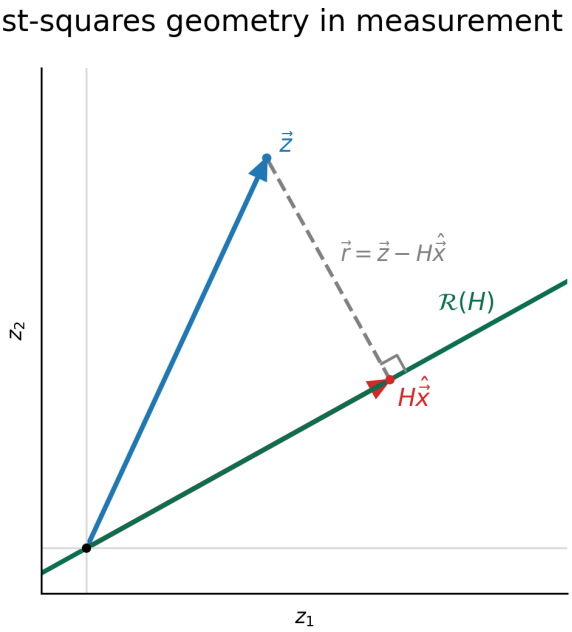


Figure 9.1: Least-squares geometry in measurement space. The fitted vector $H\hat{\vec{x}}$ is the orthogonal projection of $\vec{z}$ onto the column space $\mathcal{R}(H)$, and the residual $\vec{r} = \vec{z} - H\hat{\vec{x}}$ is perpendicular to that subspace.

That closest point is $H\hat{\vec{x}}$, and the residual

$$\vec{r} = \vec{z} - H\hat{\vec{x}}$$

is orthogonal to the entire column space of H. In other words,

$$H^T(\vec{z} - H\hat{\vec{x}}) = \vec{0}, \tag{9.19}$$

which is exactly the normal equation from above. So the normal equation is not merely an algebraic trick: it says that at the optimum the error vector is perpendicular to every direction that can be produced by the model.

It is worth being precise about where the projection lives. The projection is naturally a statement in *measurement space*: $H\hat{\vec{x}}$ is the orthogonal projection of $\vec{z}$ onto $\mathcal{R}(H)$. The pseudoinverse $H^\dagger$ then maps that projected point back to a coefficient vector $\hat{\vec{x}}$ in parameter space. If there are many vectors $\vec{x}$ that generate the same projected measurement, the pseudoinverse selects the one with minimum Euclidean norm.

9.6 Example: fitting a line to data

A very important special case of least squares is *model fitting*. Suppose we are given samples $(x_1, y_1), \ldots, (x_n, y_n)$ and would like to fit a line

$$y = mx + n$$

to these data. The parameters are now the slope and intercept,

$$\vec{\theta} = \begin{bmatrix} m \\ n \end{bmatrix},$$

and the measurement vector collects the observed y-values,

$$\vec{z} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix}.$$

For each sample, the model predicts $y_i \approx mx_i + n$, so the matrix H becomes

$$H = \begin{bmatrix} x_1 & 1 \\ x_2 & 1 \\ \vdots & \vdots \\ x_n & 1 \end{bmatrix}, \qquad \vec{z} \approx H\vec{\theta}. \tag{9.20}$$

The least-squares estimate is therefore

$$\hat{\vec{\theta}} = \begin{bmatrix} \hat{m} \\ \hat{n} \end{bmatrix} = (H^T H)^{-1} H^T \vec{z}, \tag{9.21}$$

provided H has full column rank.

For the noisier data set used in Figure 9.2, the fitted values are

$$\hat{m}_{\mathrm{LS}} \approx 1.194, \qquad \hat{n}_{\mathrm{LS}} \approx 0.764.$$

The dashed vertical segments show the residuals: the least-squares line is the one that minimizes the sum of the squared lengths of these vertical discrepancies.

9.7 Solving the same fitting problem by gradient descent

The normal equation gives a closed-form answer for the linear case. For more general cases, we may need to resort to a non-linear method. Gradient-descent comes the rescue! For the line-fitting example, the loss is

$$J(m, n) = \frac{1}{2} \sum_{i=1}^{N} \big(y_i - (mx_i + n)\big)^2. \tag{9.22}$$

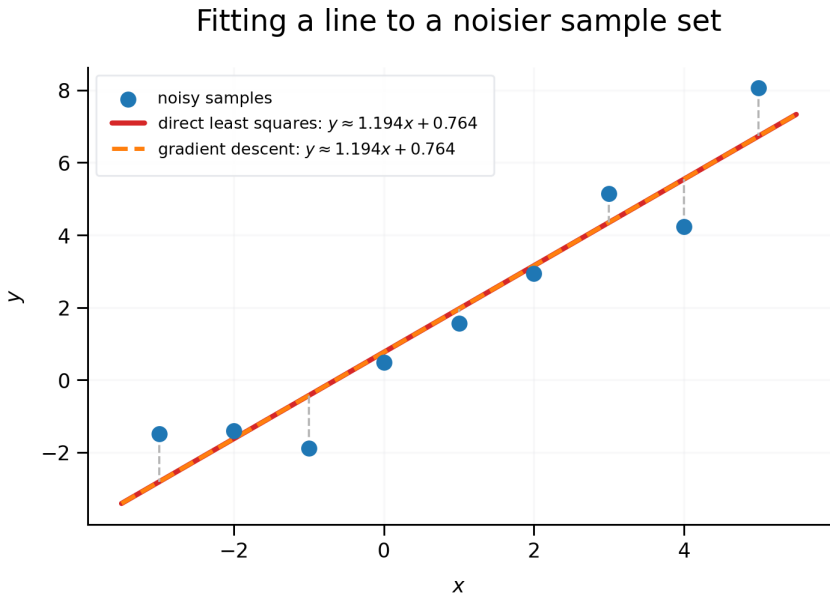


Figure 9.2: An example of line fitting. The data points are fixed, and the least-squares line is chosen to minimize the sum of squared residuals.

This is now a function on the two-dimensional *parameter space* with coordinates (m, n). Its partial derivatives are

$$\frac{\partial J}{\partial m} = -\sum_{i=1}^{N} x_i\big(y_i - (mx_i + n)\big), \tag{9.23}$$

$$\frac{\partial J}{\partial n} = -\sum_{i=1}^{N} \big(y_i - (mx_i + n)\big). \tag{9.24}$$

Starting from an initial guess (m_0, n_0), gradient descent updates the parameters by

$$\begin{bmatrix} m_{k+1} \\ n_{k+1} \end{bmatrix} = \begin{bmatrix} m_k \\ n_k \end{bmatrix} - \eta \begin{bmatrix} \dfrac{\partial J}{\partial m}(m_k, n_k) \\ \dfrac{\partial J}{\partial n}(m_k, n_k) \end{bmatrix}, \tag{9.25}$$

where $\eta > 0$ is the learning rate.

Figure 9.3 shows the loss surface over the (m, n) plane for the same fixed data set used in Figure 9.2. The contour plot makes the geometry especially clear: every point in parameter space corresponds to one candidate line, and gradient descent walks downhill on this terrain until it reaches the same minimizer found by least squares. This is the simplest version of the idea behind training large machine-learning models: instead of

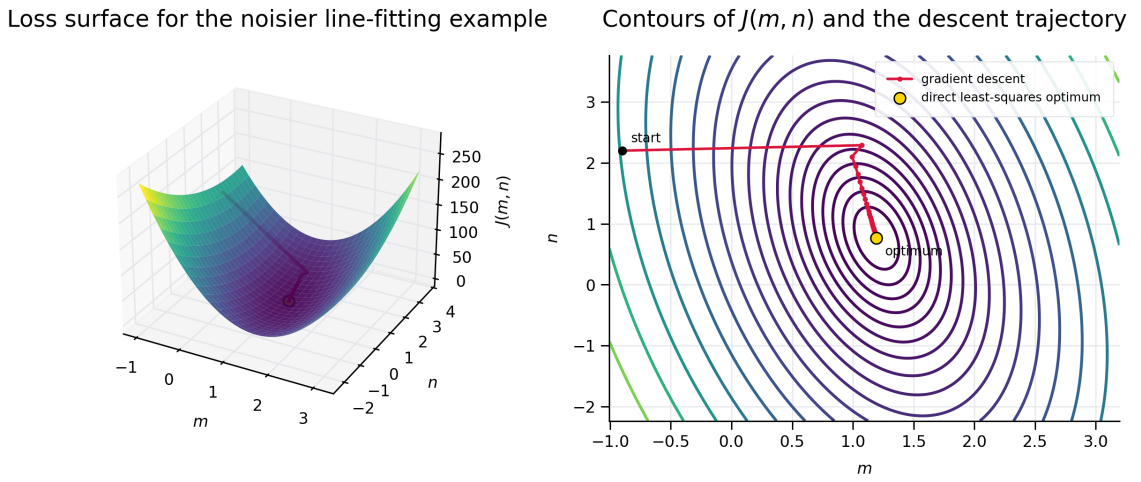


Figure 9.3: The least-squares loss $J(m, n)$ viewed as a surface over parameter space. The left panel shows the terrain itself; the right panel shows contours together with a gradient-descent trajectory that converges to the same optimum as the closed-form least-squares solution.

solving normal equations, we iteratively improve parameters by following the negative gradient.

For this example, gradient descent converges to

$$\hat{m}_{\text{GD}} \approx 1.194, \qquad \hat{n}_{\text{GD}} \approx 0.764,$$

which matches the direct least-squares solution,

$$\hat{m}_{\text{LS}} \approx 1.194, \qquad \hat{n}_{\text{LS}} \approx 0.764,$$

up to numerical precision.

A couple of Remarks to close this section:

- Later in the course we will learn about the backpropagation algorithm to efficiently fit neural network weights. Also know that there are many improved versions of gradient descent (e.g. SGD and Adam) and training large neural networks raises many algorithmic and challenging research problems. See also Appendix D.
- The formulation we gave in this section for line fitting is mostly pedagogical. In practice, the vertical distance (δy) is not an ideal error measure because it is sensitive to the slope of the line.

Appendix A

Global Paths

Apparently, the furthest major city from Austin, TX is Port Louis, Mauritius. Here are the Euclidean and geodesic shortest paths between the two cities (figures generated by GenAI).

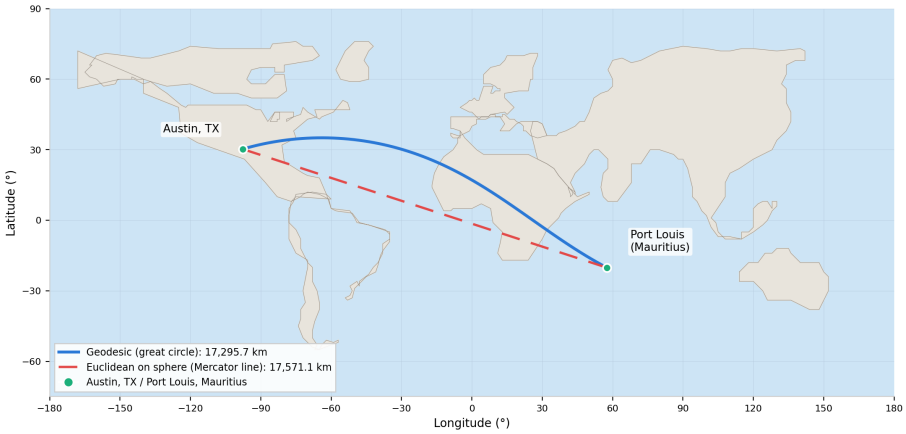


Figure A.1: **Planar (Mercator) map comparison of two paths from Austin, TX to Port Louis, Mauritius.** The **blue solid line** is the geodesic (great-circle) path, which is the true shortest route on Earth's surface (distance: **17,295.7 km**). The **red dashed line** is the Euclidean path — a straight line drawn on the Mercator projection and then interpreted as a sequence of geographic coordinates; when the arc length of this curve is measured on the sphere it totals **17,571.1 km**, which is **275.4 km** (1.59%) longer than the geodesic. The apparent shortness of the red path on the flat map is an artifact of the Mercator projection, which distorts distances, particularly at mid-latitudes.

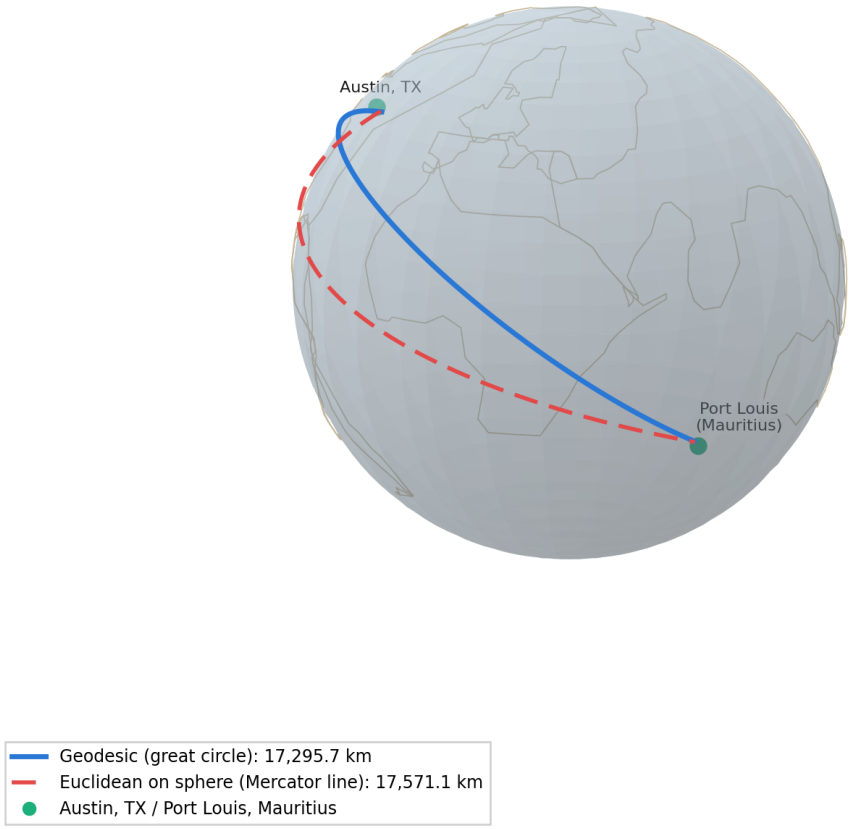


Figure A.2: **Three-dimensional globe comparison of the same two paths.** The **blue solid line** shows the geodesic great-circle path (17,295.7 km), which follows the shortest arc on the sphere. The **red dashed line** shows the Mercator straight line lifted onto the sphere (17,571.1 km): because the Mercator projection is not isometric, this curve deviates from the great circle and dips unnecessarily southward, making it 275.4 km longer. The excess confirms that minimizing Euclidean distance on a planar map does not minimize geodesic distance on the globe.

If you would like to play with gradient descent on terrains, here is a prompt you can use:

```
PROMPT  Local vs. global shortest paths on a terrain

Goal: reproduce and explore the idea that gradient descent on the length of a
path finds a LOCAL minimum (a geodesic), which may not be the GLOBAL shortest
path, while dynamic programming over the whole space finds the global optimum.

Use Python with numpy, scipy, and matplotlib.

---------------------------------------------------------------------
1. The terrain
---------------------------------------------------------------------
The agent moves on a surface z = f(x, y) over the square [-5, 5] x [-5, 5].
Use exactly this terrain so your results match the notes (a sum of Gaussian
bumps and pits plus a gentle ripple):

    import numpy as np

    rng     = np.random.default_rng(7)
    centers = rng.uniform(-4.2, 4.2, size=(9, 2))
    amps    = rng.uniform(-1.5, 1.7, size=9)
    widths  = rng.uniform(0.8, 1.7, size=(9, 2))

    def f(x, y):
        x = np.asarray(x, float); y = np.asarray(y, float)
        z = np.zeros(np.broadcast(x, y).shape)
        for (cx, cy), a, (sx, sy) in zip(centers, amps, widths):
            z += a * np.exp(-(((x - cx)**2) / (2*sx**2)
                              + ((y - cy)**2) / (2*sy**2)))
        z += 0.30 * np.sin(1.1 * x) * np.cos(0.9 * y)
        return z

First, plot f as a 3D surface and as a 2D top-down image. Find the tall peak
(the maximum of f); you will route paths around it.

---------------------------------------------------------------------
2. Length of a path ALONG THE SURFACE
---------------------------------------------------------------------
Represent a path as a sequence of waypoints (x_i, y_i), with the first and
    last
fixed (the start and goal) and the interior ones free. Measure its length on
    the
surface: densely sample each segment, lift the samples to (x, y, f(x,y)), and
sum the 3D distances between consecutive samples,
```

```
    L = sum sqrt( (dx)^2 + (dy)^2 + (dz)^2 ),   dz = f-difference.
This is the cost we minimize. Climbing adds length, so short paths avoid
    hills.

-----------------------------------------------------------------
3. Two methods to compute a path from start to goal
-----------------------------------------------------------------
(a) PATH GRADIENT DESCENT (local). Compute the gradient of L with respect to
    the
    interior waypoints (a numerical finite-difference gradient is fine), then
    repeatedly take a small step downhill, keeping the endpoints fixed:
        p_i  <-  p_i - eta * dL/dp_i.
    Guard the normalization / differences so a zero-length segment cannot
    make a
    NaN. Run until the path stops changing.

(b) COST-TO-GO DESCENT (global). Lay a grid over the square and build a graph
    in
    which each cell connects to its neighbors (use 8 or, better, 16
    neighbors),
    with edge weight = the surface length of that step. Run Dijkstra's
    algorithm
    FROM THE GOAL to get d(x, goal) at every cell, then recover the shortest
    path
    by following predecessors back from the start. (This is steepest descent
    on
    the cost-to-go function.)

-----------------------------------------------------------------
4. The experiment
-----------------------------------------------------------------
Put the start and goal on OPPOSITE sides of the tall peak. Run method (a)
    TWICE
from two different initial paths: one bowed to one side of the peak, one
    bowed to
the other (e.g. the straight line plus a +/- sine bump perpendicular to it).
    Then
run method (b) once.

Plot all the resulting paths on the top-down terrain and report the surface
length of each. You should find that the two runs of (a) converge to two
DIFFERENT paths (two local minima), that one is noticeably longer than the
    other,
and that (b) matches the shorter one  the global optimum.

-----------------------------------------------------------------
```

```
5. Questions to answer
----------------------------------------------------------------------
- Why can't the longer path slide over to become the shorter one? What is the
  barrier, and where does it live  on the terrain, or in the space of paths?
- Each path from method (a) is a geodesic. Explain why, using the fact that
    its
  length-gradient is zero. Is every geodesic the shortest path? (See the
    notes.)
- Method (b) is guaranteed to find the global optimum; method (a) is not. What
  does (b) pay for that guarantee? (Hint: what must it store and explore?)
- Change eta and the number of waypoints. What happens if eta is too large?
  Does the LENGTH of the trapped path change? Does which SIDE it is trapped on
  change?
- Move the start and goal so that BOTH ways around the peak have nearly the
    same
  length. Do the two methods still disagree?
```

Appendix B

Linear and Affine Transformations

So far, we saw that transformations involving rotations can be represented as a matrix multiplication but avoided translations which shift the origin. In other words, $\vec{O}$ did not move. Before looking into translations, let's formalize the definition of a linear transformation. This rests on two ingredients: the *field* of scalars we multiply by, and the *vector space* those scalars act on.

Definition 2 (Field)**.** A *field* $\mathbb{F}$ is a set of scalars equipped with addition and multiplication that behave like ordinary arithmetic: you can add, subtract, multiply, and divide by any nonzero element, and the usual laws hold (both operations are commutative and associative, each has an identity, 0 and $1 \neq 0$, every element has an additive inverse, every nonzero element has a multiplicative inverse, and multiplication distributes over addition).

Throughout these notes the field is $\mathbb{R}$, the real numbers, and our vectors live in $\mathbb{R}^n$. This is by far the most common choice in AI, but several other fields are used in practice: $\mathbb{C}$ (complex numbers) in Fourier analysis, signal processing, and quantum computing; $\mathbb{Q}$ (rationals) in exact symbolic computation; $\mathbb{F}_2 = \{0, 1\}$ (with addition as XOR) in boolean logic, error-correcting codes, and cryptography; $\mathbb{F}_{2^8}$ (a 256-element field) for byte-level arithmetic in AES encryption and Reed–Solomon codes; and $\mathbb{F}_p$ (integers modulo a prime) in elliptic-curve cryptography and hashing. Fields with $0/1$ elements are especially common: the one-hot chessboard encoding of Section 2, for instance, has entries in $\{0, 1\}$ though we viewed it as a vector over $\mathbb{R}$ so that Euclidean distances between boards would carry meaning. As mentioned in that example, there are other metrics like *Hamming distance*, the number of coordinates in which two vectors differ. Hamming distance is a genuine metric, but it is piecewise constant and therefore not very helpful for gradient-based learning. In practice, when a model predicts a *probability* for each coordinate rather than a hard 0 or 1, we replace it with a smooth surrogate such as the binary cross-entropy loss. This is not a metric, it is neither symmetric nor a distance between two

points, but a differentiable measure of how well a predicted probability matches a target, which is exactly what an optimizer needs.

Definition 3 (Vector space)**.** A *vector space* over a field $\mathbb{F}$ (for us, $\mathbb{F} = \mathbb{R}$) is a set V whose elements are called vectors, equipped with two operations. Vector addition takes two vectors $\vec{u}, \vec{v} \in V$ and returns a vector $\vec{u} + \vec{v} \in V$, and scalar multiplication takes a scalar $\alpha \in \mathbb{F}$ and a vector $\vec{v} \in V$ and returns a vector $\alpha\vec{v} \in V$. (This scalar multiplication $\mathbb{F} \times V \to V$ is a new operation, distinct from the field's internal multiplication $\mathbb{F} \times \mathbb{F} \to \mathbb{F}$: it combines a scalar with a vector rather than two scalars.) These operations satisfy the usual rules of arithmetic: addition is commutative and associative, there is a zero vector $\vec{0}$ with $\vec{v} + \vec{0} = \vec{v}$, every vector has an additive inverse, and scalar multiplication distributes over addition.

Definition 4 (Linear transformation)**.** Let V and W be vector spaces over a field $\mathbb{F}$. A map $T\colon V \to W$ is a *linear transformation* if for all $\vec{u}, \vec{v} \in V$ and all scalars $\alpha \in \mathbb{F}$,

$$T(\vec{u} + \vec{v}) = T(\vec{u}) + T(\vec{v}) \qquad \text{(additivity)}, \tag{B.1}$$

$$T(\alpha\vec{v}) = \alpha\, T(\vec{v}) \qquad \text{(homogeneity)}. \tag{B.2}$$

As we derived in the previous section, rotation about the origin is a transformation of the form $\vec{y} = A\vec{x}$ with

$$A = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix},$$

one checks that rotating a sum of vectors equals the sum of the rotated vectors, and rotating a scaled vector equals the scaled rotation; both properties hold, so $\vec{x} \mapsto A\vec{x}$ is linear. More generally, any map of the form $\vec{y} = A\vec{x}$ is a linear transformation.

Now add a translation, $\vec{y} = A\vec{x} + \vec{b}$ with $\vec{b} \neq \vec{0}$ — for instance, rotating and then shifting the result. This map is *no longer linear*. The quickest way to see it is that a linear map must fix the origin: setting $\alpha = 0$ in the homogeneity condition forces $T(\vec{0}) = \vec{0}$, whereas here $\vec{x} = \vec{0}$ gives $\vec{y} = \vec{b} \neq \vec{0}$. Additivity fails for the same reason: writing $f(\vec{x}) = A\vec{x} + \vec{b}$,

$$f(\vec{u} + \vec{v}) = A(\vec{u} + \vec{v}) + \vec{b}, \qquad f(\vec{u}) + f(\vec{v}) = A(\vec{u} + \vec{v}) + 2\vec{b},$$

which differ by $\vec{b}$, so the image of a sum cannot equal the sum of the images.

It is worth being precise about two words that are often used interchangeably. In one dimension, a map is *linear* (in the strict sense) if it is just a scaling,

$$y = wx,$$

a straight line *through the origin*: doubling the input doubles the output, and a zero input always gives a zero output. A map is *affine* if it is a linear map plus a constant offset,

$$y = wx + b,$$

a straight line that may sit anywhere, since the bias b shifts it up or down and lets it cross the axis away from the origin. Every linear map is affine (with $b = 0$), but not every affine map is linear. In machine learning the term "linear layer" is used loosely for the affine form $W\vec{x} + \vec{b}$; we keep the distinction in mind because the bias is exactly what will let us move a unit's kink around later on.

Appendix C

Probability Basics

Probability provides a rigorous foundation for representing uncertainty. This appendix reviews the concepts used throughout the manuscript. Random variables are denoted by uppercase letters, such as X, and their possible realizations by lowercase letters, such as x.

C.1 Sample Spaces, Events, and Probabilities

Definition 5 (Sample space and outcome)**.** A *sample space* Ω is the set of all possible outcomes of a random experiment. An element $\omega \in \Omega$ is called an *outcome*.

Definition 6 (Event)**.** An *event* is a subset $A \subseteq \Omega$. The event A occurs when the observed outcome belongs to A. More precisely, the collection of events is a set $\mathcal{F} \subseteq 2^{\Omega}$ that contains Ω, is closed under complements, and is closed under countable unions. Such a collection is called a *σ-algebra*.

Definition 7 (Probability space)**.** A *probability space* is a triple $(\Omega, \mathcal{F}, P)$, where Ω is a sample space, $\mathcal{F}$ is a σ-algebra of events, and $P : \mathcal{F} \to [0, 1]$ assigns a probability to each event.

Definition 8 (Random variable)**.** A *random variable* is a function $X : \Omega \to \mathbb{R}$. Thus, X assigns a numerical value $X(\omega)$ to each outcome ω.

For a discrete random variable, it is common to write

$$p_X(x) = P(X = x) = P\big(\{\omega \in \Omega : X(\omega) = x\}\big). \tag{C.1}$$

When the random variable is clear from context, this is abbreviated as

$$p(x) = P[X = x]. \tag{C.2}$$

In words, $p(x)$ is the probability that the random variable X takes the value x. For a continuous random variable, however, individual points normally have probability zero. In that case, $p_X(x)$ denotes a probability *density*, not the point probability $P(X = x)$.

C.2 Axioms of Probability

Definition 9 (Probability axioms)**.** A probability measure P satisfies the following axioms:

1. **Nonnegativity:** $P(A) \geq 0$ for every $A \in \mathcal{F}$.
2. **Normalization:** $P(\Omega) = 1$.
3. **Countable additivity:** If $A_1, A_2, \ldots \in \mathcal{F}$ are pairwise disjoint, then

$$P\left(\bigcup_{i=1}^{\infty} A_i\right) = \sum_{i=1}^{\infty} P(A_i). \tag{C.3}$$

Several useful facts follow immediately. In particular,

$$P(\varnothing) = 0, \qquad P(A^c) = 1 - P(A), \tag{C.4}$$

and, for any two events A and B,

$$P(A \cup B) = P(A) + P(B) - P(A \cap B). \tag{C.5}$$

C.3 Example: Rolling a Die

Consider a single roll of a fair six-sided die. The sample space is

$$\Omega = \{1, 2, 3, 4, 5, 6\}, \tag{C.6}$$

and each elementary outcome has probability $1/6$. Let

$$A = \{1, 3, 5\} \tag{C.7}$$

be the event that the roll comes up odd. Then

$$P(A) = P(1) + P(3) + P(5) = \frac{3}{6} = \frac{1}{2}. \tag{C.8}$$

Similarly, if $B = \{4, 5, 6\}$ is the event that the roll is greater than three, then

$$A \cap B = \{5\}, \qquad P(A \cap B) = \frac{1}{6}, \tag{C.9}$$

and

$$P(A \cup B) = \frac{1}{2} + \frac{1}{2} - \frac{1}{6} = \frac{5}{6}. \tag{C.10}$$

C.4 Distribution Functions, Mass Functions, and Densities

Definition 10 (Cumulative distribution function)**.** The *cumulative distribution function* (CDF) of a real-valued random variable X is

$$F_X(x) = P(X \le x), \qquad x \in \mathbb{R}. \tag{C.11}$$

The CDF is nondecreasing and right-continuous, with $\lim_{x\to-\infty} F_X(x) = 0$ and $\lim_{x\to\infty} F_X(x) = 1$.

Definition 11 (Probability mass function)**.** If X takes values in a finite or countable set $\mathcal{X}$, its *probability mass function* (PMF) is

$$p_X(x) = P(X = x), \qquad x \in \mathcal{X}. \tag{C.12}$$

It satisfies

$$p_X(x) \ge 0, \qquad \sum_{x\in\mathcal{X}} p_X(x) = 1. \tag{C.13}$$

The corresponding CDF is the step function

$$F_X(x) = \sum_{t\in\mathcal{X}:\, t\le x} p_X(t). \tag{C.14}$$

For any set $A \subseteq \mathcal{X}$,

$$P(X \in A) = \sum_{x\in A} p_X(x). \tag{C.15}$$

Definition 12 (Probability density function)**.** A random variable X is continuous if there is a nonnegative function $p_X : \mathbb{R} \to [0, \infty)$, called its *probability density function* (PDF), such that

$$P(a < X \le b) = \int_a^b p_X(x)\, dx. \tag{C.16}$$

The density satisfies

$$p_X(x) \ge 0, \qquad \int_{-\infty}^{\infty} p_X(x)\, dx = 1. \tag{C.17}$$

The CDF and PDF are related by

$$F_X(x) = \int_{-\infty}^{x} p_X(t)\, dt. \tag{C.18}$$

At points where F_X is differentiable,

$$p_X(x) = \frac{d}{dx} F_X(x). \tag{C.19}$$

For a continuous random variable, $P(X = x) = 0$ for every individual value x; probability is obtained by integrating the density over a set or interval.

C.5 Expectation, Variance, and Standard Deviation

Definition 13 (Expectation)**.** The *expectation*, or mean, of a random variable X is

$$\mathbb{E}[X] = \sum_{x \in \mathcal{X}} x \, p_X(x) \tag{C.20}$$

for a discrete random variable, and

$$\mathbb{E}[X] = \int_{-\infty}^{\infty} x \, p_X(x) \, dx \tag{C.21}$$

for a continuous random variable, provided the relevant sum or integral is well defined.

More generally, for a function g,

$$\mathbb{E}[g(X)] = \sum_{x \in \mathcal{X}} g(x) p_X(x) \tag{C.22}$$

in the discrete case, and

$$\mathbb{E}[g(X)] = \int_{-\infty}^{\infty} g(x) p_X(x) \, dx \tag{C.23}$$

in the continuous case.

Definition 14 (Variance and standard deviation)**.** Let $\mu = \mathbb{E}[X]$. The *variance* of X is

$$\mathrm{Var}(X) = \mathbb{E}\left[(X - \mu)^2\right] = \mathbb{E}[X^2] - \left(\mathbb{E}[X]\right)^2. \tag{C.24}$$

The *standard deviation* is

$$\mathrm{Std}(X) = \sqrt{\mathrm{Var}(X)}. \tag{C.25}$$

The standard deviation has the same physical units as X, whereas the variance has squared units.

C.6 The Gaussian Distribution

Definition 15 (Univariate Gaussian distribution)**.** A real-valued random variable X has a *Gaussian*, or *normal*, distribution with mean $\mu \in \mathbb{R}$ and variance $\sigma^2 > 0$, written

$$X \sim \mathcal{N}(\mu, \sigma^2), \tag{C.26}$$

if its density is

$$p_X(x) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{(x-\mu)^2}{2\sigma^2}\right). \tag{C.27}$$

For this distribution,

$$\mathbb{E}[X] = \mu, \qquad \mathrm{Var}(X) = \sigma^2, \qquad \mathrm{Std}(X) = \sigma. \tag{C.28}$$

For example, if a scalar measurement is modeled as $X \sim \mathcal{N}(10, 4)$, then its expected value is 10, its variance is 4, and its standard deviation is 2.

C.7 Joint Distributions and Random Vectors

A collection of random variables can be combined into a random vector $\mathbf{X} = (X_1, \dots, X_d)^\top$.

Definition 16 (Joint distribution)**.** The *joint cumulative distribution function* of $\mathbf{X} = (X_1, \dots, X_d)^\top$ is

$$F_{\mathbf{X}}(x_1, \dots, x_d) = P(X_1 \le x_1, \dots, X_d \le x_d). \tag{C.29}$$

In the discrete case, the joint PMF is

$$p_{\mathbf{X}}(x_1, \dots, x_d) = P(X_1 = x_1, \dots, X_d = x_d). \tag{C.30}$$

In the continuous case, a joint PDF satisfies

$$P(\mathbf{X} \in A) = \int_A p_{\mathbf{X}}(\mathbf{x})\, d\mathbf{x} \tag{C.31}$$

for suitable sets $A \subseteq \mathbb{R}^d$.

A distribution for one component can be obtained from the joint distribution by *marginalization*. For example, in the discrete two-variable case,

$$p_X(x) = \sum_{y} p_{X,Y}(x, y), \tag{C.32}$$

and in the continuous case,

$$p_X(x) = \int_{-\infty}^{\infty} p_{X,Y}(x, y)\, dy. \tag{C.33}$$

Definition 17 (Multivariate Gaussian distribution)**.** A random vector $\mathbf{X} \in \mathbb{R}^d$ has a multivariate Gaussian distribution with mean vector $\boldsymbol{\mu} \in \mathbb{R}^d$ and symmetric positive-definite covariance matrix $\boldsymbol{\Sigma} \in \mathbb{R}^{d \times d}$, written

$$\mathbf{X} \sim \mathcal{N}(\boldsymbol{\mu}, \boldsymbol{\Sigma}), \tag{C.34}$$

if its density is

$$p_{\mathbf{X}}(\mathbf{x}) = \frac{1}{(2\pi)^{d/2} |\boldsymbol{\Sigma}|^{1/2}} \exp\left(-\frac{1}{2} (\mathbf{x} - \boldsymbol{\mu})^\top \boldsymbol{\Sigma}^{-1} (\mathbf{x} - \boldsymbol{\mu}) \right). \tag{C.35}$$

Its expectation and covariance are

$$\mathbb{E}[\mathbf{X}] = \boldsymbol{\mu}, \qquad \operatorname{Cov}(\mathbf{X}) = \mathbb{E}\left[(\mathbf{X} - \boldsymbol{\mu})(\mathbf{X} - \boldsymbol{\mu})^\top \right] = \boldsymbol{\Sigma}. \tag{C.36}$$

For instance,

$$\begin{bmatrix} X_1 \\ X_2 \end{bmatrix} \sim \mathcal{N}\left(\begin{bmatrix} 0 \\ 1 \end{bmatrix}, \begin{bmatrix} 1 & 0.5 \\ 0.5 & 2 \end{bmatrix} \right) \tag{C.37}$$

has means 0 and 1, variances 1 and 2, and covariance 0.5.

C.8 Independence

Definition 18 (Independence of events)**.** Two events A and B are *independent* if

$$P(A \cap B) = P(A)P(B). \tag{C.38}$$

More generally, events $A_1, \ldots, A_n$ are mutually independent if, for every nonempty index set $I \subseteq \{1, \ldots, n\}$,

$$P\left(\bigcap_{i \in I} A_i\right) = \prod_{i \in I} P(A_i). \tag{C.39}$$

Definition 19 (Independence of random variables)**.** Discrete random variables X and Y are independent if

$$p_{X,Y}(x, y) = p_X(x)p_Y(y) \tag{C.40}$$

for all x and y. Continuous random variables are independent if their joint density factors as

$$p_{X,Y}(x, y) = p_X(x)p_Y(y) \tag{C.41}$$

for all (x, y), except possibly on a set of probability zero.

Independence implies zero covariance when the relevant expectations exist, but zero covariance does not in general imply independence. An important exception is the jointly Gaussian case, for which zero covariance does imply independence.

C.9 Conditional Probability and Conditional Independence

Definition 20 (Conditional probability)**.** For events A and B with $P(B) > 0$, the *conditional probability* of A given B is

$$P(A \mid B) = \frac{P(A \cap B)}{P(B)}. \tag{C.42}$$

Equivalently, the product rule is

$$P(A \cap B) = P(A \mid B)P(B) = P(B \mid A)P(A). \tag{C.43}$$

For discrete random variables,

$$p_{X|Y}(x \mid y) = \frac{p_{X,Y}(x, y)}{p_Y(y)}, \qquad p_Y(y) > 0. \tag{C.44}$$

For continuous random variables, the conditional density is

$$p_{X|Y}(x \mid y) = \frac{p_{X,Y}(x, y)}{p_Y(y)}, \qquad p_Y(y) > 0, \tag{C.45}$$

where the quantities on the right are densities rather than point probabilities.

Definition 21 (Conditional independence)**.** Random variables X and Y are *conditionally independent given* Z, written

$$X \perp\!\!\!\perp Y \mid Z, \tag{C.46}$$

if their conditional joint distribution factors as

$$p_{X,Y|Z}(x, y \mid z) = p_{X|Z}(x \mid z) p_{Y|Z}(y \mid z) \tag{C.47}$$

for every relevant value z for which the conditional distributions are defined.

Conditional independence does not in general imply unconditional independence, and unconditional independence does not in general imply conditional independence.

C.10 Bayes' Theorem

Theorem 1 (Bayes' theorem)**.** *For events A and B with $P(A) > 0$ and $P(B) > 0$,*

$$P(A \mid B) = \frac{P(B \mid A) P(A)}{P(B)}. \tag{C.48}$$

If $A_1, \ldots, A_n$ form a partition of Ω, with $P(A_i) > 0$, then the law of total probability gives

$$P(B) = \sum_{i=1}^{n} P(B \mid A_i) P(A_i), \tag{C.49}$$

and therefore

$$P(A_j \mid B) = \frac{P(B \mid A_j) P(A_j)}{\sum_{i=1}^{n} P(B \mid A_i) P(A_i)}. \tag{C.50}$$

For random variables, Bayes' theorem is commonly written as

$$p_{X|Y}(x \mid y) = \frac{p_{Y|X}(y \mid x) p_X(x)}{p_Y(y)}. \tag{C.51}$$

In statistical terminology, $p_X(x)$ is the *prior*, $p_{Y|X}(y \mid x)$ is the *likelihood*, $p_{X|Y}(x \mid y)$ is the *posterior*, and $p_Y(y)$ is the *evidence* or *marginal likelihood*. The denominator can be obtained by marginalization:

$$p_Y(y) = \sum_{x} p_{Y|X}(y \mid x) p_X(x) \tag{C.52}$$

when X is discrete, or

$$p_Y(y) = \int p_{Y|X}(y \mid x) p_X(x)\, dx \tag{C.53}$$

when X is continuous.

Appendix D

From Line-fitting to Neural Networks: Backpropagation

D.1 Motivation

Consider a differentiable function $f(x)$. Suppose we want to find a minimum of f with respect to x. One approach is to compute $f'(x)$, set it to zero, and solve for x. Alternatively, we can use an iterative method such as gradient descent. Starting from an initial guess $x^{(0)}$, gradient descent repeatedly applies an update of the form

$$x^{(k+1)} = x^{(k)} - \eta f'\big(x^{(k)}\big),$$

where $\eta > 0$ is the learning rate.

Model fitting is different that finding the minimum of a given function. In model fitting, we are given a family of functions together with potentially noisy observations. In the simplest case, the observations are input-target pairs (x, t), where t is the desired output associated with input x. A parameterized model f_θ predicts

$$y = f_\theta(x),$$

and our goal is to choose the parameter values so that the predictions are close to the observed targets.

In a neural network, there may be billions of model parameters, so computing the required gradients efficiently is critical. Backpropagation provides this efficiency. At its core, backpropagation is an efficient bookkeeping procedure for applying the chain rule to a computation graph. An optimization method, such as gradient descent, then uses the resulting gradients to update the model parameters.

Let us look at a simple example to illustrate parameter reuse and gradient accumulation. Consider the model

$$y = ax + a^2,$$

where the model parameters $\theta = [a]$ consist of only a. The same parameter a occurs in two terms, and therefore affects the output through two computational branches. This is the feature we want to study: in a real network, a single weight influences the loss through many paths, and we need a systematic way to combine those influences.

For a given observation (x, t), use the least-squares loss

$$L(y, t) = \frac{1}{2}(y - t)^2.$$

Thus, as a function of a,

$$L(a) = \frac{1}{2}\big(ax + a^2 - t\big)^2.$$

Our objective is to choose a to minimize this loss.

This example is small enough that we could differentiate $L(a)$ directly with one application of the chain rule. We will do exactly that at the end, as a check. The point of the machinery developed below is that it keeps working when the model is far too large to differentiate by hand.

D.2 Computation Graph

Modern automatic-differentiation systems do not manipulate formulas as strings. They track the identity of each tensor or variable and record the operations applied to it. The record is a directed acyclic graph whose nodes are values and whose edges point from an operation's inputs to its output.

For our example, the graph has two *leaf* nodes, which are supplied from outside the computation,

$$a \quad \text{(the parameter)}, \qquad x \quad \text{(the input)},$$

and three *computed* nodes,

$$p = ax, \qquad v = a^2, \qquad y = p + v,$$

followed by the loss

$$L = \frac{1}{2}(y - t)^2.$$

The target t is a fixed data value rather than something we differentiate with respect to, so we do not draw it as a separate node.

Take a look at the computation graph in Figure D.1. We will focus on a which is a single node with two outgoing edges. One edge feeds the multiplication that produces p, and the other feeds the squaring that produces v. The graph records that both occurrences of a in the expression $ax + a^2$ refer to the same parameter, rather than to two unrelated parameters that happen to hold the same value. Everything interesting in the backward pass follows from this one branching.

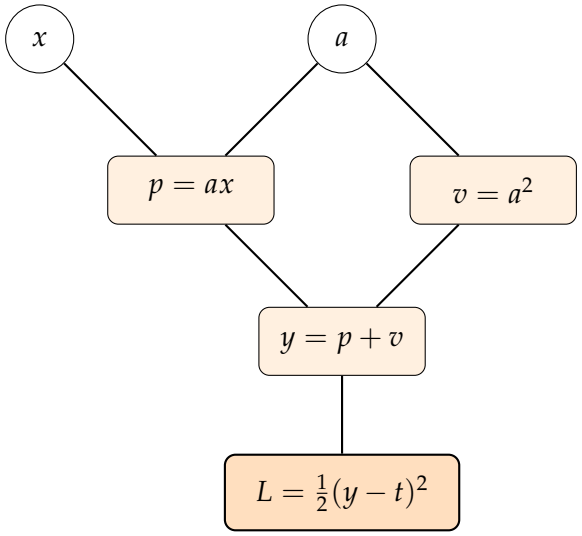


Figure D.1: Computation graph for $y = ax + a^2$ and $L = \frac{1}{2}(y-t)^2$. The parameter a feeds two computational branches. During backpropagation, the gradient contributions from the branches are accumulated at the shared parameter node.

D.3 The Forward Pass

The forward pass evaluates each node in an order consistent with the graph, so that a node is computed only after all of its inputs are available. Symbolically,

$$\begin{aligned} p &= ax, \\ v &= a^2, \\ y &= p + v = ax + a^2, \\ L &= \frac{1}{2}(y-t)^2 = \frac{1}{2}\big(ax + a^2 - t\big)^2. \end{aligned}$$

The intermediate values are retained, because the backward pass will reuse them. For example, the derivative of $v = a^2$ is $2a$, which requires the forward-pass value of a; and the derivative of the loss is $y - t$, which requires the forward-pass values of y and t. This is why training a large network consumes so much memory: the forward pass must store activations that will not be needed until the backward pass reaches them.

D.4 The Backward Pass

Backpropagation starts at the loss and traverses the computation graph in reverse. For each operation, it multiplies the derivative arriving from downstream by that operation's

local derivative.

D.4.1 Notation and the rule we are applying

Throughout, $\dfrac{\partial L}{\partial n}$ denotes the sensitivity of the scalar loss L to the value held at node n, with all other *leaf* nodes held fixed. Because L is a scalar and every node is a value in the graph, this quantity is well defined for every node.

One point deserves emphasis, because it is where the chain rule for graphs differs from the chain rule for a single chain of functions. If a node n has several outgoing edges, then changing n changes the loss through each of them, and the effects add:

$$\frac{\partial L}{\partial n} = \sum_{m \in \text{children}(n)} \frac{\partial L}{\partial m} \frac{\partial m}{\partial n}.$$

Here $\text{children}(n)$ is the set of nodes computed directly from n, and each local factor $\partial m / \partial n$ is an ordinary partial derivative that treats the *other* inputs of the operation producing m as fixed.

This single formula is the whole algorithm. Multiplication along an edge and summation over outgoing edges is precisely the multivariable chain rule; backpropagation is the observation that if we visit nodes in reverse topological order, every $\partial L / \partial m$ on the right-hand side has already been computed by the time we need it.

In our graph, y, p, v and x each have a single outgoing edge, so the sum has one term for them. The node a has two, so its sum has two terms. That is the case we care about.

D.4.2 Loss node

From

$$L = \frac{1}{2}(y - t)^2,$$

the only input we differentiate with respect to is y, since t is a fixed data value. Hence

$$\frac{\partial L}{\partial y} = y - t.$$

D.4.3 Addition node

Since

$$y = p + v,$$

the local derivatives are

$$\frac{\partial y}{\partial p} = 1, \qquad \frac{\partial y}{\partial v} = 1.$$

Applying the chain rule,

$$\frac{\partial L}{\partial p} = \frac{\partial L}{\partial y}\frac{\partial y}{\partial p} = (y - t), \qquad \frac{\partial L}{\partial v} = \frac{\partial L}{\partial y}\frac{\partial y}{\partial v} = (y - t).$$

So an addition node passes the incoming derivative unchanged to each of its inputs. This is the mechanism by which one incoming derivative becomes two outgoing ones, and it is what eventually gives a two contributions to collect.

D.4.4 Multiplication branch

The first branch computes

$$p = ax.$$

Its local derivatives are

$$\frac{\partial p}{\partial a} = x, \qquad \frac{\partial p}{\partial x} = a.$$

Note that each is taken with the other input held fixed, exactly as the rule above requires. Consequently, the contribution of this branch to the derivative with respect to a is

$$\left.\frac{\partial L}{\partial a}\right|_{p\text{-branch}} = \frac{\partial L}{\partial p}\frac{\partial p}{\partial a} = (y - t)\, x.$$

The same branch also propagates a derivative to the input node:

$$\frac{\partial L}{\partial x} = \frac{\partial L}{\partial p}\frac{\partial p}{\partial x} = (y - t)\, a.$$

We do not need $\partial L/\partial x$ to fit this model, since x is data rather than a parameter. It is worth computing anyway: in a multi-layer network, the input to one layer is the output of the previous layer, and this is precisely the quantity that gets handed backward from one layer to the next. We could also use it for gradient descent once the model parameters are fixed.

D.4.5 Square branch

The second branch computes

$$v = a^2.$$

Its local derivative is

$$\frac{\partial v}{\partial a} = 2a.$$

Therefore, this branch contributes

$$\left.\frac{\partial L}{\partial a}\right|_{v\text{-branch}} = \frac{\partial L}{\partial v}\frac{\partial v}{\partial a} = (y - t)\, 2a.$$

D.4.6 Accumulating the two branches

The node a is an input to both $p = ax$ and $v = a^2$, so it has two children and the sum in our rule has two terms:

$$\begin{aligned}\frac{\partial L}{\partial a} &= \left.\frac{\partial L}{\partial a}\right|_{p\text{-branch}} + \left.\frac{\partial L}{\partial a}\right|_{v\text{-branch}} \\ &= (y-t)x + (y-t)2a \\ &= (y-t)(x+2a).\end{aligned}$$

That is the gradient we were after:

$$\boxed{\frac{\partial L}{\partial a} = (y-t)(x+2a)}$$

and, after substituting $y = ax + a^2$,

$$\boxed{\frac{\partial L}{\partial a} = \big(ax + a^2 - t\big)(x+2a).}$$

Equivalently, we can write out the two complete chain-rule paths from L back to a:

$$\begin{aligned}\frac{\partial L}{\partial a} &= \underbrace{\frac{\partial L}{\partial y}\frac{\partial y}{\partial p}\frac{\partial p}{\partial a}}_{\text{path through } ax} + \underbrace{\frac{\partial L}{\partial y}\frac{\partial y}{\partial v}\frac{\partial v}{\partial a}}_{\text{path through } a^2} \\ &= \underbrace{(y-t)(1)(x)}_{\text{multiplication branch}} + \underbrace{(y-t)(1)(2a)}_{\text{square branch}}.\end{aligned}$$

Thus, derivatives are *multiplied* along each computational path, while contributions from multiple paths are *added*.

Enumerating paths this way is fine for a graph with two of them, but the number of paths can grow exponentially with depth, so it is not how backpropagation actually proceeds. The algorithm computes and stores one number per *node*, namely $\partial L/\partial n$, and reuses it for every edge leaving that node. This is the source of its efficiency: the cost of the backward pass is proportional to the number of edges in the graph, not the number of paths through it, and therefore only a small constant factor more than the forward pass. This is similar to how dynamic programming avoids enumerating all paths by storing optimal distances (to level sets). In this case, the distances are given by the inverse topological order – backwards from the final output.

D.5 Numerical Example

Let

$$a = 2, \qquad x = 3, \qquad t = 12.$$

D.5.1 Forward pass

Evaluating the nodes in order,

$$\begin{aligned} p &= ax = 2 \cdot 3 = 6, \\ v &= a^2 = 2^2 = 4, \\ y &= p + v = 6 + 4 = 10, \\ L &= \frac{1}{2}(y - t)^2 = \frac{1}{2}(10 - 12)^2 = 2. \end{aligned}$$

D.5.2 Backward pass

Starting from the loss,

$$\frac{\partial L}{\partial y} = y - t = 10 - 12 = -2.$$

The addition node passes this to both of its inputs,

$$\frac{\partial L}{\partial p} = -2, \qquad \frac{\partial L}{\partial v} = -2.$$

The multiplication branch contributes

$$\left.\frac{\partial L}{\partial a}\right|_{p\text{-branch}} = \frac{\partial L}{\partial p}\frac{\partial p}{\partial a} = (-2)(x) = (-2)(3) = -6,$$

and the square branch contributes

$$\left.\frac{\partial L}{\partial a}\right|_{v\text{-branch}} = \frac{\partial L}{\partial v}\frac{\partial v}{\partial a} = (-2)(2a) = (-2)(4) = -8.$$

Accumulating at the node a gives

$$\boxed{\frac{\partial L}{\partial a} = -6 + (-8) = -14.}$$

The multiplication branch also sends a derivative to the input node:

$$\boxed{\frac{\partial L}{\partial x} = \frac{\partial L}{\partial p}\frac{\partial p}{\partial x} = (-2)(2) = -4.}$$

D.5.3 Checking the answer directly

Because this model is small, we can verify the result without any graph machinery. Differentiating

$$L(a) = \frac{1}{2}\big(ax + a^2 - t\big)^2$$

with a single application of the chain rule gives

$$L'(a) = \big(ax + a^2 - t\big)(x + 2a) = (6 + 4 - 12)(3 + 2 \cdot 2) = (-2)(7) = -14,$$

which agrees with the value accumulated over the two branches. Note where the factor $(x + 2a)$ came from in the direct calculation: it is $\frac{d}{da}(ax + a^2)$, and the two terms x and $2a$ inside it are exactly the two branch contributions that backpropagation added together.

D.5.4 Taking a gradient step

If gradient descent uses learning rate $\eta > 0$, the parameter update is

$$a_{\text{new}} = a - \eta \frac{\partial L}{\partial a}.$$

For example, with $\eta = 0.01$,

$$a_{\text{new}} = 2 - 0.01(-14) = 2.14.$$

The gradient is negative, which means increasing a locally decreases the loss for this particular observation. Indeed, recomputing the forward pass at $a = 2.14$ gives $y \approx 11.00$ and $L \approx 0.50$, down from $L = 2$.