AI for Mathematics

An Undergraduate Introduction to Optimization, Neural Networks, and Equation Discovery, with Applications to Scientific Modeling

Weinan Wang
Department of Mathematics, University of Oklahoma
ww@ou.edu

May 1, 2026

Contents

Preface

These notes accompany a one-semester undergraduate course on the use of modern computational and machine-learning tools in mathematical modeling. The course is built around a single question that recurs in nearly every quantitative science: *given limited and imperfect measurements of a system, how do we recover the rule that governs it?* This is the *inverse problem*, and it is the thread that ties together every technique we will study, from classical least squares to neural networks to automatic equation discovery.

Who this is for. The notes assume single- and multivariable calculus and a first course in linear algebra. A little exposure to probability (the normal distribution, mean and variance) is helpful but not required; we develop what we need. No prior background in machine learning or numerical analysis is assumed. The material is suitable both for sophomores meeting these ideas for the first time and for upper-level students who have seen numerical methods or probability and want to understand where machine learning fits.

How to read these notes. Sections and results marked with a star ($\star$) are more demanding and may be skipped on a first reading without loss of continuity; they are included for students who want the underlying justification. Each chapter ends with exercises. Some are analytical, meant to be done with pencil and paper; others ask you to reason about computational experiments. Solution sketches are provided in the appendix.

A note on the running example. Throughout the notes we return to a single concrete problem: modeling the growth of a tumor from a handful of noisy volume measurements. This problem is simple enough to state in one line, yet rich enough to exercise every idea in the course — optimization, neural networks, automatic differentiation, identifiability, and equation discovery. The later chapters draw on a genuine undergraduate research project, including the false starts and surprises that arose along the way, because those are often more instructive than a polished final result.

On notation. We write scalars in italic (a, K, t), vectors in boldface only when the distinction matters, and reserve θ for the collection of unknown parameters of whatever model is under discussion. The symbol V almost always denotes the tumor volume of the running example. Norms are Euclidean unless stated otherwise. Numerical values quoted in the applied chapters come from the synthetic benchmark described in Chapter 2 and are reproducible from a fixed random seed.

Part I

Foundations

Chapter 1

What "AI for Mathematics" Means

1.1 Two kinds of artificial intelligence

The phrase *artificial intelligence* has, in recent years, come to be associated almost entirely with one kind of system: large language models that converse, summarize, and write code. These are remarkable tools, and we will discuss them in Chapter 14. But they are not what most of this course is about, and it is worth being clear at the outset about the distinction.

There is a second, older, and in many ways deeper use of computation in the sciences. Here the goal is not to imitate human language but to *discover or estimate the mathematical laws that govern a physical, biological, or economic system.* When an astronomer fits an orbit to a sequence of telescope observations, when an epidemiologist estimates the transmission rate of a disease from case counts, or when an engineer calibrates a model of a bridge from sensor data, they are all doing the same kind of thing: using data to pin down a mathematical description of reality. This activity long predates modern computers — Gauss was doing it by hand in 1801 to recover the orbit of the asteroid Ceres — but computation has transformed both its scale and its ambition.

This course is about that second kind of artificial intelligence, which goes by various names: *scientific machine learning*, *data-driven modeling*, or *computational inverse problems*. Its central preoccupation is the relationship between three objects:

- a *model*, which is a family of candidate descriptions of the system, usually with adjustable parameters;
- *data*, which are measurements of the system, always finite in number and corrupted by noise;
- a *procedure* for using the data to choose, within the model family, the description that best matches reality.

Almost everything we study — least squares, gradient descent, neural networks, physics-informed learning, symbolic regression — is some instance of this triple. The techniques differ in how flexible the model is, how the fitting procedure works, and how much prior knowledge we build in. But the underlying question is always the same.

It is worth dwelling for a moment on why the word "intelligence" is even appropriate for the second activity, since no conversation is involved. What these methods automate is a form of *inductive reasoning*: the inference of a general rule from particular instances. A scientist who looks at a scatter of points and conjectures "these lie on a line" is performing exactly the inference that linear regression performs mechanically. A scientist who suspects that a growth process slows as it nears a limit, and writes down a differential equation to capture that intuition, is doing what an

equation-discovery algorithm attempts to do automatically. The intelligence in "AI for mathematics" is the intelligence of disciplined induction — and, as we will see repeatedly, the discipline matters at least as much as the induction, because undisciplined inference from limited data is how one arrives confidently at false conclusions.

1.2 A very short history

The lineage of this subject is worth knowing, both because it is genuinely old and because the recurring lessons were learned long before the current enthusiasm for machine learning.

The decisive early episode is Gauss's recovery of Ceres. In 1801 the astronomer Giuseppe Piazzi observed a new object in the sky for forty-one nights before it was lost in the glare of the Sun. The question was where it would reappear months later — an inverse problem, since the orbital parameters had to be inferred from a short, noisy arc of observations. Gauss, then twenty-four, developed the method of least squares to fit an orbit to Piazzi's data and predicted where to look. The object was found almost exactly where he said it would be. The episode established a template that has not changed: gather imperfect observations, posit a parametric model (here Kepler's laws), and choose the parameters that best reconcile model with data. Every method in this book is a descendant of that idea.

The twentieth century added two crucial ingredients. The first was a rigorous understanding of *noise*: the recognition, formalized in statistics, that measurements are random and that estimation must account for their distribution. This is why least squares is not merely a convenient recipe but, under the right assumptions, the statistically optimal one (Chapter 5). The second was the *digital computer*, which made it feasible to fit models too complicated to handle by hand — and, eventually, models with millions of parameters. Neural networks, conceived in mid-century as crude models of biological neurons, became practical only when computation and data caught up with them, decades later.

The present moment adds a third ingredient, and it is the one that gives this course its urgency: *flexible, data-hungry models are now easy to fit, and the temptation to over-trust them is correspondingly strong.* A modern practitioner can, in an afternoon, fit a neural network or run an automatic equation-discovery tool on a dataset. What has *not* become easier is knowing whether the result means anything. The mathematical pitfalls that Gauss already understood — that data may not determine the parameters, that a good fit is not a correct model — are exactly as dangerous now as they were in 1801, and a great deal more likely to be ignored. Much of this course is an attempt to pair modern tools with old wisdom.

1.3 The forward problem and the inverse problem

It helps to separate two directions of reasoning.

The *forward problem* asks: *given the rule and its parameters, what behavior do we observe?* This is the direction of ordinary applied mathematics. If we know that a population grows according to a particular differential equation, with particular growth rate and carrying capacity, then we can solve the equation and predict the population at any future time. The forward problem is, at least in principle, well-posed: one set of parameters produces one trajectory.

The *inverse problem* runs in the opposite direction: *given observed behavior, what rule and parameters produced it?* This is the direction of data analysis, and it is fundamentally harder. The difficulty is not merely computational. Inverse problems are often *ill-posed*: the data may be consistent with many different parameter values, or even with many different rules, especially

when the measurements are few and noisy. A central theme of this course — developed carefully in Chapter 11 — is that recovering parameters from data is delicate in ways that are easy to overlook, and that a procedure which appears to fit the data well can nonetheless recover badly wrong answers.

Example 1.1 (Forward versus inverse, informally)**.** Suppose a quantity $V(t)$ grows over time and we believe it follows the rule

$$\frac{dV}{dt} = a\,V\left(1 - \frac{V}{K}\right),$$

the *logistic equation*, where a controls how fast it grows when small and K is the ceiling it approaches. The forward problem: given $a = 0.4$, $K = 1000$, and a starting value $V(0) = 10$, compute $V(t)$ for t up to 40. This has a clean answer. The inverse problem: given five noisy measurements of V at scattered times, estimate a and K. This, we will see, can be surprisingly treacherous — two quite different pairs (a, K) may fit the five points almost equally well while disagreeing sharply about the future.

The asymmetry between the two directions is worth stating as a principle. The forward map, from parameters to observations, is typically smooth and single-valued: nudging a parameter nudges the prediction. The inverse map, from observations to parameters, can be *many-to-one* (different parameters, same observations) and *discontinuous* (a tiny change in the data forces a large change in the inferred parameters). When either pathology is present the inverse problem is ill-posed, and no algorithm, however clever, can manufacture information the data do not contain. This is not a defect of particular methods but a property of the problem, and recognizing it is the beginning of wisdom in this subject.

Example 1.2 (A linear inverse problem that is already hard)**.** Even linear problems can be ill-posed. Suppose we wish to recover two numbers x_1, x_2 from two measurements

$$y_1 = x_1 + x_2, \qquad y_2 = x_1 + 1.001\,x_2.$$

The forward map is the matrix $\left(\begin{smallmatrix} 1 & 1 \\ 1 & 1.001 \end{smallmatrix}\right)$, perfectly invertible on paper. But the two equations are nearly identical, so the data barely distinguish x_1 from x_2: a measurement error of 0.001 in y shifts the recovered (x_1, x_2) by order 1. The combination $x_1 + x_2$ is well determined; the combination $x_1 - x_2$ is almost invisible to the data. We will meet exactly this phenomenon, in a nonlinear guise, as the *identifiability ridge* of Chapter 11.

1.4 Why this is hard: a preview of the central difficulties

Before listing the tools we will build, it is worth naming the obstacles they are meant to overcome, because the obstacles motivate the tools.

Noise. Real measurements are never exact. If we treat noisy data as though it were exact — demanding that our model pass through every data point — we end up modeling the noise rather than the signal. This is *overfitting*, and learning to avoid it is one of the organizing concerns of the field.

Sparsity. Data is often scarce. A laboratory might measure a tumor's size only a handful of times over a few weeks. With few measurements, many models become indistinguishable, and the inverse problem becomes correspondingly ambiguous.

Ill-posedness and identifiability. Even with a fixed model, the data may fail to determine the parameters uniquely. Different parameter combinations can produce nearly identical predictions over the range where we have data, while diverging outside it. We will make this precise under the heading of *identifiability* in Chapter 11.

Model misspecification. Most insidiously, the model family we choose may simply be wrong — the true rule may not lie within it. A wrong model can still fit limited data well, returning parameter estimates that are precise, confidently reported, and meaningless. Recognizing and guarding against misspecification is a theme we develop in the applied chapters, and it turns out to have real consequences: a growth law fitted with the wrong functional form can predict a tumor's future size incorrectly even when it fits the past measurements beautifully.

Differentiating noisy data. Several modern methods need estimates of *rates of change* from data — how fast a quantity is changing, not just its value. Estimating a derivative from noisy measurements is notoriously unstable, because differentiation amplifies noise. How to do this reliably is a recurring technical thread, and it will turn out to be the bottleneck in one of the most powerful techniques we study.

Each of these difficulties is a place where naive approaches fail and where the methods of this course earn their keep. They are also, importantly, not independent: sparsity worsens identifiability, noise interacts with misspecification to make a wrong model look adequate, and the instability of numerical differentiation is most acute precisely when data are sparse and noisy. A recurring strategic question, then, is which difficulty dominates a given problem, because the right response differs. Throwing a more flexible model at an ill-posed problem makes matters worse, not better; the cure for too little information in the data is not a more powerful algorithm but more or better-placed data, or stronger prior assumptions, clearly stated.

1.5 The running example: tumor growth from sparse data

We will keep returning to one problem, stated here so that the later chapters have a shared reference point.

A solid tumor, left untreated, tends to grow rapidly when small and then slow as it approaches a maximum size set by its blood supply and environment. A standard mathematical description is the *Gompertz model*,

$$\frac{dV}{dt} = a\,V \ln\!\left(\frac{K}{V}\right), \tag{1.1}$$

where $V(t)$ is the tumor volume at time t, the parameter $a > 0$ sets the growth rate, and $K > 0$ is the *carrying capacity*, the size the tumor approaches as $t \to \infty$. (We will derive properties of this model in Chapter 2; for now it is enough to know it produces the familiar S-shaped, or *sigmoidal*, growth curve.)

In a clinical or laboratory setting one does not know a or K in advance. One has instead a few measurements of V at various times — perhaps three to ten of them, each uncertain by ten or twenty percent — and the task is to estimate the parameters, and then to predict how the tumor will behave in the future. This is exactly an inverse problem of the kind described above, and it raises every difficulty we have just named:

- the measurements are *noisy*;
- they are *sparse*;

- the parameters may be poorly *identifiable* from so few points;
- we may have chosen the *wrong growth law* (perhaps the tumor does not obey Gompertz dynamics at all); and
- some methods will require us to estimate $\frac{dV}{dt}$, the growth *rate*, from the noisy measurements.

We choose this example deliberately, for several reasons. It is *low-dimensional* — two parameters — so its difficulties can be visualized and reasoned about directly, rather than hidden in high-dimensional abstraction. It is *mechanistically meaningful*: the parameters a and K have interpretations a biologist cares about, so the question of whether we recover them *correctly*, not merely whether we fit the data, is a real one. And it is *genuinely hard*: as the later chapters show, even this simple problem defeats naive applications of powerful methods, which makes it an ideal teacher. A problem easy enough to be solved by any method would teach us nothing about when methods fail.

Over the course of the notes we will attack this problem with successively more sophisticated tools. We begin with the classical approach of least squares (Chapters 4–5); we then ask whether neural networks, and in particular *physics-informed* neural networks, can do better (Chapters 7–10); and finally we ask the most ambitious question of all — whether we can *discover the growth law itself* from the data, rather than assuming it (Chapter 12). The final chapters assemble these pieces into a complete modeling pipeline and reflect on what was learned, including where the powerful methods failed and why.

1.6 What you will be able to do by the end

By the end of the course you should be able to:

1. formulate a scientific modeling question as an inverse problem, and state precisely what is being estimated and from what data;
2. explain and apply the core optimization methods — gradient descent, Gauss–Newton/ Levenberg–Marquardt, and the stochastic methods used to train neural networks;
3. describe what a neural network is, how it is trained, and what automatic differentiation computes;
4. explain how physical knowledge, such as a differential equation, can be built into a learning procedure, and what is gained and risked by doing so;
5. reason about identifiability and model misspecification, and design validation procedures (such as held-out forecasting) that expose them;
6. describe how symbolic regression attempts to discover equations from data, and why estimating derivatives from noisy data is the central obstacle; and
7. assess, with appropriate skepticism, claims about what artificial intelligence can and cannot do in the sciences.

These objectives are deliberately a mix of *technique* and *judgment*. The techniques — how to run an optimizer, how to set up a network — are the easier half, and the half most readily learned elsewhere. The judgment — knowing when a fit is trustworthy, when a problem is ill-posed, when a discovered equation is an artifact — is the harder and more durable half, and it is what the running example is designed to build.

1.7 Roadmap of the notes

The notes are organized in five parts.

Part I (Foundations) fixes notation and develops the language of models, data, and error. *Part II (Optimization)* builds the machinery of minimizing error, culminating in the classical least-squares methods that serve as our baseline. *Part III (Neural networks)* introduces neural networks and the automatic differentiation that makes them practical. *Part IV (Scientific machine learning)* brings physical knowledge into the picture — physics-informed neural networks, the theory of inverse problems and identifiability, and symbolic regression for discovering equations. *Part V (Applications and synthesis)* works the tumor-growth problem in full as an extended case study, discusses the role of large language models in scientific discovery, and closes with the principles of model selection and validation.

A reader pressed for time, or teaching a shorter course, can extract a coherent path through the essentials by reading Chapters 1, 2, and 3 for foundations; Chapters 4 and 5 for optimization and the classical baseline; Chapter 11 for the central conceptual content on identifiability and misspecification; and Chapters 13 and 15 for the synthesis. The neural-network and symbolic-regression chapters, while important, can be treated as elaborations of the core theme — that flexible methods must be disciplined by validation and prior knowledge — rather than as prerequisites for understanding it.

Exercises

Exercise 1.1 (*Classifying problems*). For each of the following, state whether it is primarily a forward problem or an inverse problem, and identify the model, the data (if any), and what is being computed or estimated.

(a) Predicting tomorrow's high temperature from a weather model whose parameters are known.

(b) Estimating the half-life of a radioactive isotope from a series of decreasing radiation measurements.

(c) Computing the trajectory of a projectile given its initial speed and angle.

(d) Determining the spring constant of a spring from measurements of its stretch under various known weights.

Exercise 1.2 (*Why the logistic levels off*). The logistic model in the text has the property that $V(t)$ approaches K as t grows large. Without solving the differential equation, explain why: what does $\frac{dV}{dt}$ do as V approaches K, and why does that force V to level off?

Exercise 1.3 (*Two fits, one dataset*). Suppose two students each fit the same model to the same five noisy data points and obtain different parameter values, yet both report an excellent fit (their curves pass very close to all five points). Give a plausible explanation for how this can happen. (You are not expected to be rigorous; the point is to anticipate the idea of identifiability.)

Exercise 1.4 (*A deceptive singularity*). Consider the Gompertz model (1.1). As V becomes very small (close to 0), the factor $\ln(K/V)$ becomes very large. Does this mean the growth rate $\frac{dV}{dt}$ becomes very large as well? Examine the product $V \ln(K/V)$ as $V \to 0^+$ and describe what happens. *(Hint: consider what each factor does, and recall that a quantity going to zero can overcome a quantity going to infinity. We will return to this point when we discuss numerical pitfalls.)*

Exercise 1.5 (*Recognizing the difficulties*). In one or two sentences each, give a real example from any field you know of in which (a) data is sparse, (b) measurements are noisy, and (c) choosing the wrong model could lead to a confidently wrong conclusion.

Exercise 1.6 (*The near-singular linear problem*). Return to Example 1.2. Solve the system for (x_1, x_2) in terms of (y_1, y_2) exactly. Then suppose y_2 is measured with an error of $+0.001$; how much does the recovered x_2 change? Which combination of x_1 and x_2 is well determined, and which is not?

Exercise 1.7 (*Forward maps are nicer*). Explain in your own words why the forward map (parameters to observations) is usually well behaved while the inverse map (observations to parameters) may be many-to-one or discontinuous. Give a one-sentence example of each pathology.

Exercise 1.8 (*Inductive reasoning*). The text describes scientific machine learning as automated inductive reasoning. Pick any everyday inference you make from data (for instance, judging from a few past experiences how long a commute will take) and identify, in that inference, the analogue of the *model*, the *data*, and the *fitting procedure*.

Chapter 2

Mathematical Preliminaries

This chapter collects the mathematical language the rest of the notes will speak. None of it is deep, and a reader comfortable with multivariable calculus and linear algebra may skim it, pausing only at the running-example material in Sections 2.5 and 2.6, which sets up numbers used throughout the book. The aim is not to develop these topics in full but to fix notation and to isolate the few facts we will lean on repeatedly: that the gradient points uphill and vanishes at minima, that Taylor expansion turns any smooth function into a line or parabola locally, that Gaussian noise is the default model of measurement error, and that a differential equation together with a starting value determines a unique trajectory. Each section ends with a worked computation of the kind that recurs later, so that the abstract statements are anchored to concrete manipulation.

2.1 Vectors, matrices, and the objects we optimize over

A *model* has *parameters*, which we collect into a vector $\theta = (\theta_1, \dots, \theta_p) \in \mathbb{R}^p$. For the Gompertz model (1.1) the parameters are the growth rate and the carrying capacity, so $\theta = (a, K)$ and $p = 2$. Fitting the model means searching the parameter space $\mathbb{R}^p$ (or a sensible subset of it, since a and K must be positive) for the value of θ that best matches the data.

Data also live in vectors. If we measure the tumor volume at times $t_1, \dots, t_N$ and obtain values $y_1, \dots, y_N$, we may write the measurements as a vector $y = (y_1, \dots, y_N) \in \mathbb{R}^N$. Note the two different dimensions already in play: the parameter vector has length p (here 2), while the data vector has length N (the number of measurements, here perhaps 5 or 10). Keeping track of which space we are in — the small parameter space or the larger data space — prevents a great deal of confusion later.

The central object we manipulate is a function

$$L : \mathbb{R}^p \to \mathbb{R}, \qquad \theta \mapsto L(\theta),$$

called the *loss* or *objective*, which measures how badly a given parameter vector fits the data. Smaller is better. All of Part II is about methods for minimizing such functions.

Matrix–vector products. Linear models and the layers of neural networks are built from the operation $\theta \mapsto A\theta$, where A is a matrix. If A is $m \times n$ then it maps a vector in $\mathbb{R}^n$ to a vector in $\mathbb{R}^m$; the requirement that inner dimensions match is the bookkeeping that governs how network layers connect (Chapter 7). We assume familiarity with matrix multiplication and with the idea that a matrix represents a linear transformation.

Rank, range, and what a matrix can represent. Two notions from linear algebra recur. The *range* (or column space) of a matrix A is the set of vectors $A\theta$ obtainable as θ varies — the outputs the linear model can produce. The *rank* is the dimension of the range. When a matrix arising in fitting has rank smaller than its number of columns — it is *rank-deficient* — distinct parameter vectors θ produce identical outputs $A\theta$, and the parameters cannot be recovered from the output. This is the linear prototype of the identifiability failures of Chapter 11, and it is no accident that the same words — rank, range, null space — reappear there. A parameter direction in the *null space* of A (the set of θ with $A\theta = 0$) is completely invisible to the data: it can be added to any solution without changing the predictions.

Norms. To say a prediction is "close" to the data we need a notion of distance. Throughout we use the *Euclidean norm*: for $v \in \mathbb{R}^N$,

$$\|v\| = \|v\|_2 = \sqrt{\sum_{i=1}^{N} v_i^2}.$$

The squared Euclidean distance between a prediction vector $\hat{y}$ and the data y,

$$\|\hat{y} - y\|^2 = \sum_{i=1}^{N} (\hat{y}_i - y_i)^2,$$

is the quantity at the heart of *least squares* (Chapter 5) and, after dividing by N, of the *mean squared error* (Chapter 3). Other norms occasionally appear — the ℓ_1 norm $\|v\|_1 = \sum_i |v_i|$, which is more robust to outliers, and the maximum norm $\|v\|_\infty = \max_i |v_i|$ — but the Euclidean norm is the default, partly for the statistical reason developed in Chapter 5 and partly because it is smooth and so amenable to the calculus of Part II.

2.2 Gradients and the geometry of several variables

Let $f : \mathbb{R}^p \to \mathbb{R}$ be a smooth function. Its *partial derivative* $\frac{\partial f}{\partial \theta_j}$ measures the rate of change of f as we vary the jth coordinate alone. Collecting the partials into a vector gives the *gradient*,

$$\nabla f(\theta) = \left(\frac{\partial f}{\partial \theta_1}, \ldots, \frac{\partial f}{\partial \theta_p} \right).$$

Two facts about the gradient organize everything in Part II.

1. **The gradient points in the direction of steepest increase.** If we stand at θ and wish to increase f as rapidly as possible, we should step in the direction $\nabla f(\theta)$; to *decrease* f we step in the opposite direction, $-\nabla f(\theta)$. This is the entire idea behind gradient descent (Chapter 4).

2. **At a minimum, the gradient is zero.** If $\theta^\star$ is a local minimum of f and f is differentiable there, then

$$\nabla f(\theta^\star) = 0. \tag{2.1}$$

   This *first-order condition* is how we characterize candidate solutions: we look for points where the gradient vanishes.

To see why the gradient is the direction of steepest ascent, recall the *directional derivative*. The rate of change of f at θ in a unit direction u is $\nabla f(\theta)^\top u$. By the Cauchy–Schwarz inequality this is largest, over all unit vectors u, when u points along $\nabla f(\theta)$ itself, and its maximum value is $\|\nabla f(\theta)\|$. Thus the gradient's direction is the steepest uphill direction and its magnitude is the steepness. This small computation is the whole geometric content of gradient descent, and it is worth carrying in mind.

The first-order condition is necessary but not sufficient: a vanishing gradient also occurs at maxima and at saddle points. To distinguish a minimum we look at curvature.

The Hessian and curvature. The second derivatives of f assemble into the *Hessian* matrix $\nabla^2 f(\theta)$, whose (i, j) entry is $\partial^2 f/\partial\theta_i\partial\theta_j$. Informally, the Hessian describes how the surface curves. At a minimum the surface curves upward in every direction, which corresponds to the Hessian being *positive definite* (all its eigenvalues positive). A direction in which the Hessian has a very small eigenvalue is a direction in which the surface is nearly flat — and such flat directions, we will see in Chapter 11, are exactly what make an inverse problem hard, because the data barely constrain the parameter combination that points along them.

The eigenvalues of the Hessian have a clean geometric meaning that we will invoke repeatedly. Near a minimum the function looks like a bowl (we make this precise with Taylor's theorem below), and the eigenvectors of the Hessian are the principal axes of that bowl while the eigenvalues are its curvatures along those axes. A large eigenvalue is a steeply curved, tightly constrained direction; a small eigenvalue is a gently curved, loosely constrained one. The ratio of largest to smallest eigenvalue, the *condition number*, measures how elongated the bowl is, and it controls both the speed of gradient descent (Chapter 4) and the severity of an identifiability problem (Chapter 11).

Remark 2.1 (Second-order condition*)**.** If $\nabla f(\theta^\star) = 0$ and $\nabla^2 f(\theta^\star)$ is positive definite, then $\theta^\star$ is a strict local minimum. If the Hessian is positive *semidefinite* (some eigenvalues zero) the test is inconclusive at second order. We will not prove this; it is the multivariable analogue of the second-derivative test from one-variable calculus, and it follows from the second-order Taylor expansion (2.2) below.

Example 2.2 (Gradient of a small least-squares loss)**.** Suppose our model predicts, at input x_i, the value $m(x_i;\theta) = \theta_1 + \theta_2 x_i$ (a straight line with intercept θ_1 and slope θ_2), and we measure y_i at inputs $x_1, \dots, x_N$. The least-squares loss is

$$L(\theta) = \sum_{i=1}^{N} \big(\theta_1 + \theta_2 x_i - y_i\big)^2.$$

Its partial derivatives are

$$\frac{\partial L}{\partial \theta_1} = 2\sum_{i=1}^{N}(\theta_1 + \theta_2 x_i - y_i), \qquad \frac{\partial L}{\partial \theta_2} = 2\sum_{i=1}^{N}(\theta_1 + \theta_2 x_i - y_i)\, x_i.$$

Setting both to zero gives two linear equations in θ_1, θ_2 — the *normal equations* — whose solution is the best-fit line. We will meet this calculation again, in matrix form, in Chapter 5; for now it is a concrete instance of using (2.1) to find a minimum. Notice that because the model is linear in θ, the gradient is linear in θ, so the stationarity equations are linear and have a closed-form solution. This is precisely the feature that nonlinear models like Gompertz lack, which is why they require the iterative methods of Chapter 5.

Example 2.3 (A nonlinear gradient)**.** Contrast the linear case with the Gompertz loss for a single data point, $L(a, K) = (V(t_1; a, K) - y_1)^2$, where $V(t; a, K) = K \exp(\ln(V_0/K)e^{-at})$. The partial derivatives now involve the chain rule applied through the exponential, and they depend nonlinearly on a and K; setting them to zero gives transcendental equations with no closed form. We will compute such derivatives systematically — by hand in Chapter 5 and automatically in Chapter 8 — but the point here is qualitative: the step from a linear to a nonlinear model is the step from "solve a linear system once" to "iterate."

2.3 Taylor expansion: the universal local approximation

The single most useful analytical tool in the course is the idea that a smooth function, examined closely enough near a point, looks like a low-degree polynomial.

In one variable, if f is twice differentiable near a, then for x close to a,

$$f(x) \approx f(a) + f'(a)(x - a) + \tfrac{1}{2} f''(a)(x - a)^2.$$

The linear part $f(a) + f'(a)(x - a)$ is the tangent line; adding the quadratic term gives the best-fitting parabola.

In several variables the analogue, expanded about $\theta_0 \in \mathbb{R}^p$, is

$$f(\theta) \approx f(\theta_0) + \nabla f(\theta_0)^\top (\theta - \theta_0) + \tfrac{1}{2} (\theta - \theta_0)^\top \nabla^2 f(\theta_0)(\theta - \theta_0). \tag{2.2}$$

This formula is the engine behind the optimization methods of Part II. Gradient descent keeps only the linear term and steps downhill along $-\nabla f$. Newton's method and its practical cousin, the Levenberg–Marquardt algorithm (Chapter 5), keep the quadratic term as well and use the curvature to take a smarter step. Whenever you see a method "approximate the objective locally," it means: replace f by its Taylor expansion (2.2).

The quadratic term deserves a second look because it is where the Hessian enters and where the geometry of the previous section becomes concrete. At a stationary point $\theta^\star$ the linear term vanishes, and (2.2) reduces to

$$f(\theta) \approx f(\theta^\star) + \tfrac{1}{2} (\theta - \theta^\star)^\top \nabla^2 f(\theta^\star)(\theta - \theta^\star).$$

This is exactly the equation of a bowl (a paraboloid) centered at $\theta^\star$, whose shape is the Hessian. If the Hessian is positive definite the bowl opens upward and $\theta^\star$ is a minimum; if it has a near-zero eigenvalue the bowl has a nearly flat floor along the corresponding eigenvector, and the minimum is poorly localized in that direction. Every later statement about "the loss surface near the optimum" is, at bottom, this approximation.

Remark 2.4 (The error of the approximation*)**.** Taylor's theorem makes the "$\approx$" precise: the error in the first-order expansion is controlled by the size of the second derivative over the interval, and the error in the second-order expansion by the third derivative. Specifically, in one variable, the first-order remainder is $\frac{1}{2} f''(\xi)(x - a)^2$ for some ξ between a and x. For our purposes the qualitative statement — the approximation is good when θ is close to θ_0, and better the smaller the neglected higher derivatives — is what matters.

Example 2.5 (Linearizing the Gompertz rate near saturation)**.** A Taylor expansion we will use in Chapter 11 concerns the Gompertz right-hand side $g(V) = aV \ln(K/V)$ near the carrying capacity $V = K$. Write $V = K(1 - s)$ for small $s > 0$. Then $\ln(K/V) = -\ln(1 - s) = s + \frac{1}{2} s^2 + \cdots$, and

$$g(V) = aK(1 - s)\left(s + \tfrac{1}{2} s^2 + \cdots\right) = aKs - \tfrac{1}{2} aKs^2 + \cdots .$$

Since $Ks = K - V$, the leading term is $g(V) \approx a(K - V)$: near saturation the Gompertz dynamics are approximately *linear relaxation* toward K at rate a. We will see that the logistic and von Bertalanffy laws have the same linear form near their carrying capacities (with different rate constants), which is precisely why the three are hard to tell apart from data taken near saturation.

2.4 A minimal amount of probability

Measurements are noisy, and to reason about noise we need a little probability. A *random variable* is a quantity whose value is uncertain; its *mean* (or expectation) $\mathbb{E}[X]$ is its average value over many repetitions, and its *variance* $\mathrm{Var}(X) = \mathbb{E}[(X - \mathbb{E}[X])^2]$ measures how much it fluctuates around that mean. The square root of the variance is the *standard deviation*, which carries the same units as X itself and is therefore the more interpretable measure of spread.

Two elementary properties of variance will be used without comment. First, for a constant c, $\mathrm{Var}(cX) = c^2\mathrm{Var}(X)$ — scaling a quantity scales its variance by the square. Second, for *independent* random variables, variances add: $\mathrm{Var}(X + Y) = \mathrm{Var}(X) + \mathrm{Var}(Y)$. These two facts are exactly what we need to analyze how noise propagates through a calculation, as in the differentiation of noisy data in Chapter 12.

The default model of measurement noise is the *normal* (or *Gaussian*) distribution, written $\mathcal{N}(\mu, \sigma^2)$ with mean μ and variance σ^2. Its density is the familiar bell curve $\frac{1}{\sqrt{2\pi}\sigma}\exp(-(x-\mu)^2/2\sigma^2)$, concentrating most of its probability within a couple of standard deviations of the mean. The normal distribution earns its default status partly through the central limit theorem (many small independent errors sum to something approximately normal) and partly through mathematical convenience: as we will see in Chapter 5, assuming Gaussian noise makes least squares the natural fitting criterion, because the exponent of the Gaussian density is precisely a sum of squares.

The noise model of the running example. Throughout the notes the synthetic tumor data is generated by taking a true trajectory $V(t_i)$ and corrupting it *multiplicatively*:

$$y_i = V(t_i)\,(1 + \sigma\varepsilon_i), \qquad \varepsilon_i \sim \mathcal{N}(0, 1), \tag{2.3}$$

where σ is the noise level. Multiplicative noise — as opposed to the additive model $y_i = V(t_i) + \sigma\varepsilon_i$ — means the *relative* error is roughly constant: a measurement is uncertain by about σ as a *fraction* of its size. With $\sigma = 0.10$ we speak of "ten percent noise." This choice reflects the fact that larger tumors are measured with larger absolute error. A consequence we will use later is that the variance of the noise on the ith measurement is $\sigma^2 V(t_i)^2$, which grows with the size of the tumor — so that, perhaps surprisingly, the measurements taken late (when the tumor is large) are the *noisiest* in absolute terms, even though they are equally noisy in relative terms.

Why we average over noise realizations. A subtle but important habit: because the data is random, any single fitted result is itself one draw from a distribution of possible results. A method that does well on one noisy dataset may do poorly on the next. Repeated evaluation therefore averages a method's performance over many independent noise realizations — typically dozens — and reports a typical value (such as the median) together with a measure of spread. We will see in Chapter 11 a striking case in which a single run badly misrepresents a method's typical behavior, and the discipline of averaging is what reveals the truth. The distinction is between a method's performance on *this* dataset, which is an accident of the particular noise drawn, and its performance *on average*, which is a property of the method — and it is the latter we usually want to know.

2.5 Ordinary differential equations as models

Many scientific models take the form of an *ordinary differential equation* (ODE), a rule that specifies the rate of change of a quantity in terms of its current value. The general autonomous first-order form is

$$\frac{dV}{dt} = f(V), \qquad V(0) = V_0, \tag{2.4}$$

where f is the *right-hand side* encoding the dynamics and V_0 is the *initial condition*. The word *autonomous* means f depends on V but not explicitly on t — the rule does not change over time — which will matter when we try to *discover* f from data (Chapter 12).

The key structural fact, which we state informally, is that a rule plus a starting point determines the future uniquely.

Theorem 2.6 (Existence and uniqueness, informal)**.** *If f is continuously differentiable, then the initial-value problem (2.4) has exactly one solution $V(t)$ on some interval around $t = 0$. One rule and one starting point produce one trajectory.*

This is what makes the *forward* problem well-posed: given f and V_0, the trajectory is determined, and we can compute it. The *inverse* problem — recovering f (or its parameters) and possibly V_0 from samples of V — is the hard direction and the subject of much of the course. It is worth appreciating that uniqueness in the forward direction does *not* imply uniqueness in the inverse direction: many different right-hand sides f can produce trajectories that agree on a finite set of sampled times, which is the source of the misspecification difficulties to come.

Separable equations and the Gompertz solution. A first-order ODE $\frac{dV}{dt} = f(V)$ is *separable*: formally, $\frac{dV}{f(V)} = dt$, and integrating both sides gives an implicit solution. The Gompertz model can be solved this way, but a substitution is cleaner. Starting from

$$\frac{dV}{dt} = aV \ln\!\left(\frac{K}{V}\right),$$

introduce $u = \ln(V/K)$, so that $V = Ke^u$ and $\frac{dV}{dt} = Ke^u \frac{du}{dt}$. Since $\ln(K/V) = -u$, the equation becomes

$$Ke^u \frac{du}{dt} = a(Ke^u)(-u) \quad \Longrightarrow \quad \frac{du}{dt} = -au.$$

This is linear with solution $u(t) = u(0)e^{-at}$, where $u(0) = \ln(V_0/K)$. Undoing the substitution,

$$V(t) = K \exp\!\left(\ln\!\left(\frac{V_0}{K}\right) e^{-at}\right). \tag{2.5}$$

One checks that $V(0) = V_0$ and that $V(t) \to K$ as $t \to \infty$, since the exponent tends to 0. The curve rises steeply at first and bends over toward the ceiling K — the sigmoidal shape characteristic of growth under a resource limit. The substitution $u = \ln(V/K)$ that linearized the equation is the same change of variable that, in the physics-informed network of Chapter 10, motivates working with $\log a$ and $\log K$ rather than a and K.

The inflection point. A feature of the Gompertz curve we will need is its *inflection point*, where the growth rate $\frac{dV}{dt}$ is maximal — the moment of fastest growth. Differentiating $g(V) = aV \ln(K/V)$ with respect to V and setting the result to zero gives $a(\ln(K/V) - 1) = 0$, so the rate peaks at $V = K/e \approx 0.368\,K$. The location of this peak, and more generally where the trajectory is most informative about the parameters, governs the sampling-design discussion of Chapter 11.

When there is no formula: numerical solution. Most ODEs cannot be solved in closed form. We can nonetheless compute their trajectories numerically by stepping forward in small increments: knowing V at time t, we use the rule $f(V)$ to estimate V a small step Δt later, and repeat. The simplest such scheme is *Euler's method*,

$$V(t + \Delta t) \approx V(t) + \Delta t\, f(V(t)),$$

which simply follows the tangent line for one step. Its error per step is of order Δt^2, accumulating to order Δt over a fixed interval, so halving the step size roughly halves the error; practical software uses more accurate variants (such as Runge–Kutta methods) that achieve far smaller error for the same step size. The essential point for us is that *we can always simulate a model forward*, even when no formula exists — a fact the discovery pipeline of Chapter 12 relies on when it refits candidate laws by solving their ODEs and comparing the resulting trajectories to data.

2.6 The shape of the running example

We can now assemble the running example precisely. The ingredients are:

- a *model*, the Gompertz ODE (1.1), with unknown parameters $\theta = (a, K)$;
- *data*, sparse noisy samples $y_i = V(t_i)(1 + \sigma\varepsilon_i)$ generated by (2.3) at a handful of times t_i;
- an *error criterion*, the mean squared error between the model's trajectory and the data, developed in the next chapter.

To study *misspecification* — the possibility that we have assumed the wrong law — we will compare three families of growth model, all of which produce sigmoidal curves but with different functional forms:

$$\text{Gompertz:} \quad \frac{dV}{dt} = aV \ln(K/V), \tag{2.6}$$

$$\text{Richards (generalized logistic):} \quad \frac{dV}{dt} = rV\left(1 - (V/K)^{\nu}\right), \tag{2.7}$$

$$\text{von Bertalanffy:} \quad \frac{dV}{dt} = \alpha V^{2/3} - \beta V. \tag{2.8}$$

Each has a carrying capacity — the value of V at which $\frac{dV}{dt} = 0$ — and we will choose parameters so that all three share the same carrying capacity $V^\star = 1000$ and the same initial volume $V_0 = 10$. For the Gompertz and Richards families the carrying capacity is simply K; for von Bertalanffy, setting $\alpha V^{2/3} = \beta V$ gives $V^\star = (\alpha/\beta)^3$, so the choice $\alpha = 4$, $\beta = 0.4$ yields $V^\star = 10^3 = 1000$ to match. This matching makes the three curves genuinely similar over the range where we observe them, so that telling them apart from sparse noisy data is a real challenge. The consequences of getting the family wrong — biased parameters, mistaken forecasts — occupy Chapters 11 and 13.

It is worth sketching the three right-hand sides $f(V)$ as functions of V on the interval $[0, 1000]$: each is positive (the tumor grows) below $V^\star$ and crosses zero at $V^\star$, but the *shape* of the approach differs, and that shape is precisely the fingerprint a discovery method must read. A second instructive sketch is the trajectories $V(t)$ themselves on a common time axis: matched at V_0 and $V^\star$, they are nearly indistinguishable to the eye over a typical observation window, parting company only in the details of the rising phase and just after. That near-coincidence, visible in a sketch, is the entire difficulty of the misspecification problem in one picture.

Exercises

Exercise 2.1 (*A simple gradient and Hessian*). Let $L(\theta_1, \theta_2) = (\theta_1 - 3)^2 + 4(\theta_2 + 1)^2$. Compute ∇L, find the unique point where it vanishes, and verify using the Hessian that this point is a minimum. What are the two eigenvalues of the Hessian, and what is the condition number?

Exercise 2.2 (*Normal equations*). Carry out the calculation of Example 2.2 for the data $(x_i, y_i) = (0, 1), (1, 3), (2, 3)$: write the two normal equations and solve for the best-fit line $\theta_1 + \theta_2 x$.

Exercise 2.3 (*Directional derivative*). Let $f(\theta_1, \theta_2) = \theta_1^2 + 3\theta_1\theta_2$. Compute ∇f at $(1, 1)$, and find the unit direction of steepest ascent there. What is the rate of increase in that direction? *(Hint: it is the norm of the gradient.)*

Exercise 2.4 (*Verifying the Gompertz solution*). Substitute (2.5) back into the Gompertz ODE (1.1) and confirm that it satisfies both the equation and the initial condition $V(0) = V_0$.

Exercise 2.5 (*Equilibria of the three families*). For each of the three growth laws (2.6)–(2.8), find the positive value of V at which $\frac{dV}{dt} = 0$ (the carrying capacity). For von Bertalanffy, express $V^\star$ in terms of α and β, and verify that $\alpha = 4, \beta = 0.4$ gives $V^\star = 1000$.

Exercise 2.6 (*Taylor approximation near saturation*). Following Example 2.5, write the first- and second-order Taylor approximations of $g(V) = V \ln(K/V)$ about the point $V = K$. *(This local behavior near the carrying capacity will reappear in Chapter 11.)*

Exercise 2.7 (*Multiplicative noise*). Under the noise model (2.3), show that the variance of the measurement y_i, given the true value $V(t_i)$, equals $\sigma^2 V(t_i)^2$. Explain in words why this means larger tumors are measured with larger absolute uncertainty but the same relative uncertainty.

Exercise 2.8 (*The inflection point*). Verify the claim in the text that the Gompertz growth rate $\frac{dV}{dt}$ is maximal at $V = K/e$. *(Differentiate the right-hand side $g(V) = aV \ln(K/V)$ with respect to V and set it to zero.)* Where on the trajectory — early, middle, or late — does this occur for the benchmark parameters $V_0 = 10$, $K = 1000$?

Exercise 2.9 (*Variance propagation*). Two independent measurements X and Y each have variance σ^2. Using the two properties of variance stated in the text, find the variance of $\frac{1}{2}(X + Y)$ and of $X - Y$. Which is smaller, and what does this suggest about averaging repeated measurements?

Exercise 2.10 (*One Euler step*). For the logistic equation $\frac{dV}{dt} = 0.4\,V(1 - V/1000)$ with $V(0) = 10$, take a single Euler step of size $\Delta t = 1$ to estimate $V(1)$. *(You need only evaluate the right-hand side at $V = 10$ and follow the tangent line.)*

Chapter 3

Models, Data, and Error

The previous chapter assembled our vocabulary. This one turns it into a working framework: what exactly is a model, what does it mean to fit one to data, and how do we measure success in a way that does not fool us? These questions sound elementary, and the basic answers are, but the chapter also introduces the ideas that separate competent modeling from naive curve-fitting — the distinction between fitting and generalizing, the phenomenon of overfitting, and the bias–variance tradeoff that explains it — which recur in every later chapter.

3.1 What a model is

A *parametric model* is a family of functions indexed by parameters. In the running example the family is the set of Gompertz trajectories

$$\mathcal{M} = \left\{ V(\,\cdot\,; a, K) : a > 0,\ K > 0 \right\},$$

one curve for each choice of (a, K). Fitting the model means selecting the member of this family that best matches the data. The richness of a model is governed by how many parameters it has and how flexibly they reshape the predictions: a straight line ($\theta_1 + \theta_2 x$, two parameters) is a rigid model; a high-degree polynomial or a large neural network is a flexible one. A theme we develop below, and return to in Chapter 15, is that flexibility is double-edged: a more flexible model can fit more, including the noise we would rather it ignore.

It is worth distinguishing two ways a model can relate to reality. A *mechanistic* model encodes a hypothesis about how the system works — the Gompertz equation embodies the idea that the per-capita growth rate $\frac{1}{V}\frac{dV}{dt} = a\ln(K/V)$ declines logarithmically as the tumor fills its niche. A *phenomenological* or *black-box* model, such as a polynomial or a neural network, makes no such commitment; it simply offers enough flexibility to match whatever shape the data take. Much of this course lives in the tension between the two: mechanistic models are interpretable and extrapolate sensibly when correct, but are useless when misspecified; black-box models are flexible but can fit nonsense. The scientific-machine-learning methods of Part IV are largely attempts to get the benefits of both.

The distinction also governs what a fitted model is *for*. A mechanistic model is fit in order to learn its parameters — the growth rate, the carrying capacity — which are themselves the scientific quantities of interest, and to extrapolate beyond the data on the strength of the mechanism. A phenomenological model is fit in order to interpolate or summarize; its parameters (the coefficients of a polynomial, the weights of a network) are usually not interpretable and not the point. Confusing these aims is a common error: reading mechanistic meaning into the coefficients of a black-box

fit, or judging a mechanistic model only by its interpolation accuracy while ignoring whether its parameters are correct. The running example keeps the distinction sharp, because there the parameters genuinely matter.

3.2 Residuals and the mean squared error

Given a parameter vector θ and data (t_i, y_i) for $i = 1, \dots, N$, the *residual* at the ith point is the discrepancy between model and measurement,

$$r_i(\theta) = m(t_i; \theta) - y_i,$$

where $m(t_i; \theta)$ is the model's prediction at t_i. For the Gompertz model the prediction is the trajectory value $V(t_i; a, K)$ from (2.5).

The standard way to aggregate the residuals into a single number is the *mean squared error* (MSE),

$$\mathrm{MSE}(\theta) = \frac{1}{N} \sum_{i=1}^{N} r_i(\theta)^2 = \frac{1}{N} \sum_{i=1}^{N} \big(m(t_i; \theta) - y_i\big)^2. \tag{3.1}$$

The closely related *sum of squared errors* (SSE) omits the division by N; the two differ only by a constant factor and have the same minimizer. Squaring serves two purposes: it makes positive and negative residuals count equally, and — as we will see in Chapter 5 — it is exactly the criterion that emerges from assuming Gaussian measurement noise.

Other error measures. The squared error is not the only choice, and the alternatives illuminate it by contrast. The *mean absolute error* $\frac{1}{N} \sum_i |r_i|$ penalizes large residuals less severely, making it more robust to outliers but non-differentiable at zero residual, which complicates optimization. The *maximum error* $\max_i |r_i|$ cares only about the worst point. Each corresponds to a different implicit assumption about the noise (the absolute error to heavier-tailed noise, for instance) and a different notion of what "a good fit" means. We default to squared error for its smoothness and its Gaussian pedigree, but a practitioner should know that the choice of error measure is a modeling decision, not a law of nature, and that a fit can look good under one measure and poor under another.

Remark 3.1 (Units and the meaning of an MSE value)**.** Because the residuals carry the units of V (here, mm^3), the MSE carries units of V^2 (mm^6), an awkward quantity to interpret directly. It is often more meaningful to compare an MSE against the variance of the noise. If the measurements have noise variance $\sigma^2_{\mathrm{noise}}$, then no model, however good, should drive the MSE much below $\sigma^2_{\mathrm{noise}}$ — doing so means the model is fitting the noise. We make this benchmark precise in Chapter 15 and use it in the case study of Chapter 13, where a misspecified model that achieves a "too good" training error is exposed precisely by this comparison. A useful dimensionless summary is therefore the *ratio* of the MSE to the noise variance: a value near one indicates a model that has captured the signal and is now fitting only noise; a value well below one signals overfitting; a value well above one signals that the model is missing real structure.

3.3 Fitting versus generalizing

Here is the central conceptual distinction of the chapter. There are two very different questions one might ask of a fitted model:

1. **Goodness of fit:** how well does the model match the data we used to fit it?

2. **Generalization:** how well does the model predict data we did *not* use — future measurements, or measurements at new conditions?

These are not the same, and conflating them is the most common error in applied modeling. A model can achieve a superb goodness of fit and yet generalize terribly. The training MSE (3.1) measures only the first.

In the running example the distinction is sharp and practical. We fit the growth model to measurements taken over, say, the first twenty days. Goodness of fit asks how close the fitted curve passes to those measurements. Generalization asks something we actually care about more: does the model correctly predict the tumor's size on day forty? A model can thread the early measurements beautifully and still forecast the future wildly wrong — especially if the model is misspecified or the parameters are poorly identified. We will therefore lean heavily, throughout the applied chapters, on *held-out forecasting* as the true measure of a model's worth: hide the later measurements during fitting, then test the model against them.

3.4 Overfitting

Overfitting is what happens when a model is flexible enough to fit the noise in the data, not merely the signal. The symptom is a small training error accompanied by poor generalization. The cause is a mismatch between the flexibility of the model and the information content of the data: too many free parameters, or too few and too noisy data points, or both.

Example 3.2 (Polynomial overfitting)**.** Suppose the truth is a smooth sigmoidal curve and we observe six noisy points along it. A straight line (two parameters) fits poorly — it cannot bend — but its few parameters cannot chase the noise either. A fifth-degree polynomial (six parameters) can pass exactly through all six points, driving the training error to zero. Yet between and beyond the points it may oscillate wildly, and its prediction of the seventh point can be far worse than the straight line's. The polynomial has fit the noise. This is not a contrived worry: in Chapter 12 we will see a real instance in which a polynomial surrogate fit to tumor data turns over and *decreases* past the last data point — predicting, absurdly, that the tumor shrinks — precisely because the flexible model extrapolated the noise rather than the trend.

The example illustrates a general principle. Increasing model flexibility always decreases (or cannot increase) the training error, because a more flexible model contains the less flexible one as a special case. But generalization error typically falls and then *rises* as flexibility increases: too little flexibility underfits, too much overfits, and the best generalization sits at an intermediate "sweet spot." Finding that spot — or sidestepping the problem by building in prior knowledge so that flexibility is not wasted on noise — is a recurring goal. Physics-informed methods (Chapter 10) and the deterministic filters of the discovery pipeline (Chapter 12) are both, in part, devices for preventing flexible methods from fitting nonsense.

3.5 The bias–variance tradeoff

The intermediate sweet spot has a precise explanation, one of the most useful organizing ideas in all of statistics: the *bias–variance tradeoff*. It explains *why* both too-rigid and too-flexible models generalize poorly, and it gives the vague word "flexibility" a quantitative meaning.

Imagine repeating the whole experiment many times: each time we draw a fresh noisy dataset from the same underlying truth, fit our model, and record its prediction at some test input. Because

the data are random, the prediction is random too — a different number each time. Its average error decomposes into three parts:

$$\underbrace{\text{expected squared error}}_{\text{how wrong, on average}} = \underbrace{\text{bias}^2}_{\substack{\text{systematic}\\ \text{error}}} + \underbrace{\text{variance}}_{\substack{\text{sensitivity}\\ \text{to the data}}} + \underbrace{\text{noise}}_{\text{irreducible}} . \tag{3.2}$$

The *bias* is the error of the model's average prediction — how far the typical fit sits from the truth, a systematic error that flexibility reduces. The *variance* is how much the prediction jumps around from one dataset to the next — the model's sensitivity to the particular noise it happened to see, which flexibility *increases*. The *noise* term is irreducible: it is the measurement error itself, which no model can predict.

The tradeoff is now visible in (3.2). A rigid model (a straight line) has high bias — it cannot capture the true curve — but low variance, since it is too stiff to chase the noise. A flexible model (a high-degree polynomial) has low bias — it can match almost any shape — but high variance, since it bends to fit whatever noise is present. Total error is minimized in between, where bias and variance are balanced. This is the formal content of the "sweet spot," and it reframes every later technique: a physics constraint reduces variance by restricting the fit to obey a law; held-out validation estimates the sum of bias and variance directly; and the identifiability ridge of Chapter 11 is, in these terms, an extreme variance problem, where the data leave a parameter combination almost free to vary.

Remark 3.3 (Bias, variance, and the running example)**.** The tradeoff casts the misspecification problem in a sharp light. Fitting a *wrong* mechanistic model (Gompertz to non-Gompertz data) is a high-*bias* choice: the model family does not contain the truth, so even the average fit is systematically off, no matter how much data we collect — the bias does not vanish as $N \to \infty$. Fitting an overly flexible black-box model is a high-*variance* choice. Neither extreme is safe, and the discipline of the later chapters is precisely the management of this tradeoff: enough structure to control variance, the right structure to avoid bias.

3.6 Training, validation, and test data

The practical apparatus for distinguishing fitting from generalizing is to *partition* the data. The model is fitted on a *training* set; its generalization is estimated on data held aside. In large-data machine learning one typically uses three partitions: a training set to fit parameters, a *validation* set to tune choices such as model flexibility, and a *test* set, touched only once at the very end, to give an unbiased estimate of performance.

The three roles must be kept distinct, and the reason is subtle enough to state carefully. The validation set is used to *choose* among models, so the chosen model is, by construction, the one that happened to do best on the validation set — which makes the validation error an optimistic estimate of true performance. That is why a separate, untouched test set is needed for an unbiased final number: once data has been used to make any decision, it can no longer give an unbiased estimate of the consequences of that decision. The single most common way to fool oneself in applied machine learning is to let the test set leak into the fitting process, whether by tuning against it, peeking at it, or reusing it across many modeling attempts.

In the small-data, scientific setting of this course the partitions are necessarily modest, and the most natural split is *temporal*: fit on early measurements, test on later ones. This directly measures forecasting skill, which is usually what a scientific model is for. We will use the notation of a training window (say $t \le 20$) and a forecast window (say $20 < t \le 40$) consistently in the applied

chapters. The discipline is simple to state and easy to violate: *never let the test data influence the fit.* A model that has seen the future is no longer being tested on it.

Cross-validation when data is scarce. With only a handful of points, setting aside a fixed test set wastes information one can ill afford to lose. *Cross-validation*, developed fully in Chapter 15, addresses this by rotating the role of the held-out data: each point takes a turn as the test point while the rest are used for fitting, and the held-out errors are averaged. In the extreme of *leave-one-out* cross-validation, the model is refit N times, each time omitting a single point and predicting it. This extracts the most generalization information from scarce data, at the cost of repeated fitting — a tradeoff usually worth making in the data-starved regime of scientific modeling.

3.7 A first look at the fitting problem

We can now state the inverse problem of the running example as an optimization problem, which is where Part II takes over. We seek

$$\hat{\theta} = \arg\min_{\theta} \ \mathrm{MSE}(\theta) = \arg\min_{a,K} \ \frac{1}{N}\sum_{i=1}^{N}\big(V(t_i;a,K) - y_i\big)^2, \tag{3.3}$$

the parameters minimizing the mean squared error over the training data. Two features of this problem foreshadow the difficulties ahead. First, the prediction $V(t_i;a,K)$ depends *nonlinearly* on the parameters (through the exponential in (2.5)), so the loss surface is not the simple bowl of linear least squares and may have flat directions or multiple basins. Second, with N small the surface is shaped by only a few noisy numbers, so its minimizer is itself uncertain — the high-variance situation of Section 3.5. Chapter 4 develops the tools to *find* a minimizer; Chapter 11 confronts what it means when the minimizer is not where the truth lies.

Exercises

Exercise 3.1 (*Fit versus generalization*). Give an example, from any domain, of a predictive model that fits its training data almost perfectly yet generalizes poorly. Identify what plays the role of "noise" that the model has fit.

Exercise 3.2 (*MSE arithmetic*). A model produces predictions $(2.1, 3.9, 6.2, 7.8)$ at four points where the measured values are $(2, 4, 6, 8)$. Compute the residuals, the SSE, and the MSE.

Exercise 3.3 (*Comparing error measures*). For the same data as the previous exercise, compute the mean absolute error and the maximum error. Suppose one of the measurements were instead a gross outlier, say 20 in place of 8; qualitatively, how would each of the three error measures (MSE, mean absolute, maximum) respond, and which is most robust?

Exercise 3.4 (*Flexibility and training error*). Explain why a degree-$(N-1)$ polynomial can always achieve zero training error on N data points with distinct inputs. Why does this make training error useless as a way to choose the polynomial degree?

Exercise 3.5 (*Bias or variance?*). Classify each as primarily a bias problem or a variance problem, and justify briefly: (a) fitting a straight line to data that clearly curve; (b) fitting a degree-10 polynomial to seven noisy points; (c) fitting a Gompertz model to data generated by a von Bertalanffy law; (d) fitting a large neural network to a handful of measurements.

Exercise 3.6 (*Designing a held-out test*). You are given tumor volume measurements at days 2, 5, 8, 11, 14, 17, 20, 30, and 40. Propose a training/forecast split that would let you assess whether a fitted model predicts the future well, and explain why your split is reasonable.

Exercise 3.7 (*Noise floor*). Suppose measurements have noise variance $\sigma^2_{\text{noise}} = 1600$ (in units of V^2). A student reports that their fitted model achieves a training MSE of 200. Why should this make you suspicious rather than pleased? *(This anticipates the "too good to be true" diagnostic of Chapter 15.)*

Exercise 3.8 (*Mechanistic versus phenomenological*). Classify each as primarily mechanistic or phenomenological, and say what each would and would not be good for: (a) the Gompertz growth law; (b) a cubic polynomial fit to growth data; (c) Newton's law of cooling; (d) a neural network mapping time to tumor volume.

Exercise 3.9 (*Why a separate test set*). Explain, in your own words, why the validation error is an optimistic estimate of a model's true performance, and why an untouched test set is needed for an unbiased final assessment. What specifically goes wrong if one tunes a model repeatedly against the test set?

Exercise 3.10 (*Irreducible error*). In the decomposition (3.2), the noise term cannot be reduced by any model. For the running example under 10% multiplicative noise, what is the source of this irreducible error, and why can no growth model, however good, predict it away?

Part II

Optimization

Chapter 4

Optimization Basics

Fitting a model means minimizing a loss, and minimizing is the business of *optimization*. This chapter develops the core idea — follow the gradient downhill — and the vocabulary needed to understand both the classical fitting methods of Chapter 5 and the neural-network training of Chapter 9. We keep the treatment concrete and geometric; the goal is working understanding, not a course in nonlinear programming. Throughout, the reader should keep the picture of a landscape in mind: the loss is a terrain over the parameter space, and optimization is the search for its lowest valley.

4.1 The optimization problem

We are given a loss function $L : \mathbb{R}^p \to \mathbb{R}$ and wish to find

$$\theta^\star = \arg\min_{\theta \in \mathbb{R}^p} L(\theta),$$

a parameter vector at which L is as small as possible. We distinguish a *global* minimum, the smallest value anywhere, from a *local* minimum, a point at least as good as all nearby points. For a general function the two differ, and most practical methods find only local minima; much of the art of optimization is arranging for the local minimum one finds to be good enough, or for the problem to have no bad local minima.

Recall the first-order condition (2.1): at any interior minimum the gradient vanishes, $\nabla L(\theta^\star) = 0$. Such points are called *stationary*. Stationarity is necessary but not sufficient — maxima and saddle points are stationary too — so methods that seek stationary points must be combined with some safeguard, usually the simple expedient of only ever moving downhill. A *saddle point* is a stationary point that is a minimum along some directions and a maximum along others; in high dimensions saddles vastly outnumber local minima, and a recurring concern in training large models is not getting trapped in bad minima (which turn out to be rare) but slowing to a crawl near saddles.

4.2 Convexity: when local is global

A function is *convex* if its graph curves upward everywhere, so that the line segment between any two points on the graph lies on or above the graph. The formal condition is that for all θ, ϕ and all $\lambda \in [0, 1]$,

$$L\big(\lambda\theta + (1-\lambda)\phi\big) \le \lambda L(\theta) + (1-\lambda) L(\phi).$$

For twice-differentiable functions there is an equivalent and often more usable test: L is convex if and only if its Hessian $\nabla^2 L(\theta)$ is positive semidefinite everywhere. In one dimension this is just $L''(\theta) \geq 0$ — the graph never curves downward. Convex functions are the easy case of optimization, for one decisive reason.

Theorem 4.1 (Local equals global for convex functions)**.** *If L is convex, then every local minimum is a global minimum. If L is strictly convex, the minimizer is unique.*

The least-squares loss for a *linear* model is convex — indeed a quadratic bowl — which is why linear regression has a clean, unique solution (Chapter 5). The trouble is that most models we care about are nonlinear in their parameters, and their loss surfaces are *nonconvex*: bowls dented with multiple basins, ridges, and saddle points. The Gompertz fitting problem (3.3) is nonconvex, as is every neural-network training problem. We therefore cannot expect the guarantees of the convex case; we make peace with local minima and rely on good initialization and, sometimes, multiple restarts.

Remark 4.2 (Why convexity is worth recognizing)**.** Even though our central problems are nonconvex, it pays to notice when a subproblem is convex, because convex problems can be solved reliably and without the anxieties of initialization. Linear least squares is convex; so is the problem of fitting the coefficients of a model that is linear in its parameters, however nonlinear in its inputs. A useful strategy, which recurs in the discovery pipeline of Chapter 12, is to isolate a convex subproblem (fitting linear coefficients) from the nonconvex remainder (choosing the functional form), solving the former exactly and searching only over the latter.

4.3 Gradient descent

The fundamental algorithm is *gradient descent*. Since $-\nabla L(\theta)$ is the direction of steepest decrease (Section 2.2), we repeatedly step in that direction:

$$\theta_{k+1} = \theta_k - \eta \, \nabla L(\theta_k), \tag{4.1}$$

where $\eta > 0$ is the *step size* or *learning rate*. Starting from an initial guess θ_0, the iteration (4.1) traces a path that, under suitable conditions, descends toward a stationary point.

The learning rate governs everything about the method's behavior.

- If η is *too small*, progress is slow: many iterations are needed to travel any distance, and training is expensive.
- If η is *too large*, the steps overshoot the minimum and the iteration can oscillate or diverge, the loss increasing rather than decreasing.
- A *well-chosen* η makes steady progress. In practice η is tuned by trial, or adapted automatically by the methods of Chapter 6.

Example 4.3 (Descent on a quadratic bowl)**.** Let $L(\theta) = \frac{1}{2}\theta^\top A\theta$ with A symmetric positive definite, the simplest nontrivial loss. Then $\nabla L = A\theta$ and the update is $\theta_{k+1} = (I - \eta A)\theta_k$. Writing θ_k in the eigenbasis of A, each coordinate evolves independently as $(1 - \eta\lambda_i)^k$ times its initial value, where λ_i are the eigenvalues of A. Convergence requires $|1 - \eta\lambda_i| < 1$ for every i, i.e. $0 < \eta < 2/\lambda_{\max}$. The slowest-shrinking coordinate governs the overall rate, and it shrinks like $(1 - \eta\lambda_{\min})^k$. The ratio $\kappa = \lambda_{\max}/\lambda_{\min}$, the *condition number*, thus controls how hard the problem is: a large κ (a long,

narrow valley) forces a small η to avoid divergence along the steep direction, which makes progress along the shallow direction agonizingly slow. This is exactly the difficulty an *ill-conditioned* inverse problem presents, and it foreshadows the identifiability ridge of Chapter 11.

Remark 4.4 (Convergence rate, quantified*)**.** The example gives the rate explicitly. The error along the worst direction contracts by a factor $1 - \eta\lambda_{\min}$ per step; with the best stable choice of η this works out to a contraction governed by $(\kappa - 1)/(\kappa + 1)$ per step. When κ is close to one, convergence is fast (the factor is near zero); when κ is large, the factor is near one and convergence is slow, requiring on the order of κ iterations to make definite progress. The practical moral is that *conditioning, not dimension, is what makes gradient descent slow*, and that reshaping the problem to improve its conditioning — by rescaling parameters, or by using curvature as Newton-type methods do — is often more valuable than tuning the learning rate.

4.4 Line search and step-size selection

Rather than fix the learning rate in advance, one can choose it adaptively at each step by asking how far to go in the descent direction — a *line search*. Given the current point θ_k and direction $d_k = -\nabla L(\theta_k)$, a line search seeks a step size η_k that decreases the loss "sufficiently," for instance by approximately minimizing $L(\theta_k + \eta d_k)$ over η, or by accepting any η that achieves a fraction of the decrease the gradient predicts (the *Armijo* condition). Line search frees the user from guessing a single global learning rate and makes plain gradient descent considerably more robust. The cost is extra evaluations of the loss at each iteration. For the small problems of Part II this cost is negligible, and the Levenberg–Marquardt method of the next chapter can be understood as a sophisticated, curvature-aware relative of line search, in which the step's length *and* direction are both adapted.

4.5 Step size, conditioning, and the shape of the valley

Example 4.3 carries a general lesson. The local behavior of any smooth loss near a minimum is, by Taylor's theorem (2.2), approximately the quadratic bowl $\frac{1}{2}(\theta - \theta^\star)^\top \nabla^2 L(\theta^\star)(\theta - \theta^\star)$. The eigenvalues of the Hessian $\nabla^2 L(\theta^\star)$ are the curvatures of the bowl along its principal directions. When they are comparable the bowl is round and gradient descent works well; when they differ by orders of magnitude the bowl is a steep-walled canyon, and plain gradient descent zigzags slowly down its floor.

Two broad responses exist. One can *precondition* — rescale the parameters so the bowl becomes rounder — which in the extreme leads to Newton's method, dividing by the curvature (Chapter 5). Or one can *augment the step* with information from past steps, as momentum methods do (Chapter 6). Both are attempts to cope with badly shaped valleys, and a recurring sign of a hard inverse problem is precisely a loss surface with a long, flat-floored valley along some parameter combination.

4.6 Newton's method, in brief

The most direct way to use curvature is *Newton's method*. Approximating L near θ_k by its second-order Taylor expansion (2.2) and minimizing that quadratic exactly gives the update

$$\theta_{k+1} = \theta_k - \big[\nabla^2 L(\theta_k)\big]^{-1} \nabla L(\theta_k).$$

Where gradient descent steps along the gradient, Newton's method steps along the gradient *corrected by the inverse Hessian*, which rescales each direction by its curvature and so accounts for the shape

of the valley. Near a minimum Newton's method converges extremely fast (the error roughly squares each step). Its drawbacks are that it requires the Hessian — expensive to form and store for large models — and that far from a minimum the Hessian may not be positive definite, so the "step" may point uphill. The Levenberg–Marquardt method of Chapter 5 is precisely a practical repair of Newton's method for least-squares problems, using an approximation to the Hessian and damping it to stay well behaved far from the solution.

4.7 Stopping and the limits of optimization

When do we stop iterating? Common criteria are a small gradient ($\|\nabla L(\theta_k)\|$ below a tolerance, signalling near-stationarity), a small change in parameters or loss between iterations, or simply a fixed budget of iterations. In the noisy, small-data setting it is wise to remember that the minimizer of the training loss is not the truth: driving the gradient to machine precision buys nothing if the loss itself is built from five noisy points. Optimization finds the bottom of the loss surface we hand it; whether that bottom corresponds to the right parameters is a separate question, answered not by the optimizer but by the analysis of Chapters 11 and 15.

Remark 4.5 (Optimization error versus statistical error)**.** It is useful to separate two kinds of error. *Optimization error* is the gap between the parameters we found and the true minimizer of the loss; better algorithms or more iterations reduce it. *Statistical error* is the gap between the loss's minimizer and the parameters that generated the data; it comes from noise and sparsity and cannot be reduced by any amount of optimization. A common confusion in practice is to attack statistical error with optimization effort — to run the optimizer longer in the hope of a better answer — when the limiting factor is the data. We will see a clean example in Chapter 11. The two errors call for entirely different remedies: optimization error for a better algorithm or more iterations, statistical error for more data, better experimental design, or stronger prior assumptions.

Exercises

Exercise 4.1 (*Gradient descent by hand*). Let $L(\theta) = (\theta - 5)^2$, a one-dimensional loss. Write the gradient-descent update with learning rate η, and starting from $\theta_0 = 0$ compute $\theta_1, \theta_2, \theta_3$ for $\eta = 0.1$. To what value does the iteration converge?

Exercise 4.2 (*Divergence from a large step*). For the same $L(\theta) = (\theta - 5)^2$, find the range of learning rates η for which gradient descent converges. What happens exactly at the boundary of this range, and beyond it? *(Hint: the update is linear in $\theta_k - 5$.)*

Exercise 4.3 (*Conditioning*). Consider $L(\theta_1, \theta_2) = \theta_1^2 + 100\,\theta_2^2$. Compute the gradient and the Hessian. What is the condition number? Explain why gradient descent with a single learning rate must be slow on this function, and which coordinate forces the learning rate to be small.

Exercise 4.4 (*Convex or not*). Determine whether each function is convex on its natural domain, using the Hessian test where convenient: (a) e^{θ}; (b) θ^4; (c) $\sin\theta$; (d) $\|\theta\|^2$ on $\mathbb{R}^p$; (e) $\theta_1^2 - \theta_2^2$. Briefly justify.

Exercise 4.5 (*Stationary but not a minimum*). Give a function of two variables with a stationary point that is a saddle, not a minimum, and identify the saddle. Why might gradient descent still escape a saddle in practice, whereas it cannot escape a local minimum?

Exercise 4.6 (*A Newton step*). For $L(\theta) = \theta^4$ (one variable), write the Newton update and take one step from $\theta_0 = 1$. Compare the result to one gradient-descent step with $\eta = 0.1$. *(Note that L'' varies, which is exactly the curvature information Newton's method uses.)*

Exercise 4.7 (*Line search idea*). Explain in words what a line search does and why it makes plain gradient descent more robust. What is the cost, and why is that cost acceptable for small problems but potentially prohibitive for very large ones?

Exercise 4.8 (*Optimization versus statistical error*). In your own words, explain the difference between optimization error and statistical error for the Gompertz fitting problem. If you fit the model and are unhappy with the result, what diagnostic would tell you which kind of error dominates?

Exercise 4.9 (*Conditioning beats dimension*). Two quadratic problems both live in $\mathbb{R}^2$. The first has Hessian eigenvalues 1 and 1; the second has 1 and 10^4. Which is harder for gradient descent, and by roughly what factor in the number of iterations? Why does this show that conditioning, not the number of parameters, governs the difficulty here?

Chapter 5

Least Squares and the Levenberg–Marquardt Method

Least squares is the oldest and still the most important method for fitting a model to data. This chapter develops it in two stages: the *linear* case, where the answer is a clean formula, and the *nonlinear* case — which the Gompertz problem requires — where we iterate. The nonlinear method we build, Levenberg–Marquardt, is the classical workhorse of curve fitting and the baseline against which the neural approaches of later chapters will be measured. It is precisely the algorithm behind the `lsqnonlin`-style routines used in the running research project.

5.1 Linear least squares

Suppose the model is *linear in its parameters*: the prediction at input x_i is a linear combination

$$m(x_i;\theta) = \sum_{j=1}^{p} \theta_j \, \phi_j(x_i),$$

where the ϕ_j are fixed functions ("features") of the input. A straight line uses $\phi_1 = 1, \phi_2 = x$; a cubic uses $1, x, x^2, x^3$. Crucially, the ϕ_j may be nonlinear in x; what matters is linearity in θ. Stacking the predictions gives a matrix equation: with Φ the $N \times p$ matrix whose (i,j) entry is $\phi_j(x_i)$ (the *design matrix*), the prediction vector is $\Phi\theta$, and the least-squares loss is

$$L(\theta) = \|\Phi\theta - y\|^2 .$$

This loss is a convex quadratic in θ, so its unique minimizer is found by setting the gradient to zero. Differentiating,

$$\nabla L(\theta) = 2\Phi^\top(\Phi\theta - y) = 0,$$

which rearranges to the *normal equations*

$$\Phi^\top\Phi\,\theta = \Phi^\top y, \qquad \Longrightarrow \qquad \hat{\theta} = (\Phi^\top\Phi)^{-1}\Phi^\top y, \tag{5.1}$$

valid whenever $\Phi^\top\Phi$ is invertible. This formula is the engine of linear regression. Example 2.2 was the $p = 2$ case written out by hand.

A geometric reading. The normal equations have an illuminating geometry. The achievable predictions $\Phi\theta$ form a p-dimensional subspace of $\mathbb{R}^N$ — the range of Φ. Least squares seeks the point in this subspace closest to the data vector y, which is the *orthogonal projection* of y onto the subspace. The condition $\Phi^\top(\Phi\theta - y) = 0$ says exactly that the residual $\Phi\theta - y$ is orthogonal to every column of Φ, i.e. to the subspace — the defining property of an orthogonal projection. This is why least squares is so natural: among all predictions the model can make, it chooses the one geometrically nearest the data.

Remark 5.1 (Conditioning of the normal equations*)**.** The matrix $\Phi^\top\Phi$ can be ill-conditioned — nearly singular — when the features are nearly linearly dependent, for instance when fitting a high-degree polynomial (the monomials x^k become nearly parallel on a bounded interval). Then the inverse in (5.1) amplifies noise in y, and small data perturbations cause large parameter swings. This is the linear shadow of the identifiability problems we study in Chapter 11. In practice one never forms the inverse explicitly; stable factorizations — the QR decomposition, or the *singular value decomposition* (SVD) — solve the least-squares problem with far better numerical behavior. The SVD additionally exposes the ill-conditioning directly, through small singular values, and underlies regularization methods that we revisit in Chapter 11.

5.2 Why squares? The Gaussian connection

Why minimize the sum of *squared* residuals rather than, say, the sum of absolute values? One answer is mathematical convenience: squares give a differentiable, convex loss with the closed form (5.1). A deeper answer comes from probability. Suppose the data are generated by the model plus independent Gaussian noise,

$$y_i = m(x_i;\theta) + \varepsilon_i, \qquad \varepsilon_i \sim \mathcal{N}(0,\sigma^2).$$

The *likelihood* of the data under parameters θ is the product of Gaussian densities, and its logarithm is

$$\log p(y \mid \theta) = -\frac{1}{2\sigma^2}\sum_{i=1}^{N}\big(y_i - m(x_i;\theta)\big)^2 + \text{const.}$$

Maximizing the likelihood over θ is therefore *identical* to minimizing the sum of squared residuals. Least squares is maximum-likelihood estimation under Gaussian noise. This is worth remembering for two reasons: it tells us when least squares is the statistically natural choice (additive Gaussian noise), and it warns us when it is not — for instance under the multiplicative noise (2.3) of the running example, where a weighted version is strictly more appropriate, though the unweighted form is common and usually adequate.

Maximum likelihood as a unifying principle. The likelihood viewpoint extends well beyond least squares and is worth stating in general, because it recurs implicitly throughout the course. Given a probabilistic model of how the data arise — a rule for the probability $p(y \mid \theta)$ of observing the data given the parameters — the *maximum-likelihood estimate* is the θ that makes the observed data most probable. Different noise assumptions give different losses: Gaussian noise gives squared error, as we have seen; heavier-tailed noise gives a more robust loss that down-weights outliers; and for count or categorical data, entirely different likelihoods give the losses used in those settings. When we later write down a loss without derivation, it is usually a maximum-likelihood loss in disguise, and asking "what noise model does this loss assume?" is a sharp diagnostic question.

5.3 Nonlinear least squares

The Gompertz model is *not* linear in its parameters: the prediction $V(t_i; a, K)$ from (2.5) involves a and K inside an exponential. There is no formula like (5.1). We must minimize

$$L(\theta) = \sum_{i=1}^{N} r_i(\theta)^2, \qquad r_i(\theta) = m(t_i; \theta) - y_i, \tag{5.2}$$

iteratively. The structure that makes this tractable — and distinguishes it from general optimization — is that the loss is a *sum of squares* of residuals, and we can exploit that structure.

The gradient of (5.2) is, by the chain rule,

$$\nabla L(\theta) = 2J(\theta)^\top r(\theta),$$

where $r(\theta) = (r_1, \ldots, r_N)$ is the residual vector and $J(\theta)$ is the *Jacobian*, the $N \times p$ matrix of partial derivatives $J_{ij} = \partial r_i / \partial \theta_j$. The Hessian is

$$\nabla^2 L(\theta) = 2J^\top J + 2\sum_i r_i \nabla^2 r_i.$$

The first term, $2J^\top J$, uses only first derivatives of the residuals; the second involves their second derivatives and the residuals themselves. This decomposition is the key structural fact of nonlinear least squares, and the basis of the Gauss–Newton approximation below.

5.4 Gauss–Newton and Levenberg–Marquardt

The *Gauss–Newton* method makes a brilliant simplification: near a good fit the residuals r_i are small, so the second term in the Hessian is negligible, and we approximate

$$\nabla^2 L \approx 2J^\top J.$$

Applying Newton's method (Section 4.6) with this approximation gives the Gauss–Newton update

$$\theta_{k+1} = \theta_k - (J^\top J)^{-1} J^\top r,$$

all quantities evaluated at θ_k. The beauty of the approximation is that $J^\top J$ requires only first derivatives, which are far cheaper than the full Hessian, yet near the solution it captures the curvature well enough to inherit much of Newton's speed. The method converges quickly near a solution but can behave badly far from one, where $J^\top J$ may be nearly singular and the step wildly large.

The *Levenberg–Marquardt* (LM) method repairs this by interpolating between Gauss–Newton and gradient descent. It adds a multiple of the identity to the approximate Hessian:

$$\theta_{k+1} = \theta_k - (J^\top J + \mu I)^{-1} J^\top r. \tag{5.3}$$

The *damping parameter* $\mu \geq 0$ controls the blend. When μ is small, (5.3) is essentially Gauss–Newton, taking large, curvature-aware steps. When μ is large, the μI term dominates and the update reduces to a small gradient-descent step, $\theta_{k+1} \approx \theta_k - \mu^{-1} J^\top r$. The algorithm adjusts μ automatically: if a step decreases the loss, μ is reduced (trust the curvature more); if a step fails, μ is increased (retreat toward safe gradient descent). This adaptive damping makes LM both fast and robust, which is why it is the default for nonlinear curve fitting. One can read the μI term as a "trust region": it limits the step to a neighborhood where the local quadratic model is reliable, expanding the region when the model proves accurate and shrinking it when it does not.

5.5 Multiple starts and local minima

Because the nonlinear loss (5.2) is nonconvex, LM converges to a *local* minimum that depends on the starting point θ_0. For the Gompertz problem with very sparse data this is a genuine hazard: from a poor initial guess the optimizer may land in a spurious basin or fail to converge. The standard defense is *multi-start*: run LM from many random initial guesses and keep the result with the smallest loss. In the research project behind these notes, single-start LM failed to find the global optimum in a substantial fraction of trials at the sparsest setting ($N = 3$), while a thirty-start scheme reduced such failures to a negligible level. Multi-start costs more computation but is essential for trustworthy fits when data is scarce.

Good initialization can reduce the number of starts needed. For the Gompertz model, sensible initial guesses are available from the data itself: the carrying capacity K can be initialized near the largest observed volume, and the rate a from a rough estimate of how quickly the early points rise. Initializing in the logarithmic parameters $\log a$ and $\log K$ — which keeps them positive and spans their plausible ranges more evenly — further improves robustness, the same reparametrization that helps the physics-informed network of Chapter 10.

Remark 5.2 (What LM does and does not guarantee)**.** Levenberg–Marquardt, even with multi-start, solves the *optimization* problem: it finds parameters minimizing the training loss. It says nothing about whether those parameters are correct. If the model is misspecified, or the data too sparse to identify the parameters, LM will nonetheless return a confident-looking fit — a precise minimizer of the wrong or under-determined loss. Separating this optimization success from the statistical question of correctness is the entire burden of Chapter 11.

5.6 Application to the running example

For the Gompertz fitting problem (3.3) the residuals are $r_i(a, K) = V(t_i; a, K) - y_i$, and the Jacobian has columns $\partial V/\partial a$ and $\partial V/\partial K$, computable by differentiating the closed-form solution (2.5) or, when no formula is available, by differentiating the numerical solver (a topic we revisit under automatic differentiation in Chapter 8). Running multi-start LM on a sparse noisy dataset yields parameter estimates $\hat{a}, \hat{K}$ — together with, as we will see, considerable uncertainty in those estimates when the data are concentrated near the carrying capacity. The LM fit is our baseline: every more elaborate method in the course must justify itself against what this classical algorithm already achieves.

It is worth previewing how the baseline behaves, because it sets expectations for the chapters to come. On data generated by the Gompertz model itself, multi-start LM recovers the parameters well when the data are reasonably informative, and its estimates degrade gracefully as noise rises and data grow sparse — with the carrying capacity typically better determined than the rate, for reasons the identifiability analysis of Chapter 11 will make precise. On data generated by a *different* law, LM still returns a clean-looking Gompertz fit, and therein lies the danger that motivates much of the rest of the book.

Exercises

Exercise 5.1 (*Normal equations in practice*). Set up the matrix Φ for fitting a quadratic $\theta_1 + \theta_2 x + \theta_3 x^2$ to the four points $x = (-1, 0, 1, 2)$. You need not solve; just write Φ and the system $\Phi^\top \Phi\, \theta = \Phi^\top y$.

Exercise 5.2 (*The projection picture*). Explain, in terms of orthogonal projection, why the least-squares residual is orthogonal to every column of the design matrix Φ. What does this say geometrically about the relationship between the data vector y, the fitted prediction $\Phi\hat{\theta}$, and the residual?

Exercise 5.3 (*Maximum likelihood*). Starting from the Gaussian likelihood, show in detail that maximizing $\log p(y \mid \theta)$ is equivalent to minimizing $\sum_i (y_i - m(x_i;\theta))^2$. Where does the noise variance σ^2 go, and why does it not affect the minimizer?

Exercise 5.4 (*Weighted least squares*). Under multiplicative noise (2.3), the variance of y_i is $\sigma^2 V(t_i)^2$, which differs across points. Argue that the statistically natural loss weights each squared residual by $1/V(t_i)^2$, and write the resulting weighted objective. *(This is why fitting raw squared residuals is slightly suboptimal here, though often used anyway.)*

Exercise 5.5 (*What loss is this?*). A colleague fits a model by minimizing $\sum_i |y_i - m(x_i;\theta)|$ instead of squared error. What noise model does this loss correspond to under the maximum-likelihood viewpoint, and in what situation would it be the wiser choice?

Exercise 5.6 (*Gauss–Newton step*). For a model with a single parameter θ and residuals $r_i(\theta)$, write out the Gauss–Newton update explicitly in terms of r_i and $r_i' = dr_i/d\theta$. Show it reduces to a recognizable form when the model is linear, $m(x_i;\theta) = \theta\, x_i$.

Exercise 5.7 (*The role of damping*). In the LM update (5.3), describe the limiting behavior of the step as $\mu \to 0$ and as $\mu \to \infty$. Why is it advantageous to increase μ after a step that fails to reduce the loss? Relate the damping to the idea of a trust region.

Exercise 5.8 (*Why multi-start*). Explain, in terms of the nonconvex loss surface, why running LM from a single starting point can give a poor result for very sparse data, and why trying many starting points helps. Would multi-start help if the problem were convex? Why or why not?

Exercise 5.9 (*Initializing from data*). For the Gompertz model, propose data-driven initial guesses for a and K given a set of observations. Why might initializing in $\log a$ and $\log K$ be more robust than in a and K directly?

Exercise 5.10 (*Gauss–Newton ignores a term*). The full Hessian of the least-squares loss is $2J^\top J + 2\sum_i r_i \nabla^2 r_i$. Explain why dropping the second term is reasonable near a good fit but potentially harmful far from one, and how the LM damping compensates for the resulting inaccuracy.

Chapter 6

Stochastic Optimization and Adam

The methods of the previous chapter compute the full gradient or Jacobian at every step, using all the data at once. When the data is small this is fine. But neural networks are trained on large datasets and have many parameters, and computing the exact gradient at every step becomes too expensive. The remedy — estimating the gradient from a random subset of the data — gives *stochastic gradient descent* and its descendants, of which the Adam optimizer is the one most used to train the networks of Part III. This chapter explains the idea and the optimizer, and notes why the stochastic machinery, though essential at scale, is not the bottleneck in the small-data scientific problems at the heart of these notes.

6.1 From full-batch to stochastic gradients

Most machine-learning losses are *averages* over data points,

$$L(\theta) = \frac{1}{N}\sum_{i=1}^{N} \ell_i(\theta),$$

where ℓ_i is the loss on the ith example. The exact gradient is then also an average, $\nabla L = \frac{1}{N}\sum_i \nabla \ell_i$, costing N gradient evaluations per step. The key observation is that we do not need the exact gradient to make progress — a noisy but unbiased *estimate* suffices. Drawing a random subset $B \subset \{1, \dots, N\}$, called a *minibatch*, the average over the batch,

$$\widehat{\nabla L}(\theta) = \frac{1}{|B|}\sum_{i \in B} \nabla \ell_i(\theta),$$

is an unbiased estimate of ∇L that costs only $|B|$ evaluations. The resulting *stochastic gradient descent* (SGD) update is

$$\theta_{k+1} = \theta_k - \eta\, \widehat{\nabla L}(\theta_k), \tag{6.1}$$

identical to (4.1) but with the batch-estimated gradient. A full pass through the data is called an *epoch*; training proceeds for many epochs, each visiting the data in fresh random batches.

The stochasticity is a mixed blessing. On one hand, the gradient estimate is noisy, so the iterates jitter rather than descend smoothly. On the other hand, the noise is computationally cheap and can even help escape shallow local minima and saddle points — the jitter knocks the iterate off a saddle it would otherwise stall near. The step size η typically must decrease over training (a *schedule*) so that the jitter shrinks as the iterates approach a minimum.

The economics of the minibatch. Why does this trade pay off? A full-batch gradient is N times more expensive than a single-example gradient but is not N times more useful: much of the information in the full gradient is redundant, because data points resemble one another. A modest batch captures most of the gradient's direction at a small fraction of the cost, so many cheap, slightly noisy steps make far more progress per unit of computation than few expensive, exact ones. This is the central economic argument for stochastic methods, and it explains why batch sizes in practice are large enough to be efficient on the hardware but far smaller than the full dataset.

6.2 Learning-rate schedules

The requirement that the step size shrink over training deserves elaboration, because it is where much practical tuning lives. With a *constant* step size, the gradient noise keeps the iterate bouncing within a region around the minimum whose size is set by η and the noise level; it never settles. A *decreasing* schedule shrinks this region toward zero. Classical theory asks that the step sizes sum to infinity (so the iterate can travel any distance) while their squares sum to a finite value (so the accumulated noise is controlled); schedules like $\eta_k \propto 1/k$ satisfy this. In modern practice one often uses gentler schedules — holding η constant for a while and then decreasing it in stages, or decaying it smoothly — chosen by experiment. The qualitative lesson is robust: too slow a decay leaves the iterate rattling around the minimum, too fast a decay stalls it before it arrives.

6.3 Momentum

Plain SGD struggles in the long, narrow valleys diagnosed in Section 4.3: it oscillates across the steep walls while creeping along the shallow floor. *Momentum* damps the oscillation by averaging gradients over time. Introducing a velocity vector v,

$$v_{k+1} = \beta v_k + \widehat{\nabla L}(\theta_k), \qquad \theta_{k+1} = \theta_k - \eta\, v_{k+1},$$

with $\beta \in [0, 1)$ a decay factor (commonly 0.9). The velocity accumulates consistent gradient directions and cancels oscillating ones, so the iterate gathers speed along the valley floor while its cross-valley swings are suppressed. The mental image is a heavy ball rolling downhill, its inertia smoothing out the bumps. A refinement, *Nesterov momentum*, evaluates the gradient at a look-ahead point (where the velocity is about to carry the iterate) rather than at the current point, which gives a modest but real improvement in convergence and is widely used.

6.4 Adaptive learning rates and Adam

A single global learning rate is a blunt instrument when different parameters need different step sizes — exactly the conditioning problem of Section 4.3. *Adaptive* methods give each parameter its own effective learning rate, scaled by the recent magnitude of its gradient. The *Adam* optimizer combines this idea with momentum and is the de facto standard for training neural networks.

Adam maintains two running averages for each parameter: an estimate m of the mean gradient (the "first moment," i.e. momentum) and an estimate v of the mean squared gradient (the "second moment"). With decay rates β_1, β_2 (typically 0.9 and 0.999),

$$m_{k+1} = \beta_1 m_k + (1 - \beta_1)\, g_k, \qquad v_{k+1} = \beta_2 v_k + (1 - \beta_2)\, g_k^2,$$

where $g_k = \widehat{\nabla L}(\theta_k)$ and the square is elementwise. Because m and v start at zero, they are biased toward zero in the early steps; Adam corrects this with the *bias correction*

$$\hat{m}_{k+1} = \frac{m_{k+1}}{1-\beta_1^{k+1}}, \qquad \hat{v}_{k+1} = \frac{v_{k+1}}{1-\beta_2^{k+1}},$$

which inflates the early estimates to compensate (the denominators are near zero at first and approach one as k grows). The update is then

$$\theta_{k+1} = \theta_k - \eta \, \frac{\hat{m}_{k+1}}{\sqrt{\hat{v}_{k+1}} + \epsilon}, \tag{6.2}$$

elementwise, with ϵ a tiny constant preventing division by zero. The effect of dividing by $\sqrt{\hat{v}}$ is that parameters with large, volatile gradients take smaller steps, and parameters with small, steady gradients take larger ones — an automatic, per-parameter rescaling that makes Adam relatively insensitive to the global choice of η and robust on the badly conditioned losses neural networks present. One can read the second-moment rescaling as a cheap, diagonal approximation to the preconditioning that Newton's method (Section 4.6) performs with the full Hessian: instead of dividing by curvature, Adam divides by the observed gradient scale, which is far cheaper and often nearly as effective.

Remark 6.1 (Why Adam matters here)**.** We introduce Adam because it is the optimizer used to train the neural networks of Chapters 7–10, including the physics-informed networks at the center of the applied story. When such a network is trained, (6.2) is what updates both the network weights and any physical parameters (such as a and K) that are being learned alongside them. The research project behind these notes ran into a characteristic Adam-tuning issue — the interaction between the learning rate for the physical parameters and the weighting of the physics term in the loss — which we dissect in Chapter 10. The per-parameter rescaling of Adam interacts subtly with the scaling of the loss terms, and understanding that interaction is part of using the method well.

6.5 Stochasticity is not the scientific bottleneck

It is worth a word of perspective. The stochastic machinery of this chapter exists to cope with *large* data and *many* parameters. The scientific inverse problems at the heart of these notes have the opposite character: a handful of data points and a handful of parameters. There, the full gradient is cheap, classical methods like Levenberg–Marquardt (Chapter 5) are preferable for fitting the mechanistic parameters, and the limiting factor is not optimization speed but the *information content of the data*. We will see that throwing more sophisticated optimization at a sparse, ill-posed inverse problem does not rescue it; the difficulty lies in the problem, not the optimizer. Adam earns its place in the course specifically when a flexible neural network enters the picture, not as a universal upgrade over the classical methods.

This is a point worth holding onto against the cultural momentum of the field, which tends to equate "more advanced optimizer" with "better results." For a two-parameter mechanistic fit to five points, Adam offers nothing over multi-start Levenberg–Marquardt and several disadvantages: it needs a learning rate and a schedule, it converges more slowly on small smooth problems, and its stochasticity is pointless when the full gradient is cheap. Choosing the right tool for the problem's scale is itself a skill, and the right tool here is usually the classical one.

Exercises

Exercise 6.1 (*Unbiasedness of the batch gradient*). Show that if the minibatch B is drawn uniformly at random, then the batch-estimated gradient $\widehat{\nabla L}$ has expectation equal to the full gradient ∇L. Why is unbiasedness the property that makes SGD work?

Exercise 6.2 (*Batch size and noise*). Qualitatively, how does the variance of the batch gradient estimate change as the batch size $|B|$ increases? Using the variance facts from Section 2.4, explain why the standard deviation of the estimate falls like $1/\sqrt{|B|}$, and discuss the tradeoff between large and small batches.

Exercise 6.3 (*Momentum on a valley*). Explain in words why momentum accelerates progress along the floor of a long narrow valley while suppressing oscillation across it. Relate this to the eigen-decomposition picture of Example 4.3.

Exercise 6.4 (*Adam's rescaling*). Two parameters have gradients that, over recent steps, are roughly $(10, 10, 10, \dots)$ and $(0.1, -0.1, 0.1, \dots)$ respectively. Describe qualitatively how Adam's division by $\sqrt{\hat{v}}$ treats the two, and why this is sensible.

Exercise 6.5 (*Bias correction*). Explain why the running averages m and v in Adam are biased toward zero in the early steps, and how the bias-correction factors $1/(1-\beta^k)$ compensate. What happens to these factors as k grows large?

Exercise 6.6 (*When not to use Adam*). For fitting the two Gompertz parameters to five data points, would you reach for Adam or Levenberg–Marquardt? Justify your answer in terms of data size, number of parameters, and what limits the quality of the fit.

Exercise 6.7 (*Learning-rate schedule*). Explain why a constant step size leaves SGD jittering around a minimum rather than settling into it, and why decreasing the step size over training addresses this. What goes wrong if the step size is decreased too quickly?

Exercise 6.8 (*Adam as cheap preconditioning*). Explain the sense in which Adam's division by $\sqrt{\hat{v}}$ approximates the preconditioning that Newton's method performs with the Hessian. Why is the Adam version far cheaper, and in what way is it cruder?

Exercise 6.9 (*Epochs versus steps*). A dataset has $N = 10{,}000$ examples and is trained with batch size 100. How many parameter updates occur in one epoch? If training runs for 50 epochs, how many total updates is that? Why is the number of *updates*, not epochs, what governs progress?

Part III

Neural Networks

Chapter 7

Neural Networks from Scratch

We now meet the central object of modern machine learning: the neural network. Our treatment is deliberately elementary and mathematical. A neural network, stripped of mystique, is simply a flexible parametric function built by composing linear maps with simple nonlinearities. This chapter defines that function, explains why composing it gives enormous flexibility, and states (without proof) the approximation theorem that justifies its use. The next chapter explains how to differentiate it; the one after, how to train it.

7.1 The single neuron

The building block is the *neuron*, a function that takes an input vector $x \in \mathbb{R}^n$, forms a weighted sum, adds a constant, and passes the result through a nonlinear function:

$$z = w^\top x + b, \qquad \text{output} = \phi(z).$$

Here $w \in \mathbb{R}^n$ is the vector of *weights*, $b \in \mathbb{R}$ is the *bias*, and $\phi : \mathbb{R} \to \mathbb{R}$ is the *activation function*. The weighted sum $w^\top x + b$ is an affine map — a linear transformation plus a shift — and the activation ϕ introduces the nonlinearity without which a network could represent only linear functions.

The neuron is loosely inspired by biology — the weighted sum as the integration of incoming signals, the activation as a firing threshold — but the analogy is best held lightly. For our purposes the neuron is simply a parametrized scalar function, and a network is what we get by composing many of them. The biological story explains the name, not the mathematics.

7.2 Activation functions

The activation function must be nonlinear, and for the methods of later chapters it should be smooth (differentiable). Common choices include:

- the *logistic sigmoid* $\phi(z) = 1/(1 + e^{-z})$, which squashes its input into $(0, 1)$;
- the *hyperbolic tangent* $\phi(z) = \tanh(z)$, which squashes into $(-1, 1)$ and is smooth with simple derivatives;
- the *rectified linear unit* (ReLU) $\phi(z) = \max(0, z)$, which is cheap and effective but not differentiable at the origin;
- the *softplus* $\phi(z) = \ln(1+e^z)$, a smooth approximation to ReLU that is everywhere differentiable and *positive*.

The choice matters for the running example. In a physics-informed network for tumor growth (Chapter 10) the output must represent a volume, which is necessarily positive, and the dynamics involve $\ln(K/V)$, which demands $V > 0$. A *softplus* output activation enforces positivity automatically and sidesteps the logarithmic singularity at $V = 0$ — a small but consequential design choice that recurs in the project. Smooth activations such as tanh in the hidden layers are likewise preferred there because the method requires differentiating the network.

Why smoothness matters here. The preference for smooth activations is not universal — ReLU dominates much of modern deep learning precisely because it is cheap and trains well in very deep networks — but it is sharp in our setting. A physics-informed network must compute the derivative of its own output with respect to time (Chapter 10), and that derivative is taken by automatic differentiation through the activations. A kink in the activation (as in ReLU) produces a discontinuous derivative, which makes the physics residual behave badly. Smooth activations like tanh give smooth derivatives, which the differential-equation constraint requires. The choice of activation, often treated as a minor default, is here dictated by the mathematics of the method.

7.3 Layers and the multilayer perceptron

A single neuron is weak. Power comes from arranging many neurons in *layers* and stacking the layers. A *layer* of m neurons, each seeing the same input $x \in \mathbb{R}^n$, produces an output in $\mathbb{R}^m$:

$$h = \phi(Wx + b),$$

where now W is an $m \times n$ weight matrix, $b \in \mathbb{R}^m$ a bias vector, and ϕ is applied elementwise. This is exactly the matrix–vector product of Section 2.2 wrapped in a nonlinearity, and the dimension bookkeeping — the output of one layer must match the input width of the next — is the discipline mentioned there.

Stacking L such layers gives a *multilayer perceptron* (MLP), or *feedforward network*. With input $x = h^{(0)}$, the network computes

$$h^{(\ell)} = \phi\big(W^{(\ell)} h^{(\ell-1)} + b^{(\ell)}\big), \qquad \ell = 1, \dots, L, \tag{7.1}$$

and the final layer $h^{(L)}$ (often with the activation omitted) is the network's output. The intermediate layers are *hidden*; their widths are design choices. The collection of all weight matrices and bias vectors,

$$\theta = \big\{W^{(\ell)}, b^{(\ell)} : \ell = 1, \dots, L\big\},$$

constitutes the network's parameters — often thousands or millions of them, all to be learned from data. Note that the network is a function $x \mapsto h^{(L)}(x; \theta)$ that is nonlinear in x *and* nonlinear in θ; this double nonlinearity is the source of both its expressive power and the nonconvexity of its training problem.

Example 7.1 (A forward pass through a tiny network)**.** Consider a network with one input, a single hidden layer of two tanh neurons, and a linear output. Let the hidden weights and biases be $W^{(1)} = \left(\begin{smallmatrix} 1 \\ -2 \end{smallmatrix}\right)$, $b^{(1)} = \left(\begin{smallmatrix} 0 \\ 1 \end{smallmatrix}\right)$, and the output weights $W^{(2)} = \left(\begin{smallmatrix} 1 & 1 \end{smallmatrix}\right)$, $b^{(2)} = 0$. On input $x = 1$ the hidden pre-activations are $z = (1, -1)$, the hidden activations are $(\tanh 1, \tanh(-1)) \approx (0.762, -0.762)$, and the output is their sum, ≈ 0. The computation is nothing more than matrix multiplications interleaved with the elementwise tanh — and it is exactly this chain of operations that automatic differentiation will traverse in the next chapter to compute gradients.

7.4 Why depth and width give flexibility

A useful way to see the flexibility of (7.1) is to track what each layer does geometrically. The affine map $Wh + b$ stretches, rotates, and shifts space; the activation ϕ bends it. Composing many such operations folds the input space in increasingly intricate ways, so that by the output layer the network can represent functions of great complexity. A network with one hidden layer can already approximate any continuous function on a bounded region, given enough neurons; deeper networks can represent some functions far more efficiently (with fewer total neurons) than shallow ones, which is part of why depth is useful in practice.

It helps to think of the hidden layer as building a set of *basis functions* adapted to the data. Each hidden neuron contributes one bump or ramp — a tanh or ReLU shape positioned and scaled by its weights and bias — and the output layer takes a weighted combination of these. With enough such adaptive pieces, almost any shape can be assembled. This is the same idea as fitting with a fixed basis (polynomials, say), except that the network *learns* the basis functions rather than fixing them in advance, which is what gives it its flexibility and, simultaneously, its capacity to overfit.

Theorem 7.2 (Universal approximation, informal)**.** *Let f be a continuous function on a closed bounded subset of $\mathbb{R}^n$. For any desired accuracy $\varepsilon > 0$, there is a single-hidden-layer network with a nonpolynomial activation and enough hidden neurons whose output approximates f to within ε everywhere on that set.*

We state this without proof. Two cautions temper it. First, it is an *existence* result: it guarantees that suitable weights exist, not that our training procedure will find them. Second, "enough neurons" may be a very large number, and a network large enough to approximate anything is also flexible enough to *overfit* anything (Section 3.4). Universal approximation explains why neural networks *can* work; it does not promise that they will, and it is silent on the central scientific worry — that a network may fit sparse data while extrapolating nonsensically.

Remark 7.3 (Approximation is not extrapolation)**.** The universal approximation theorem concerns accuracy *on a bounded set* — typically the region where data live. It says nothing about behavior outside that region. A network fit to tumor data over the first twenty days may approximate the trajectory beautifully there and yet predict anything at all on day forty, because the theorem offers no control beyond the training region. This is the formal counterpart of a practical fact we will see repeatedly: flexible models interpolate well and extrapolate unpredictably, which is exactly why forecasting — extrapolation into the future — is the demanding test that exposes them.

7.5 Networks as models in the inverse-problem framework

It is worth placing the neural network back in the triple of Chapter 1: model, data, fitting procedure. The network is a *model* — an extremely flexible, phenomenological one (Section 3), making no commitment to any mechanism. Its parameters θ are fit to data by minimizing a loss, just as for any other model; because θ is high-dimensional and the loss nonconvex, the fitting uses the stochastic, gradient-based optimizers of Chapter 6, and the gradients are supplied by the automatic differentiation of Chapter 8. What makes a network special is only its flexibility, and that flexibility is exactly what must be disciplined — by held-out validation, by regularization, or, in the scientific setting, by building in physical law (Chapter 10) — if it is to produce trustworthy models rather than elaborate overfits.

This placement also clarifies what a network is *not*. It is not a mechanistic model: its weights carry no scientific interpretation, and one cannot read a growth rate or a carrying capacity off a

trained network as one can off a fitted Gompertz model. When the scientific goal is to estimate interpretable parameters — as in the running example — a pure network is the wrong tool, and the physics-informed hybrid of Chapter 10, which couples a flexible network to interpretable mechanistic parameters, is the attempt to have it both ways.

Exercises

Exercise 7.1 (*A neuron computes*). A neuron has weights $w = (2, -1)$, bias $b = 0.5$, and tanh activation. Compute its output on input $x = (1, 3)$.

Exercise 7.2 (*Why nonlinearity*). Show that a two-layer network with the *identity* activation (i.e. $\phi(z) = z$) computes only an affine function of its input, no matter how many layers are stacked. Conclude that the nonlinearity is essential to flexibility.

Exercise 7.3 (*A forward pass*). Repeat the computation of Example 7.1 on input $x = 0$ instead of $x = 1$. What is the output? *(Note how the bias in the second hidden neuron breaks the symmetry between the two.)*

Exercise 7.4 (*Counting parameters*). An MLP has input dimension 1 (time t), two hidden layers of width 16, and output dimension 1 (volume V). Count the total number of weights and biases. *(This is roughly the size of network one might use for the running example.)*

Exercise 7.5 (*Choosing an output activation*). Explain why a softplus output activation is a sensible choice for a network predicting tumor volume, referring both to positivity and to the $\ln(K/V)$ term in the Gompertz dynamics. What could go wrong with a linear output activation?

Exercise 7.6 (*Why smooth hidden activations*). In a physics-informed network the derivative of the output with respect to time appears in the loss. Explain why this makes a smooth activation like tanh preferable to ReLU, whose derivative is discontinuous.

Exercise 7.7 (*Learned basis functions*). Explain the sense in which a hidden layer provides a set of basis functions, and how this differs from fitting with a fixed basis such as polynomials. Why does learning the basis increase both flexibility and the risk of overfitting?

Exercise 7.8 (*Existence versus findability*). The universal approximation theorem guarantees a network exists that approximates a given function. List two reasons this guarantee does not ensure that training on data will produce such a network.

Exercise 7.9 (*Approximation is not extrapolation*). Explain why the universal approximation theorem offers no guarantee about a network's predictions outside the region where data live. Relate this to why held-out forecasting is a demanding test of a fitted network.

Exercise 7.10 (*Flexibility and overfitting*). A colleague proposes using a network with 10,000 parameters to fit six noisy tumor measurements. Using the ideas of Section 3.4, explain the danger, and suggest what would have to be added to make such a flexible model trustworthy on so little data.

Chapter 8

Automatic Differentiation

Training a neural network, or fitting any model by gradient methods, requires the gradient of the loss with respect to the parameters. For a network with millions of parameters, computing these derivatives by hand is out of the question, and approximating them by finite differences is both inaccurate and ruinously expensive. *Automatic differentiation* (AD) solves the problem: it computes exact derivatives of any function expressed as a program, at a cost comparable to evaluating the function itself. AD is the technology that makes deep learning practical, and understanding what it computes — and what it does *not* — dispels a good deal of confusion. The chapter is also where we settle a misconception that arose in the running research project: that automatic differentiation is a kind of numerical differentiation. It is not.

8.1 Three ways to take a derivative

There are three distinct ways a computer might obtain a derivative, and they are routinely confused.

Numerical (finite) differences. Approximate $f'(x) \approx (f(x+h) - f(x))/h$ for some small h. This is simple but plagued by a dilemma: too large an h gives a poor approximation (the difference quotient differs from the true derivative by a term of order h), while too small an h gives catastrophic cancellation in floating-point arithmetic (subtracting two nearly equal numbers loses precision). The error therefore first decreases and then *increases* as h shrinks, bottoming out at a best achievable accuracy far short of machine precision. For a function of p variables one gradient also requires $p+1$ evaluations. Finite differences are a tool of last resort.

Symbolic differentiation. Manipulate the formula for f using the rules of calculus to produce a formula for f', as a computer algebra system does. This gives an exact expression, but for a deeply nested function the expression can blow up in size ("expression swell"), and it requires the function to be available as a clean formula rather than as a program with loops and branches.

Automatic differentiation. Evaluate the derivative *exactly* by applying the chain rule mechanically to the elementary operations the program performs, propagating numerical derivative values alongside the ordinary computation. AD is neither approximate (unlike finite differences) nor symbolic (it produces numbers, not formulas). It is the best of both: exact to machine precision, and efficient.

Remark 8.1 (A misconception worth correcting)**.** In the research project behind these notes, a natural early guess was that the framework's autodiff (used to compute dV/dt for the physics

residual of Chapter 10) might be introducing finite-difference error, and that this could explain a puzzling sensitivity to a certain weighting parameter. It cannot: automatic differentiation computes the derivative *exactly*, with no step-size parameter and no truncation error. The puzzle had another source, which we trace in Chapter 10. Knowing what AD actually does — exact chain-rule propagation, not finite differencing — rules out an entire class of false explanations, and saves the time one might otherwise spend chasing a numerical artifact that is not there.

8.2 The chain rule on a computational graph

The key insight is that any function a program computes, however complicated, is built from a sequence of elementary operations — additions, multiplications, exponentials, logarithms — each with a known derivative. The computation can be drawn as a *computational graph*: nodes are intermediate values, edges carry results from one operation to the next. AD walks this graph applying the chain rule.

Consider a small example: $f(x_1, x_2) = \ln(x_1) + x_1 x_2$. The program computes intermediate values

$$a = \ln(x_1), \quad b = x_1 x_2, \quad f = a + b,$$

and AD augments each with its derivative contribution. The chain rule says that to find $\partial f/\partial x_1$ we accumulate the contributions along every path from x_1 to f: the path through a contributes $1/x_1$, the path through b contributes x_2, giving $\partial f/\partial x_1 = 1/x_1 + x_2$. AD performs exactly this accumulation, numerically, for functions with millions of nodes.

8.3 Forward and reverse mode

There are two ways to traverse the graph, differing in efficiency depending on the shape of the function.

Forward mode propagates derivatives from inputs to outputs. Alongside each intermediate value it carries the derivative of that value with respect to *one chosen input*. One forward pass yields the derivatives of all outputs with respect to that one input. To get the gradient with respect to all p inputs requires p passes. Forward mode is therefore efficient when there are few inputs and many outputs. A clean way to think of forward mode is through *dual numbers*: one carries, alongside each value v, a companion $\dot{v}$ representing its derivative, and the elementary operations are extended to act on the pair $(v, \dot{v})$ according to the usual differentiation rules — so that $(v, \dot{v}) \cdot (w, \dot{w}) = (vw, v\dot{w} + w\dot{v})$ encodes the product rule, and so on.

Reverse mode propagates derivatives from outputs back to inputs. A forward pass computes and stores all intermediate values; a backward pass then computes the derivative of *one chosen output* with respect to every intermediate value, and hence with respect to all inputs, in a single sweep. Reverse mode is efficient when there are many inputs and few outputs — which is exactly the situation in machine learning, where the loss is a single scalar output and the parameters number in the thousands or millions. In that setting reverse mode computes the entire gradient at a cost of about two function evaluations, independent of the number of parameters. The price is memory: the intermediate values from the forward pass must be stored for use in the backward pass.

Backpropagation is precisely reverse-mode automatic differentiation applied to a neural network. The famous backpropagation algorithm is not a special invention for networks but an instance of the general reverse-mode procedure, exploiting the layered structure (7.1) so that the backward sweep reuses each layer's stored quantities. When a deep-learning framework "computes the gradients," it is running reverse-mode AD over the computational graph of the loss. Recognizing backpropagation as a special case of a general principle — rather than a mysterious neural-network-specific trick — demystifies it and explains why the same machinery differentiates not just networks but any differentiable program.

Example 8.2 (A reverse-mode sweep)**.** Take $f(x_1, x_2) = \ln(x_1) + x_1 x_2$ again, at $(x_1, x_2) = (2, 3)$. The forward pass records $a = \ln 2$, $b = 6$, $f = \ln 2 + 6$. The backward pass starts with $\partial f/\partial f = 1$ and pushes it back: through the sum, $\partial f/\partial a = 1$ and $\partial f/\partial b = 1$; through $b = x_1 x_2$, the contribution to x_1 is $x_2 = 3$ and to x_2 is $x_1 = 2$; through $a = \ln x_1$, the contribution to x_1 is $1/x_1 = 1/2$. Summing the two contributions to x_1 gives $\partial f/\partial x_1 = 3 + 1/2 = 3.5$, and $\partial f/\partial x_2 = 2$. One forward pass and one backward pass have produced both partial derivatives exactly — and the same two-sweep cost would deliver the gradient no matter how many inputs there were.

8.4 Differentiating through a simulation

AD is not limited to closed-form functions; it can differentiate any program, including one that solves a differential equation numerically by stepping forward in time. This matters for the running example. To fit a mechanistic model whose trajectory has no closed form, one can run a numerical ODE solver and apply AD to the whole solver, obtaining exact derivatives of the trajectory with respect to the model parameters — the Jacobian needed by Levenberg–Marquardt (Section 5.4), or the gradient needed to train a physics-informed network. Likewise, the physics residual of Chapter 10 requires dV/dt where V is the network's output; AD supplies this derivative of the network with respect to its *input* (time), exactly, by the same machinery that computes parameter gradients. The ability to differentiate freely through solvers and networks alike is what makes scientific machine learning computationally feasible.

Remark 8.3 (Two derivatives, one tool)**.** It is worth distinguishing the two roles AD plays in a physics-informed network, because they use the same machinery for different ends. To evaluate the *physics residual*, AD differentiates the network's output with respect to its *input*, time — this is the dV/dt that must match the right-hand side of the ODE. To *train* the network, AD differentiates the loss with respect to the *parameters* — the weights and the physical constants. Both are exact, both come from the same reverse- and forward-mode procedures, and a single framework computes them together. Keeping the two straight prevents confusion when reading or writing such code.

Exercises

Exercise 8.1 (*Finite-difference dilemma*). For $f(x) = e^x$ at $x = 1$, the true derivative is $e \approx 2.71828$. Compute the finite-difference estimate $(f(1 + h) - f(1))/h$ for $h = 10^{-1}, 10^{-4}, 10^{-8}, 10^{-12}$ (a calculator suffices). Describe how the error first decreases and then, for very small h, grows again, and explain why.

Exercise 8.2 (*Chain rule by graph*). Draw the computational graph of $f(x_1, x_2) = \sin(x_1 x_2) + x_1^2$ and use it to compute both partial derivatives at $(x_1, x_2) = (1, 0)$.

Exercise 8.3 (*A reverse-mode sweep*). Following Example 8.2, perform a reverse-mode sweep for $f(x_1, x_2) = x_1^2 x_2 + x_2$ at $(x_1, x_2) = (3, 2)$. Record the forward values, then push derivatives backward to obtain both partials.

Exercise 8.4 (*Forward versus reverse cost*). A function has $p = 1000$ inputs and 1 output. Roughly how many passes does forward-mode AD need to compute the full gradient, and how many does reverse mode need? Reverse the situation (1 input, 1000 outputs) and answer again.

Exercise 8.5 (*Dual numbers*). Using the dual-number rule $(v, \dot{v})(w, \dot{w}) = (vw, v\dot{w} + w\dot{v})$ and the analogous rules for sum and for ln, compute the forward-mode derivative of $f(x) = x \ln x$ at $x = 2$ with respect to x (set the input dual to $(2, 1)$).

Exercise 8.6 (*AD is exact*). Explain in one or two sentences why automatic differentiation, unlike finite differences, has no step-size parameter and no truncation error. Why does this rule out AD as the source of a step-size-like numerical artifact?

Exercise 8.7 (*Backprop is reverse mode*). In your own words, state the relationship between "backpropagation" and "reverse-mode automatic differentiation." Why is the layered structure of a network convenient for the backward sweep?

Exercise 8.8 (*Differentiating a solver*). Explain why being able to apply AD to a numerical ODE solver is useful for fitting a mechanistic model that has no closed-form solution. What quantity does it provide that Levenberg–Marquardt needs?

Exercise 8.9 (*Two derivatives*). In a physics-informed network, AD is used to compute (i) the derivative of the output with respect to time and (ii) the derivative of the loss with respect to the parameters. Explain what each is for and why both are needed.

Exercise 8.10 (*The cost of memory*). Reverse mode stores all intermediate values from the forward pass for use in the backward pass. For a very deep network this memory can be large. Explain why reverse mode incurs this memory cost while forward mode does not, and why we nonetheless prefer reverse mode for training.

Chapter 9

Training Neural Networks

We have a flexible model (Chapter 7), a way to compute gradients (Chapter 8), and optimizers to use them (Chapter 6). Training a network is the act of putting these together: choosing a loss, initializing the parameters, and iterating until the network fits the data. This chapter surveys the practical ingredients and, more importantly for our purposes, the characteristic ways training fails — because the applied chapters will diagnose real instances of these failures.

9.1 The training loop

Training proceeds by repeating four steps. First, a minibatch of data is drawn. Second, a *forward pass* computes the network's predictions and the resulting loss. Third, a *backward pass* (reverse-mode AD) computes the gradient of the loss with respect to every parameter. Fourth, the optimizer — usually Adam (Section 6.4) — updates the parameters. The cycle repeats for many epochs, the loss ideally decreasing until it plateaus. Around this core sit the choices that determine whether training succeeds: the loss function, the initialization, the learning rate, and the criteria for stopping.

It is useful to watch the *training curve* — the loss plotted against iteration — because its shape diagnoses much. A healthy curve falls quickly at first and then flattens as the iterate nears a minimum. A curve that diverges or oscillates signals too large a learning rate; one that barely moves signals too small a rate or a vanishing gradient; one that falls and then *rises* on held-out data while continuing to fall on training data signals overfitting. Reading the training curve is the first diagnostic skill of practical machine learning, and much of the rest of this chapter is about interpreting what it shows.

9.2 Loss functions

For regression — predicting a continuous quantity such as volume — the default loss is the mean squared error (3.1) between predictions and targets, for the reasons given in Chapter 5. Other losses suit other tasks (for instance cross-entropy for classification), but squared error is what the running example uses. A central theme of Chapter 10 is that the loss can encode more than fit to data: it can also penalize violations of a physical law, by adding a term that measures how badly the network's output disobeys a differential equation. Designing such *composite* losses — and balancing their terms — is where scientific machine learning departs from generic training, and we devote much of the next chapter to the subtleties it introduces.

9.3 Initialization

Because the training problem is nonconvex (Chapter 7), the starting parameters matter. Weights are typically initialized to small random values, scaled according to layer width so that signals neither vanish nor explode as they propagate through the layers. Poor initialization can stall training entirely — if all weights start equal, every neuron in a layer computes the same thing and receives the same gradient, so they never differentiate, a pathology called *symmetry* that random initialization is designed to break. For the small networks of the running example, initialization is less fraught than for very deep networks, but the principle stands: training is a descent from a starting point, and where one starts influences where one lands.

The same logic applies to any *physical* parameters learned alongside the network. In the running example the growth rate and carrying capacity are initialized from the data — the carrying capacity near the largest observed volume, the rate from the early slope — and represented through their logarithms to keep them positive, exactly as in the classical fit (Section 5.5). A good initialization of the physical parameters matters more than that of the network weights, because the physical parameters are few, interpretable, and the actual object of interest.

9.4 How training fails

More instructive than the recipe is the catalogue of failure modes, because recognizing them is half of using networks well.

The loss does not decrease. If the learning rate is too large, the iterates overshoot and the loss diverges or oscillates; if too small, progress is imperceptible. The first diagnostic for any failed training run is to examine the learning rate, ideally by trying a range spanning several orders of magnitude and watching which produces steady decrease.

The loss decreases but the result is wrong. This is the insidious case. The network drives the *training* loss down by fitting the data — possibly including its noise — yet predicts poorly on held-out data. This is overfitting (Section 3.4) in its training-dynamics guise, and no amount of further optimization cures it; the remedy is more data, less flexibility, or built-in prior knowledge. The training curve alone cannot reveal this failure — it looks like success — which is why a held-out validation curve must be watched alongside it.

Parameters that should change do not. A subtle failure occurs when a quantity one intends to learn is not actually connected to the loss gradient. In the research project behind these notes, an early version of a physics-informed network failed to update the physical parameters a and K at all: the loss decreased (the network's flexible part was fitting the data) while a and K sat frozen at their initial values. The cause was structural — the parameters had not been registered among the quantities the optimizer updates, so although they entered the loss, no update was applied to them. The fix was to compute their gradients explicitly and update them alongside the network weights. The general lesson: if a parameter is not moving, check that a gradient actually reaches it and that the optimizer is actually stepping it, before suspecting anything subtler. This is a failure of plumbing, not of mathematics, and it is among the most common and most maddening bugs in scientific machine learning precisely because the loss curve looks healthy throughout.

Competing loss terms. When the loss is a sum of terms that pull in different directions — as in the composite losses of Chapter 10 — training can stall in a compromise that satisfies neither term well, or be dominated by whichever term happens to be larger in magnitude. Diagnosing and balancing such terms is a recurring theme of the applied chapters, and the next chapter treats it in detail for the physics-informed case.

Vanishing and exploding gradients. In deep networks the gradient, propagated back through many layers, can shrink toward zero or grow without bound, stalling or destabilizing training. This is less a concern for the shallow networks of the running example, but it explains several standard practices — careful initialization, certain activation functions, and architectural devices — that exist largely to keep gradients well behaved through depth.

9.5 Regularization

Regularization is any modification that discourages the network from overfitting. Several standard forms recur.

A *weight penalty* (also called weight decay, or ℓ_2 regularization) adds a multiple of $\|\theta\|^2$ to the loss, so the optimizer prefers smaller weights and hence smoother functions; an ℓ_1 penalty, adding $\|\theta\|_1$, instead drives many weights exactly to zero, yielding a sparse model. *Early stopping* halts training when held-out error stops improving, before the network has had time to fit the noise — a remarkably effective and nearly free form of regularization. *Dropout* randomly disables a fraction of neurons during each training step, preventing the network from relying too heavily on any one of them and acting as a kind of averaging over many smaller networks. *Data augmentation* enlarges the effective dataset by adding transformed copies of the data, though it is more natural for images than for the small numerical datasets here.

For the scientific problems of this course the most powerful form of regularization is not generic but *physical*: constraining the network to obey a known law, which is the subject of the next chapter. A physical constraint is a far stronger prior than a generic smoothness penalty, because it encodes specific, correct information about the system — when, that is, the law is correct, a caveat that Chapter 11 shows to be decisive. The general principle is that regularization injects prior belief, and the more specific and correct the belief, the more it helps; a generic smoothness penalty believes only that the truth is smooth, while a physics constraint believes a particular equation, which is far more — and far more dangerous if wrong.

Exercises

Exercise 9.1 (*Diagnosing a flat loss*). During training the loss does not move from its initial value across many epochs. List three distinct possible causes and, for each, the check you would run.

Exercise 9.2 (*Reading a training curve*). Sketch (or describe) the training curve you would expect from (a) a well-chosen learning rate, (b) too large a learning rate, (c) too small a learning rate. What additional curve must you watch to detect overfitting, and why is the training curve alone insufficient?

Exercise 9.3 (*Train–test gap*). A network achieves training MSE 0.1 and held-out MSE 50. What is this pattern called, and what are two interventions that might help? Would running the optimizer longer help?

Exercise 9.4 (*A frozen parameter*). Suppose a physical parameter enters the loss but never changes during training. Explain how this can happen even though the loss does decrease, and describe how you would verify whether a gradient is reaching the parameter.

Exercise 9.5 (*Initialization symmetry*). Why is it a mistake to initialize all weights in a layer to the same value? What property of the neurons does random initialization break, and what happens to the gradients if the symmetry is not broken?

Exercise 9.6 (*Forms of regularization*). Briefly describe weight decay, early stopping, and dropout, and explain in one sentence each how it discourages overfitting.

Exercise 9.7 (*Physical versus generic regularization*). Contrast a generic weight-size penalty with a physics-based constraint as forms of regularization. Under what circumstance does the physics-based constraint stop being helpful and start being harmful? *(This anticipates the misspecification analysis of Chapter 11.)*

Exercise 9.8 (*Balancing loss terms*). A composite loss is $L_{\text{data}} + \lambda L_{\text{phys}}$. Describe qualitatively what happens to the trained network as $\lambda \to 0$ and as $\lambda \to \infty$, and why an intermediate value is usually wanted.

Exercise 9.9 (*Initializing physical parameters*). For the running example, the physical parameters a and K are learned alongside the network weights. Propose data-driven initial values for them, and explain why getting their initialization right matters more than the initialization of the network weights.

Exercise 9.10 (ℓ_1 *versus* ℓ_2). A weight penalty can use $\|\theta\|^2$ or $\|\theta\|_1$. Explain the qualitatively different effect of the two on the fitted weights, and why the ℓ_1 choice is associated with sparsity. *(This idea returns in the sparse-regression approach to equation discovery in Chapter 12.)*

Part IV

Scientific Machine Learning

Chapter 10

Physics-Informed Neural Networks

We arrive at the first method that genuinely fuses the two cultures of the course: the flexibility of neural networks and the structure of mechanistic models. A *physics-informed neural network* (PINN) is a network trained not only to fit data but also to obey a differential equation. The hope is that the physics acts as a powerful regularizer, taming the network's tendency to overfit sparse noisy data. This chapter explains the construction, works it out for the running example, and confronts both its promise and a subtlety in its loss design that the research project surfaced. The deeper question — whether the physics constraint helps when the assumed physics is *wrong* — is taken up in Chapter 11.

10.1 The idea: data loss plus physics loss

Suppose we believe a quantity $V(t)$ obeys an ODE $dV/dt = f(V;\theta)$ with unknown parameters θ, and we have sparse noisy measurements (t_i, y_i). A PINN represents the trajectory by a neural network $\hat{V}_\omega(t)$ with weights ω, and trains it to do two things at once:

- **fit the data**: the network's output at the measurement times should match the measurements;
- **obey the physics**: the network's output should satisfy the differential equation at a set of points throughout the interval.

These two demands become two terms in a composite loss. The *data loss* is the familiar mean squared error at the measurement times,

$$L_{\text{data}}(\omega) = \frac{1}{N}\sum_{i=1}^{N}\big(\hat{V}_\omega(t_i) - y_i\big)^2.$$

The *physics loss* measures how badly the network violates the ODE. At a set of *collocation* points τ_j spread across the time interval, we ask that the network's own derivative match the right-hand side of the ODE:

$$L_{\text{phys}}(\omega,\theta) = \frac{1}{M}\sum_{j=1}^{M}\left(\frac{d\hat{V}_\omega}{dt}(\tau_j) - f\big(\hat{V}_\omega(\tau_j);\theta\big)\right)^2.$$

Here $d\hat{V}_\omega/dt$ is the derivative of the network with respect to its input, supplied exactly by automatic differentiation (Chapter 8). The total loss is their weighted sum,

$$L_{\text{total}}(\omega,\theta) = L_{\text{data}}(\omega) + \lambda\, L_{\text{phys}}(\omega,\theta), \tag{10.1}$$

with a *physics weight* $\lambda > 0$ controlling how much the network is required to respect the dynamics relative to fitting the data.

Two features deserve emphasis. First, both the network weights ω and the *physical* parameters $\theta = (a, K)$ are learned simultaneously by minimizing (10.1) — the PINN solves the inverse problem (recover θ) and the forward problem (represent the trajectory) in one optimization. Second, the physics loss can be evaluated at *any* points τ_j, not only where data exist, because it asks about the equation rather than about measurements; this is how physical knowledge fills the gaps between sparse data.

10.2 The collocation method

The idea of enforcing an equation at scattered points rather than solving it exactly is older than neural networks; it is the *collocation* method of numerical analysis. To solve a differential equation by collocation, one posits a flexible trial function (classically a combination of polynomials or other basis functions; here, a neural network) and requires it to satisfy the equation exactly at a finite set of collocation points, choosing the trial function's parameters to make this so. The PINN is collocation with a neural network as the trial function and gradient-based optimization as the solver.

Two practical questions attend the method: how many collocation points, and where? More points enforce the physics more thoroughly but cost more to evaluate; too few leave gaps where the network can violate the equation unpenalized. Their placement matters for the same reason sampling design matters for data (Chapter 11): collocation points concentrated where the dynamics are most active constrain the solution most effectively. In the running example, points spread across the time interval — with adequate coverage of the early, fast-growing phase — suffice, because the trajectory is one-dimensional and smooth. In higher-dimensional problems the placement of collocation points becomes a serious design question in its own right.

10.3 Construction for the running example

For the Gompertz model the right-hand side is $f(V; a, K) = aV \ln(K/V)$, so the physics residual at a collocation point is

$$\frac{d\hat{V}_\omega}{dt}(\tau_j) - a\,\hat{V}_\omega(\tau_j) \ln\left(\frac{K}{\hat{V}_\omega(\tau_j)}\right).$$

Two design choices from Chapter 7 now pay off. The network's output activation is *softplus*, guaranteeing $\hat{V}_\omega(t) > 0$ so that the logarithm is always defined; and the hidden activations are smooth (tanh) so that $d\hat{V}_\omega/dt$ is well behaved. The physical parameters are represented through their logarithms — learning $\log a$ and $\log K$ rather than a and K — which automatically keeps them positive and improves the conditioning of the optimization. Training minimizes (10.1) over ω, $\log a$, and $\log K$ jointly, using Adam (Section 6.4); the gradients with respect to all three reach the optimizer through reverse-mode AD, and — recalling Section 9.4 — one must ensure the physical parameters are actually among the quantities the optimizer updates, a step whose omission once left them frozen.

Remark 10.1 (Why learn the logarithms of the parameters)**.** The reparametrization $a \mapsto \log a$, $K \mapsto \log K$ does two things at once. It enforces positivity for free — any real value of $\log a$ corresponds to a positive a — so the optimizer never needs to be constrained or to stumble into negative, meaningless values. And it improves conditioning: the carrying capacity $K \approx 1000$ and the rate $a \approx 0.4$ differ by orders of magnitude, so in the raw parameters the loss surface is hugely

elongated (the conditioning problem of Section 4.3), while in the logarithms the two are comparable in scale and the surface is rounder. This is the same change of variable that linearized the Gompertz equation in Section 2.5, and it is a recurring trick: work in the variables where the problem is best conditioned.

10.4 The loss-balancing problem

The physics weight λ is the PINN's most delicate knob, and it conceals a trap that cost the research project some confusion. The difficulty is that the data loss and the physics loss are measured in *different units and different scales*. The data loss is a squared volume (mm^6); the physics loss is a squared rate (mm^6 per day^2). Their raw magnitudes can differ by orders of magnitude, so a "natural-looking" value of λ such as 1 may in effect almost ignore one term.

Remark 10.2 (A puzzle and its resolution)**.** In the project, a fixed-scale implementation found that λ had a noticeable effect on the result only when it was extremely small — around 10^{-3} — which seemed mysterious, since the literature and intuition suggested order-one weights. The resolution is scale. When the physics residual is numerically much larger than the data residual, the product λL_{phys} only becomes comparable to L_{data} once λ is correspondingly tiny. The effective weight is not λ but λ times the ratio of the two terms' typical magnitudes. A cleaner formulation *normalizes* each loss term by its characteristic scale before weighting,

$$L_{\text{total}} = \frac{L_{\text{data}}}{\bar{\ell}_{\text{data}}} + \lambda \, \frac{L_{\text{phys}}}{\bar{\ell}_{\text{phys}}},$$

so that λ has a consistent, interpretable meaning across problems and the mysterious 10^{-3} becomes an order-one weight in disguise. The episode is a reminder that the numbers an optimizer sees depend on units, and that matching scales is part of designing a composite loss.

Adaptive weighting. Beyond fixing the scales once, a body of recent work lets the weight λ *adapt* during training, adjusting it so that the two loss terms contribute comparably to the gradient at each stage. The motivation is that the right balance early in training (when the network is far from fitting anything) may differ from the right balance late (when it is refining a good fit). Schemes for this monitor the relative sizes of the two terms' gradients and rescale λ to keep them in proportion. Such adaptivity can help, but it adds machinery and tuning of its own, and for a problem as small as the running example a well-chosen fixed, scale-normalized weight is usually adequate. The general lesson stands regardless of the scheme: the balance between data-fitting and physics-enforcement is a first-class design decision, not an afterthought.

10.5 What the physics loss buys, and what it cannot

When the assumed law is correct, the physics loss is a strong regularizer. In the data-rich, high-noise regime it can fight the network's urge to oscillate through noisy points, pulling the trajectory toward a smooth curve that obeys the dynamics; and in the sparse regime it constrains the network between the few data points, where pure data-fitting would leave the trajectory underdetermined. The research project observed exactly this benefit in the sparse regime, where the physics anchor kept the network from chasing noise. In the language of Chapter 3, the physics loss reduces *variance* — it stiffens the fit against the particular noise drawn — at the risk of introducing *bias* if the assumed law is wrong.

But a regularizer encodes an assumption, and an assumption can be false. If the true dynamics are *not* Gompertz — if the tumor follows a different growth law — then the physics loss pulls the network toward the wrong equation, and a larger λ makes matters worse, not better, by enforcing the wrong constraint more strictly. The PINN is only as good as the physics it is told to obey. This is not a flaw to be tuned away but a fundamental limit, and articulating it precisely — showing how a misspecified physics loss biases the recovered parameters, and why more physics weight amplifies the bias — is the work of the next chapter. The PINN, in short, is a powerful tool for the *forward-correct* inverse problem and a potential trap for the misspecified one. The bias–variance language makes the danger precise: turning up λ trades variance for bias, and when the physics is wrong, the bias it buys is not worth the variance it saves.

Exercises

Exercise 10.1 (*Anatomy of the loss*). Write out the full PINN loss (10.1) for the logistic model $f(V) = rV(1 - V/K)$, giving both terms explicitly in terms of the network $\hat{V}_\omega$ and the parameters r, K.

Exercise 10.2 (*Collocation points*). Explain why the physics loss can be evaluated at points where no data exist, whereas the data loss cannot. How does this let physical knowledge "fill in" between sparse measurements?

Exercise 10.3 (*Collocation, classically*). The collocation method predates neural networks. Describe, in one or two sentences, what the method does with a classical trial function (such as a polynomial), and identify what plays the role of the trial function and the solver in a PINN.

Exercise 10.4 (*Units*). The data loss for tumor volume has units of mm^6. Show that the Gompertz physics loss has units of mm^6/day^2. Use this to explain why a single dimensionless λ multiplying the raw physics loss has a scale-dependent, and therefore unintuitive, effect.

Exercise 10.5 (*Why logarithms*). Explain the two benefits of learning $\log a$ and $\log K$ rather than a and K directly: positivity and conditioning. Why is conditioning especially relevant when $K \approx 1000$ and $a \approx 0.4$?

Exercise 10.6 (*The frozen-parameter trap*). In a PINN one learns network weights and physical parameters together. Referring to Section 9.4, explain what symptom indicates the physical parameters are not being updated, and what must be checked.

Exercise 10.7 (*When more physics hurts*). Suppose the true dynamics are von Bertalanffy but the PINN's physics loss assumes Gompertz. Argue qualitatively that increasing λ will, beyond some point, worsen the recovered parameters. *(You will make this precise in Chapter 11.)*

Exercise 10.8 (*Normalization*). Explain how normalizing each loss term by its characteristic magnitude before applying λ makes the weight's meaning consistent across problems. Why does this turn a mysterious optimal $\lambda \approx 10^{-3}$ into an order-one quantity?

Exercise 10.9 (*Bias and variance, again*). Cast the effect of the physics weight λ in the bias–variance language of Section 3.5. What does increasing λ do to variance, and what does it do to bias when the assumed law is (a) correct and (b) wrong?

Exercise 10.10 (*Adaptive weighting*). Explain the motivation for letting λ adapt during training rather than fixing it. What does an adaptive scheme monitor, and why might the ideal balance differ between early and late training?

Chapter 11

Inverse Problems and Identifiability

This is the conceptual heart of the course. Everything so far has been about *finding* parameters that fit data. This chapter is about a harder and more important question: when the parameters we find are trustworthy, and when they are not. We will see that a fit can be excellent and the parameters still meaningless — because the data do not determine them (poor *identifiability*), or because the model itself is wrong (*misspecification*). These are not exotic failures; they are the normal condition of sparse, noisy inverse problems, and recognizing them is what separates sound scientific modeling from confident self-deception.

11.1 Well-posed and ill-posed problems

A problem is *well-posed*, in Hadamard's classical sense, if a solution exists, is unique, and depends continuously on the data. Inverse problems routinely violate the last two conditions. Uniqueness fails when different parameters produce the same observations. Continuity fails when tiny changes in the data — a little more noise — produce large changes in the recovered parameters. A problem failing either is *ill-posed*, and ill-posedness is the rule rather than the exception when we try to infer a continuous model from finite noisy samples.

The forward problem of Chapter 2 is well-posed: one rule and one initial condition give one trajectory, depending smoothly on the parameters. The inverse problem inherits none of these guarantees. Running an optimizer to convergence tells us we have found a minimizer of the loss; it does not tell us the minimizer is unique, stable, or correct. The recurring theme of this chapter is that the optimizer's success and the estimate's trustworthiness are different things, and that establishing the latter requires work the optimizer does not do.

11.2 Structural versus practical identifiability

It is useful to separate two notions.

A model is *structurally identifiable* if, given *perfect* continuous data, distinct parameters give distinct outputs — so the parameters could in principle be recovered exactly. This is a property of the model alone, independent of any particular dataset. The Gompertz, Richards, and von Bertalanffy families are each structurally identifiable in isolation: with the entire noise-free trajectory in hand, their parameters are determined.

A model is *practically identifiable* from a *given* dataset if the finite, noisy data actually pin the parameters down to a useful precision. This is the property that fails in practice. A model can be perfectly identifiable in principle yet hopelessly unidentifiable from five noisy points, because over the

range those points span, a whole range of parameter combinations produce nearly indistinguishable trajectories. Practical identifiability depends on the model, the noise, *and* the sampling design — where and how often we measured.

Remark 11.1 (A third, subtler failure)**.** Beyond these two lies the failure mode that matters most for scientific modeling: *misspecification*. Structural and practical identifiability both assume the true law lies in the model family. When it does not — when we fit Gompertz to data generated by a different law — the parameters we recover are not estimates of anything real, however identifiable the wrong family may be. We treat this in Section 11.5.

11.3 The identifiability ridge

The geometry of poor practical identifiability is worth seeing concretely, because it explains a great deal of otherwise puzzling behavior. Recall from Section 4.3 that the loss near a minimum looks like a quadratic bowl whose shape is set by the Hessian. When the bowl is round, the minimizer is sharply located. When the bowl is instead a long, nearly flat-floored valley — one Hessian eigenvalue tiny — the minimizer is sharply located only *across* the valley; *along* the valley floor, a wide range of parameter values give almost the same loss. Such a valley is called an *identifiability ridge*, and a parameter combination lying along it is poorly determined by the data.

For the running example this is not hypothetical. Consider fitting the Gompertz parameters (a, K) to data that saturate near the carrying capacity. The growth rate a and the carrying capacity K trade off against each other: a smaller a with a larger K can produce nearly the same trajectory over the observed range as a larger a with a smaller K. The loss therefore has a ridge running diagonally through the (a, K) plane, and the data choose a point along it only weakly. Two consequences follow, both observable.

Strong anticorrelation of estimates. If we refit the model on many noisy realizations of the same experiment, the estimates $(\hat{a}, \hat{K})$ do not scatter in a round blob; they scatter *along the ridge*, tracing out a strong negative correlation. In the project's benchmark, the correlation between $\log \hat{a}$ and $\log \hat{K}$ across repeated fits was about -0.97 — almost perfectly anticorrelated. A large fitted a reliably comes paired with a small fitted K, not because the tumor is any particular way, but because that is the direction the data fail to constrain.

Large variability of each parameter alone. Because the ridge is long, each parameter individually varies a great deal from one noise realization to the next, even though their *combination* (the position across the ridge) is stable. One reports a precise-looking $\hat{a}$ from a single dataset at one's peril: a second dataset from the same experiment may give a substantially different $\hat{a}$, sliding along the ridge.

Example 11.2 (A two-parameter ridge by linearization)**.** The ridge can be seen analytically. Near the best fit, expand the loss to second order (Section 2.3); the level sets of the loss are ellipses governed by the Hessian $H = 2J^\top J$, where J is the Jacobian of the trajectory with respect to $(\log a, \log K)$. The eigenvector of H with the small eigenvalue is the ridge direction — the combination of $\log a$ and $\log K$ the data barely constrain — and the ratio of the large to the small eigenvalue is the condition number, which measures how elongated the ellipse is. A correlation of -0.97 between the estimates corresponds to a markedly elongated ellipse tilted along the line $\log a + \log K \approx \text{const}$; that is, the product aK (roughly, the early growth behavior) is well determined while the two factors separately are not. Computing J and its eigen-decomposition for a given sampling design is the quantitative way to predict, before collecting data, which parameter combinations will be identifiable.

Tikhonov regularization as a response. One classical response to an identifiability ridge is *regularization*: adding to the loss a penalty that gently disfavors extreme parameter values,

$$L_{\text{reg}}(\theta) = L(\theta) + \gamma \left\|\theta - \theta_0\right\|^2,$$

where θ_0 is a prior guess and γ controls the strength. This *Tikhonov* penalty curves the flat floor of the ridge, giving the loss a unique, stable minimizer where before it had a near-flat valley. The cost is bias: the estimate is pulled toward θ_0, so one trades the wild variance of the unregularized fit for a controlled bias — the bias–variance tradeoff of Section 3.5 in yet another guise. The physics loss of Chapter 10 is, in this light, a structured regularizer, and the danger of a wrong physics is exactly the danger of regularizing toward a wrong θ_0.

11.4 Profiling and sensitivity

How does one *detect* an identifiability problem in practice, short of refitting on hundreds of synthetic datasets? Two standard tools.

The profile likelihood. Fix one parameter at a range of values away from its best-fit value; at each fixed value, re-optimize over all the *other* parameters and record the resulting best loss. Plotting this "profiled" loss against the fixed parameter reveals how much the data actually constrain it. A sharp, steep profile means the parameter is well determined — moving it away from the optimum forces the loss up quickly. A flat profile means it is poorly determined — the parameter can be moved far with little penalty, the others compensating. The profile likelihood thus diagnoses practical identifiability one parameter at a time, and a flat profile is the signature of a ridge.

Sensitivity and the Fisher information. A complementary view comes from *sensitivities*: the partial derivatives of the trajectory with respect to the parameters, $\partial V/\partial\theta_j$, which form the columns of the Jacobian J. Where a sensitivity is large, the data carry much information about that parameter; where it is small, little. The matrix $J^\top J$ (closely related to the *Fisher information*) summarizes this, and its near-singularity is the formal statement that some parameter combination is poorly informed — the same near-singularity that makes the Hessian's small eigenvalue and the ridge. For the Gompertz model, the sensitivity to the growth rate a is largest in the intermediate-volume region, somewhat past the inflection point at $V = K/e$ but before saturation; data concentrated there are most informative about a, a fact we exploit below.

11.5 Misspecification: fitting the wrong equation

The deepest difficulty is not that the right model is hard to pin down, but that we may be fitting the wrong model entirely. Suppose the tumor truly follows the Richards law, but we fit Gompertz. The Gompertz family does not contain the truth, so there is no "correct" (a, K) to recover. The optimizer will nonetheless return the Gompertz parameters that best mimic the Richards data over the observed range — a precise, confident answer to a question with no real answer.

What makes this dangerous is that the misspecified fit can look *good*. The three growth families of Section 2.6 were deliberately matched to share a carrying capacity and initial volume, making their trajectories genuinely similar over a typical observation window. A Gompertz curve can thread Richards data so closely that no diagnostic based on goodness of fit raises alarm.

Remark 11.3 (Near the carrying capacity, the laws coincide*)**.** There is a precise reason the misspecified fit is so seductive. Near the carrying capacity $V^\star$, every one of the three growth laws is, to first order, the same linear relaxation toward equilibrium: writing $V = V^\star - \delta$ for small δ, each law reduces to $dV/dt \approx c\,(V^\star - V)$ for some constant c (Example 2.5 did this for Gompertz; the others are similar, with $c = r\nu$ for Richards and $c = \beta/3$ for von Bertalanffy). The laws differ only in their *higher-order* behavior and in the early-growth region away from saturation. So data dominated by measurements near $V^\star$ carry almost no information distinguishing the families, and a Gompertz model can absorb Richards data by matching this shared near-equilibrium behavior — at the cost of biased parameters that mean nothing biologically. This is why both the misspecification and the sampling-design issues point to the same culprit: too much of the data sitting where the models agree.

The training error gives no warning. One might hope that a misspecified model would betray itself through a poor fit. It does not. In the project's benchmark, a Gompertz model fit to Richards data achieved a *lower* training error than a correctly specified Gompertz fit to genuine Gompertz data — partly because the particular Richards trajectory carried less effective noise over the window, but the point stands: the training error ranked the misspecified model *above* the correct one. A practitioner choosing models by training error alone would have preferred the wrong model. Crucially, *goodness of fit cannot diagnose misspecification.* In the bias–variance language, a misspecified mechanistic model is a high-bias choice whose bias does not vanish with more data — yet on a single finite sample, that systematic bias can be masked by, and even smaller than, the noise-driven error of a correctly specified model.

11.6 What does reveal these failures

If training error is blind to both poor identifiability and misspecification, what is not? Three things, developed further in Chapter 15.

Held-out forecasting. A misspecified or poorly identified model may fit the observed window yet predict the future badly. Hiding later measurements during fitting and testing against them is the single most informative diagnostic, because it asks the model to do the thing it was actually built for — predict — under conditions it has not seen. In the project, the misspecified Gompertz fit, despite its excellent training error, forecast the tumor's later size substantially wrong.

Repeated refitting. Refitting on many noise realizations exposes the identifiability ridge: if the estimates scatter enormously, or trace a tight anticorrelation, the parameters are poorly determined regardless of how good any single fit looks. The profile likelihood (Section 11.4) achieves the same diagnosis more cheaply.

Comparison across model families. Fitting several candidate laws and comparing their forecasts — not their training errors — can reveal that the data do not actually select among them, which is itself the crucial finding that one's parameters are family-dependent and not to be trusted.

These diagnostics share a philosophy: never trust a model on the data used to fit it. Test it on what it has not seen, perturb the data and refit, and compare alternatives. The remainder of the course puts this philosophy to work.

Exercises

Exercise 11.1 (*Hadamard*). State the three conditions for a problem to be well-posed, and for each give a way an inverse problem might violate it.

Exercise 11.2 (*Structural versus practical*). Explain the difference between structural and practical identifiability. Give an example of a model that is structurally identifiable but, from a particular sparse dataset, practically unidentifiable.

Exercise 11.3 (*Reading a ridge*). You refit a two-parameter model on 30 noisy datasets and plot the resulting $(\hat{a}, \hat{K})$ pairs. They fall tightly along a downward-sloping line. What does this tell you about the identifiability of a and K individually, and about a particular combination of them?

Exercise 11.4 (*The Hessian and the ridge*). In Example 11.2, the ridge direction is the eigenvector of the Hessian with the small eigenvalue. Explain why a small Hessian eigenvalue corresponds to a poorly determined parameter combination, and how the condition number relates to the strength of the anticorrelation between estimates.

Exercise 11.5 (*Profile likelihood*). Describe the profile-likelihood procedure for a two-parameter model. What does a flat profile for one parameter indicate, and how does this connect to the identifiability ridge?

Exercise 11.6 (*Tikhonov tradeoff*). Adding a penalty $\gamma \left\| \theta - \theta_0 \right\|^2$ to the loss makes an ill-posed problem have a unique stable minimizer. Explain the benefit (reduced variance) and the cost (bias toward θ_0), and relate the choice of γ to the bias–variance tradeoff.

Exercise 11.7 (*Near-equilibrium coincidence*). Using the result of Example 2.5, explain why three different growth laws all look like $dV/dt \approx c(V^\star - V)$ near their shared carrying capacity, and why this makes them hard to distinguish from data taken near saturation.

Exercise 11.8 (*Training error is blind*). A student fits two models to the same data; model A (correctly specified) has training MSE 3000, model B (misspecified) has training MSE 500. Explain why the student should *not* conclude that B is the better model, and propose a diagnostic that would settle the matter.

Exercise 11.9 (*Sampling design*). For Gompertz dynamics, explain why measurements concentrated in the early, fast-growing phase determine the growth rate a better than measurements spread evenly in time, most of which fall near saturation. Relate your answer to the sensitivity $\partial V/\partial a$ and the orientation of the identifiability ridge.

Exercise 11.10 (*Misspecification as bias*). In the bias–variance language, explain why a misspecified mechanistic model is a high-bias choice whose bias does not vanish as the amount of data grows, and how this bias can nonetheless be masked on a single finite sample.

Chapter 12

Symbolic Regression and Equation Discovery

The methods so far have all assumed a model and estimated its parameters. We now ask the most ambitious question of the course: can we discover the *equation itself* from data, rather than positing it? This is *symbolic regression* — searching over mathematical expressions for one that explains the data. It is the technical heart of "AI for mathematics" in the equation-discovery sense, and it is also where the gap between promise and practice is widest. This chapter explains the approach, confronts its central obstacle (estimating derivatives from noisy data), surveys the two main families of discovery method, and lays out the validation pipeline the running project converged on after its naive version failed.

12.1 The goal: discovering the right-hand side

For an autonomous ODE $dV/dt = f(V)$, discovering the dynamics means finding the function f. Symbolic regression attempts this by searching over a space of candidate expressions — built from a chosen set of operations such as $+, -, \times, \div, \ln$, and powers — for the expression that best fits the data, while penalizing complexity so that the result is interpretable rather than an arbitrary curve. The output is not a single number or a fitted coefficient but a *formula*, which is what makes the approach so appealing: a discovered equation can be read, interpreted, and connected to mechanism in a way that a neural network's weights cannot.

What is fit to what matters here. To discover $dV/dt = f(V)$, the regressor needs pairs of V and its rate of change dV/dt: it fits a function from volume to growth rate. The inputs are values of V; the targets are the corresponding values of dV/dt. This is the right framing, and getting it right is not automatic — a natural early misstep, encountered in the project, is to fit V as a function of time t instead, which discovers a description of the particular trajectory rather than the underlying law. The law lives in the $(V, dV/dt)$ relationship, not the (t, V) one. The distinction matters because the law is what generalizes: the same $f(V)$ governs every trajectory regardless of initial condition, whereas a fitted $V(t)$ describes only the one trajectory observed.

12.2 Two families of method

Two broad approaches to equation discovery have matured, and they make different tradeoffs.

Genetic programming. The older and more general approach searches the space of expressions directly, using an evolutionary algorithm. Candidate expressions are represented as trees (a node is an operation, its children its arguments), and a population of such trees is evolved: high-performing trees are kept, combined ("crossover" swaps subtrees), and mutated (a node or subtree is changed at random), generation after generation, under selection pressure toward expressions that fit well and stay simple. Genetic programming can in principle discover any expression in its operator set, but the search is stochastic and can be slow, and it offers no guarantee of finding the best expression. Modern symbolic-regression tools in this family are nonetheless effective candidate generators.

Sparse regression. A newer approach trades generality for reliability. Rather than search all expressions, one fixes in advance a *library* of candidate terms — say $1, V, V^2, V\ln V, \sqrt{V}, \ldots$ — and seeks a *sparse* linear combination of them that matches the data, using a penalty (the ℓ_1 idea of Section 9) that drives most coefficients to zero. The result is an equation built from a few library terms, found by solving a convex or nearly-convex problem rather than by stochastic search. This approach — exemplified by methods that identify nonlinear dynamics from data — is fast and reliable *when the true terms are in the library*, but it can discover nothing outside the library one supplies, so it trades the open-ended reach of genetic programming for speed and stability. The choice between the families is itself a modeling decision: sparse regression when one has a good guess at the relevant terms, genetic programming when one does not.

12.3 The central obstacle: derivatives from noisy data

Here is the difficulty that dominates everything, whichever family one chooses. The targets dV/dt are not measured; they must be *estimated* from the noisy samples of V. And estimating a derivative from noisy data is notoriously unstable, because differentiation amplifies noise: the difference between two nearby noisy measurements is dominated by the noise, not the signal. A naive finite difference between consecutive sparse, noisy points can have an error far larger than the quantity it estimates — recall from Section 2.4 that differencing two measurements each of noise standard deviation s, a time h apart, yields an estimate with noise standard deviation $s\sqrt{2}/h$, which *grows* as the points are placed closer together.

The standard remedy is to *smooth first, differentiate second.* One fits a smooth curve $\widetilde{V}(t)$ through the noisy points — not passing exactly through them, but tracking their trend — and then differentiates that smooth curve analytically to obtain $\widetilde{V}'(t)$, and hence the estimated pairs $(\widetilde{V}(t_i), \widetilde{V}'(t_i))$ to feed the regressor. The quality of the discovered law is largely determined by the quality of this derivative estimate, which is the bottleneck the project spent the most effort on.

Methods for estimating the derivative. Several smoothing strategies trade off bias and variance differently. *Smoothing splines* fit a piecewise-polynomial curve minimizing a balance between fidelity to the data and a roughness penalty (the integrated squared second derivative); a smoothing parameter sets the balance. *Local polynomial* or *Savitzky–Golay* methods fit a low-degree polynomial in a sliding window and read off its derivative, smoothing by the choice of window size. *Total-variation regularization* estimates the derivative directly as the function whose integral matches the data and whose total variation is small, which handles sharp features better than spline smoothing. All share the same fundamental tension: too little smoothing and the derivative is swamped by noise; too much and real features are flattened away. The natural way to set the smoothing strength is to tie it to the known noise level, so the smoother is allowed to miss each

point by about as much as the noise — no more, no less. When the data are noise-free the smoother should essentially interpolate; as the noise grows, so should the smoothing.

The overshoot pathology. A vivid failure mode, observed directly in the project, arises when a polynomial or insufficiently constrained smoother is fit to a saturating curve. A polynomial of even modest degree, fit to data that level off, must *turn over* past the last point — a parabola has nowhere to go but down. The differentiated curve then predicts $dV/dt < 0$ inside or just beyond the data range, implying, absurdly, that the tumor shrinks. The downstream regressor, handed these corrupted pairs, then "discovers" a law with the wrong sign or a spurious singularity. The cure is to use a smoother that respects the qualitative shape — monotone, saturating — rather than one free to oscillate, and to treat the derivative-estimation step as the delicate operation it is rather than a routine preprocessing detail.

12.4 From discovery to a validation pipeline

The naive hope — that a symbolic regressor, handed sparse noisy data, simply outputs the correct law — does not survive contact with reality. In the project's controlled experiments the regressor reliably recovered a Gompertz-like law only in the data-rich, low-noise regime; under realistic noise and sparsity it often returned a *surrogate* that fit the saturation phase while violating the dynamics elsewhere — a recurring example being $dV/dt = \ln(C - V)$, which we examine below. The response is not to abandon discovery but to surround it with discipline. The pipeline has four stages.

1. Generate candidates. Use symbolic regression only as a *candidate generator*, producing a shortlist of plausible right-hand sides $f_1(V), \ldots, f_M(V)$ along with their complexity scores. Do not trust its top choice; trust only that the truth is somewhere on the list.

2. Filter by mathematical and biological plausibility. Reject candidates that violate properties any legitimate growth law must have. These deterministic checks — no human or machine judgment required — include: the law should be defined and finite on the relevant range of V; it should admit a positive stable equilibrium (a carrying capacity), i.e. a $V^\star > 0$ with $f(V^\star) = 0$ and $f'(V^\star) < 0$; and it should be parsimonious. A candidate like $dV/dt = \ln(C - V)$ fails on inspection: it is undefined for $V \geq C$ and does not vanish as $V \to 0$, so although it mimics saturation it is biologically untenable.

3. Refit each survivor by trajectory fitting. For each candidate that passes the filters, do *not* judge it by how well it fit in the derivative space (where the corrupted pairs live). Instead, treat the candidate as an ODE, solve it forward (numerically, Chapter 2), and refit its constants to the *original* data by minimizing trajectory error, exactly the nonlinear least-squares problem of Chapter 5. Rank candidates by this trajectory fit and, above all, by held-out forecast error.

4. Optionally, explain and rank with a language model. Only after the deterministic filtering and trajectory refitting — never as a substitute for them — a large language model may be used to rank the surviving candidates and articulate why one is preferable, a role we scrutinize in Chapter 14. The model's judgment is advisory; it cannot override a mathematical filter.

Remark 12.1 (Separating a convex subproblem)**.** The pipeline embodies a strategy from Chapter 4: isolate the convex part of the problem and solve it exactly. Choosing the functional *form* of f is the

hard, nonconvex, combinatorial search — delegated to the regressor as candidate generation. But once a form is fixed, fitting its *constants* to trajectory data is an ordinary (if nonlinear) least-squares problem, solved reliably by the methods of Chapter 5. Stage 3 is exactly this separation: do not let the unreliable search also estimate the constants from noisy derivatives; re-estimate them cleanly against the trajectory.

Remark 12.2 (Why the ranking can flip — a methodological point)**.** The reason stage 3 matters is that a candidate can win in derivative space and lose in trajectory space. The surrogate $\ln(C - V)$ may fit the $(V, dV/dt)$ pairs well, because most of those pairs sit near saturation where it mimics the truth. But when integrated into a trajectory and compared against the data over the whole range — especially the early growth it gets wrong — it forecasts poorly, while a correctly-shaped law that fit the derivative pairs slightly worse forecasts much better. This *ranking flip* between derivative-space fit and trajectory-space fit is itself a finding: it demonstrates that selecting discovered equations by their apparent fit to estimated derivatives can systematically prefer biologically wrong laws, and that trajectory-level validation is essential.

12.5 What equation discovery can claim

The pipeline supports a measured claim. Symbolic regression can recover the correct dynamics when the data are rich and clean, and even under noise and sparsity it can propose a useful shortlist of candidates. What it cannot do — on sparse, noisy tumor data — is reliably output the single correct law unaided. The value added by the surrounding apparatus is exactly the discipline of Chapter 11: filter for plausibility, validate on held-out forecasts, and never mistake a good fit in the wrong space for a correct discovery. This framing, neither dismissive nor over-promising, is the stance we carry into the case study and the discussion of language models that follow.

Exercises

Exercise 12.1 (*Right framing*). Explain why, to discover the law $dV/dt = f(V)$, one should fit a function from V to dV/dt rather than from t to V. What does fitting $t \mapsto V$ actually recover, and why does the $(V, dV/dt)$ relationship generalize where the (t, V) one does not?

Exercise 12.2 (*Noise amplification*). Two consecutive measurements of V, taken a time h apart, each carry independent noise of standard deviation s. Show that the finite-difference derivative estimate $(V_2 - V_1)/h$ has noise standard deviation $s\sqrt{2}/h$, and explain why small h (closely spaced points) makes the estimate *worse*.

Exercise 12.3 (*Two families*). Contrast genetic programming and sparse regression as approaches to equation discovery. What does each search over, what does each guarantee or fail to guarantee, and in what situation would you prefer each?

Exercise 12.4 (*Library limits*). Sparse regression discovers an equation as a sparse combination of library terms. Explain why this is reliable when the true terms are in the library but useless when they are not. What does this imply about the importance of choosing the library?

Exercise 12.5 (*Smoothing parameter*). Explain qualitatively what happens to the estimated derivative when a smoothing spline's smoothing parameter is set far too small, and when it is set far too large. Why is it sensible to tie the parameter to the noise level?

Exercise 12.6 (*The overshoot*). A degree-2 polynomial is fit to six measurements of a saturating growth curve. Explain why the fitted parabola must eventually decrease, and why this corrupts the estimated $(V, dV/dt)$ pairs for the discovery step.

Exercise 12.7 (*Filtering a surrogate*). The candidate $dV/dt = \ln(C - V)$ is proposed. Identify two distinct reasons it should be rejected by a biological plausibility filter. *(Consider its domain, and its behavior as $V \to 0$.)*

Exercise 12.8 (*Separating the convex part*). Explain the strategy of using the regressor to choose the functional form but re-estimating the constants by trajectory fitting. Which part of the problem is the hard nonconvex search, and which part is the reliable convex (or nearly convex) fit?

Exercise 12.9 (*Ranking flip*). Explain how a candidate law can fit the estimated derivative pairs better than the true law yet forecast the trajectory worse. Why does this argue for ranking candidates by trajectory and forecast error rather than by derivative-space fit?

Exercise 12.10 (*A measured claim*). Write one or two sentences stating what symbolic regression can be trusted to do, and where it fails, on sparse noisy data, of the kind you would be willing to put in a paper's conclusion.

Part V

Applications and Synthesis

Chapter 13

Case Study: Discovering Tumor Growth Laws

This chapter works one problem from beginning to end, drawing the earlier chapters' tools together. It follows a real undergraduate research project — the one whose fragments have appeared throughout these notes — from its first formulation through its dead ends to its conclusions. The missteps are kept in deliberately, because a finished result hides the reasoning that produced it. Every tool here has appeared in an earlier chapter; what is new is seeing them work together on a single problem.

13.1 The question and the plan

The project began with a clean hypothesis. Physics-informed neural networks (Chapter 10) were said to regularize inverse problems; perhaps they could recover tumor growth parameters from sparse noisy data better than the classical Levenberg–Marquardt baseline (Chapter 5). The plan was a factorial experiment: generate synthetic tumor trajectories at several levels of data sparsity ($N \in \{3, 5, 10, 20\}$ points) and noise (1% to 20%), fit them with both methods, and find the regime — if any — where the PINN's physics regularization won.

To make the comparison fair, every result was to come from *fixed* synthetic datasets generated once from a fixed random seed, rather than freshly randomized at each run. This discipline, established early after an initial version produced irreproducible comparisons, is the foundation of everything that followed: without it, no two numbers could be compared. The synthetic setting was itself a deliberate choice. Because the data were generated from a known law with known parameters, the project could measure not merely *fit* but *correctness* — the gap between the recovered parameters and the truth — which is impossible with real data where the truth is unknown. A synthetic benchmark trades realism for the ability to grade the answer, and for a methodological study that trade is the right one.

13.2 First findings: the regularization effect, and its limits

The early results were encouraging and subtle. In the dense, high-noise regime the PINN and the classical method fought the same noise to a draw. But in the *sparse*, high-noise regime — few points, much noise — the physics loss behaved as advertised: it anchored the network between the scarce points, preventing it from chasing noise, and recovered parameters more stably than the unconstrained fit. This was the hoped-for regularization effect, observed in exactly the data-starved regime where it would matter clinically. In the language of Chapter 3, the physics constraint was

buying a reduction in variance, and that reduction was most valuable precisely where variance was largest — when the data were too sparse to pin the trajectory down on their own.

Two complications tempered the success. First, a question of metrics: the student initially gauged robustness by the training loss, and had to be redirected (Chapter 11) to measure what actually matters — parameter recovery error against the known truth, and forecast error on held-out later times. A model can drive its training loss to zero by fitting noise; only held-out performance reveals whether it learned the dynamics. Second, the physics weight λ proved maddening to set, having a noticeable effect only at values around 10^{-3}, far from the order-one weights intuition suggested. The resolution, traced in Section 10.4, was that the data and physics losses lived on vastly different numerical scales, so the bare λ was not the effective weight; normalizing the loss terms restored its intuitive meaning.

Remark 13.1 (A false lead, instructive in hindsight)**.** Considerable effort went into whether the framework's automatic differentiation might be corrupting the physics gradient. It was not: automatic differentiation is exact (Chapter 8), with no step-size error to blame. Ruling this out — on the strength of understanding what AD actually does — redirected attention to the true culprit, the loss scaling. Time spent understanding one's tools is rarely wasted; it eliminates whole categories of phantom explanations.

A further subtlety emerged in the search for a single best λ: no one value simultaneously recovered both parameters well. A weight that pinned down the growth rate left the carrying capacity wildly off, and vice versa. Rather than a nuisance, this became a genuine finding — a symptom of the identifiability ridge (Section 11.3), on which the two parameters trade off so strongly that no single regularization can fix both at once. The profile-likelihood and sensitivity tools of Section 11.4 give the formal vocabulary for what the project saw empirically: the data simply did not carry enough independent information about a and K separately, and no amount of tuning a scalar weight could manufacture it.

13.3 The pivot: from benchmarking to discovery under misspecification

The project's center of gravity then shifted, and the shift is the intellectual crux. Benchmarking two fitting methods *under a fixed, correct Gompertz law* is a narrow question. The richer and more revealing question is what happens when the assumed law is *wrong* — when the tumor does not obey Gompertz dynamics at all, as no real tumor exactly does. This reframing turned a methods comparison into a study of *misspecification*, the theme of Chapter 11.

The experimental design followed naturally. Generate data from a *secret* law — a Richards or von Bertalanffy model, not Gompertz — and ask what the Gompertz-based methods do with it. The families were matched to share carrying capacity and initial volume (Section 2.6), so their trajectories were genuinely hard to tell apart over a realistic window. The predicted outcome, borne out in the data: the misspecified Gompertz fit matched the observations well — sometimes with a *lower* training error than a correctly specified fit — while forecasting the tumor's future size substantially wrong. Goodness of fit was blind to the misspecification; only held-out forecasting exposed it. This is the key result, made concrete: a model can be confidently fit, internally consistent, and wrong.

The result deserves restating in the bias–variance language of Section 3.5, because that is what makes it more than an anecdote. Fitting the wrong family is a high-bias choice whose error does not vanish with more data; the model is systematically wrong in a way no quantity of measurements

repairs. Yet on a single finite, noisy sample, that systematic bias can be smaller than — and hidden by — the noise-driven error of a correctly specified fit. Training error, which sees only the single sample, therefore cannot distinguish the two, and can even rank the biased model higher. Only a test that probes the systematic component — forecasting into a region the misspecified model gets wrong — separates them.

13.4 Attempting discovery: symbolic regression and its discipline

If assuming a law is dangerous, can we instead *discover* it? The project turned to symbolic regression (Chapter 12), and its experience recapitulates that chapter's lessons in miniature.

A sanity check came first, as it should: on dense ($N = 40$), noise-free Gompertz data, could the pipeline recover Gompertz? It could. The regressor returned a law algebraically equivalent to $dV/dt = aV\ln(K/V)$ with the correct constants — confirming that the machinery worked when conditions were ideal. This success is the necessary baseline; a pipeline that failed here would be broken, not merely challenged. The discipline of testing a method first on a problem whose answer is known — and refusing to trust it on hard problems until it passes the easy one — is worth absorbing as a general habit.

Then the difficulties of realism arrived, exactly as Chapter 12 warned. The first stage — estimating dV/dt from noisy samples (Section 12.3) — proved to be the bottleneck. Naive polynomial fits to the saturating trajectory overshot, turning over past the last data point and producing negative growth rates that corrupted the derivative pairs; a discovered "law" from such pairs had a spurious singularity inside the data range. Switching to a smoothing spline helped but introduced its own sensitivity to the smoothing parameter, which had to be tied to the noise level. Even then, under realistic noise the regressor frequently preferred a surrogate, $dV/dt = \ln(C - V)$, that mimicked saturation while being biologically untenable — undefined beyond $V = C$ and failing to vanish at $V = 0$.

The resolution was the validation pipeline of Section 12.4: treat the regressor as a candidate *generator*; filter candidates by deterministic plausibility checks that reject the likes of $\ln(C - V)$; refit each survivor by integrating its ODE and minimizing *trajectory* error against the original data; and rank by held-out forecast error rather than by fit in the noisy derivative space. The payoff was the ranking flip (Section 12.4): the surrogate that won in derivative space lost in trajectory space to a correctly-shaped law, demonstrating that selection-by-derivative-fit systematically prefers wrong equations and that trajectory-level validation is indispensable.

13.5 Sampling design as a first-class variable

A final thread tied the project back to identifiability. If evenly-spaced measurements land mostly near saturation, where the growth laws agree and the rate parameter is poorly determined (Section 11.3), then *where* one measures should matter as much as how much. The project tested this directly, comparing uniform time sampling against an "early-enriched" design placing more points in the fast-growing phase, where the sensitivity $\partial V/\partial a$ is largest (Section 11.4). The enriched design improved recovery of the growth rate in its region — though it could degrade the fit elsewhere, so it was no free lunch but a controlled trade. The lesson generalizes beyond tumors: the information a dataset carries about a parameter depends on the experimental design, and an inverse problem can sometimes be rescued not by a cleverer algorithm but by measuring in the right place. This is among the most practically useful morals of the whole study, because experimental design is often within the scientist's control in a way that the noise level and the underlying biology are not.

13.6 What the experiments establish

The summary is layered. Under misspecification, fixed-law inference can be biased even when the optimizer converges and the fit looks excellent. A discovery-plus-filtering pipeline improves forecast robustness over blindly assuming a law, on controlled synthetic benchmarks. Physics-informed networks help stabilize inference *when the assumed law is approximately correct*, but they are not magic under misspecification, and more physics weight can worsen a wrong assumption. Symbolic regression recovers the right law in the easy regime and proposes useful candidates in the hard one, but cannot, unaided on sparse noisy data, be trusted to output the single correct equation. And what an associated language model may claim is bounded accordingly — the subject of the next chapter.

None of these is a sweeping triumph. The project's contribution is methodological: a demonstration, on data where the truth is known, of how confident inference goes wrong and what discipline guards against it. That is a more durable contribution than a brittle claim of having discovered biology from six points. The argument has a fixed shape: fit, then distrust the fit; validate on what was held out; perturb and refit to expose fragility; compare alternatives fairly; and claim only what the controlled experiment demonstrates.

Exercises

Exercise 13.1 (*Reconstruct the argument*). In a few sentences each, state the project's central misspecification finding, the diagnostic that revealed it, and why that diagnostic succeeded where training error failed.

Exercise 13.2 (*Why synthetic data*). Explain why generating data from a known law with known parameters lets the project measure something that real data cannot. What is given up in exchange, and why is the trade appropriate for a methodological study?

Exercise 13.3 (*Why fix the datasets*). Explain why generating fresh random data at each run made the early PINN-versus-LM comparisons untrustworthy, and how a fixed seed repairs this. What scientific principle is at stake?

Exercise 13.4 (*The λ puzzle*). The physics weight mattered only near 10^{-3}. Explain the cause and the fix, and connect it to the units argument of Chapter 10.

Exercise 13.5 (*No single λ*). Why is the failure of any single physics weight to recover both parameters well a symptom of the identifiability ridge rather than a tuning failure? *(Refer to Section 11.3.)*

Exercise 13.6 (*Bias hidden by noise*). Explain, in the bias–variance language, how a misspecified model can achieve a lower training error than a correctly specified one on a single noisy dataset, even though its systematic error does not vanish with more data.

Exercise 13.7 (*Sanity-check first*). The project tested its discovery pipeline on dense, noise-free data before trusting it on hard data. Explain why this ordering is good scientific practice, and what one would conclude if the pipeline failed even on the easy case.

Exercise 13.8 (*Discovery discipline*). List the four stages of the disciplined discovery pipeline as the project applied them, and state what each stage contributed to rejecting the $\ln(C - V)$ surrogate.

Exercise 13.9 (*Design beats algorithm*). Explain how early-enriched sampling can improve parameter recovery without any change to the fitting method. Give the caveat the project reported, and relate the whole effect to where the informative data lie on the growth curve.

Exercise 13.10 (*The portable moral*). In your own words, state the general methodology the case study illustrates, in a form that would apply to a modeling problem in a different field entirely.

Chapter 14

Large Language Models in Scientific Discovery

Large language models (LLMs) are the systems most people now mean by "artificial intelligence." This chapter asks what they can contribute to mathematical and scientific discovery and where the claims made for them outrun what they can do. An LLM is a fitted model, and the difficulties of inference from data (Chapter 11) constrain it as they constrain any other method.

14.1 What a language model is

A large language model is a neural network (Chapter 7) trained on large amounts of text to predict the next token from the preceding ones. Through that training it acquires a strong command of the patterns of human language, including the language of mathematics and code. It is a parametric function fit to a dataset by the optimization of Part II — a statistical model of text, at very large scale, rather than a reasoning engine of a new kind. The consequences are the ones that hold for any fitted model: it interpolates its training distribution well, extrapolates beyond it unreliably, and a good fit is not understanding.

The architecture. The networks of Chapter 7 were feedforward: input in, output out, through a fixed stack of layers. Language models use a variant, the *transformer*, built around *attention* — a mechanism that lets each position in the input draw on every other, weighting their relevance dynamically. Attention is what lets the model track long-range structure in text, such as a pronoun and its referent or an opening bracket and its close, far better than earlier architectures. The mechanics are beyond our scope; the relevant fact is that a transformer is still a parametric function fit by gradient descent on a loss, differing from a multilayer perceptron in architecture, scale, and training data, not in kind.

14.2 Useful roles in the scientific pipeline

An LLM has several legitimate uses within the modeling workflow.

Proposing and explaining. An LLM can suggest candidate model forms, draft code to fit them, summarize relevant literature, and explain a result in prose. In the discovery pipeline of Chapter 12 the LLM's role was exactly this: *after* deterministic filtering and trajectory refitting, it ranks the

surviving candidates and articulates why one is biologically preferable. For the parts of science made of language, it is a fast and capable assistant.

Acting as a structured prior. More ambitiously, one can ask an LLM to score a candidate equation's plausibility in scientific context, using its absorbed knowledge as a soft prior. This is a testable proposition: one can measure whether LLM-filtered shortlists contain the true law more often than shortlists filtered by parsimony alone. Treated as a hypothesis to be checked rather than a capability to be assumed, it is a legitimate use, and it is how the running project treated the model.

Navigating a combinatorial space. A third role, between the other two, uses the model's fluency to propose *where to look* in a large search space: which functional forms to try, which terms to include in a sparse-regression library (Section 12.2), which experiments might be most informative. Here the model is an informed heuristic guiding a search whose results are still checked by the deterministic machinery. It is useful without needing to be *right*, only better than random, with correctness established downstream.

In each role the LLM is *advisory and verifiable*: it suggests, and deterministic checks and held-out data decide. Its output is an input to the pipeline, not the final authority.

14.3 What language models do not fix

The inflated claim is that a language model, pointed at data, *discovers* the governing law — that one can hand it noisy measurements and receive back correct biology. The difficulties cataloged earlier say otherwise.

The hardness is in the data, not the algorithm. Ill-posedness, poor identifiability, and misspecification (Chapter 11) are properties of the data and the problem. Sparse noisy measurements that fail to determine a parameter do not become determinative because a language model is involved, and a model that fits the observed window while forecasting the future wrongly is no less wrong for having been proposed by an LLM. A direct test of any claimed LLM discovery: ask whether the same data, handed to a classical method, would have determined the answer. If not, the LLM has not recovered information that was never in the data; at best it has supplied a prior, which must be defended like any prior.

Plausibility is not correctness. An LLM can produce an equation that *looks* right — well-formed, appropriately named, biologically evocative — exactly as the surrogate $\ln(C - V)$ looked reasonable until it was checked (Chapter 12). A model trained to produce plausible text is, if anything, especially prone to generating plausible-but-wrong mathematics, since plausibility is what it optimizes. The defense is the one used for any candidate: filter by hard constraints, refit, and validate on held-out forecasts. The model's confidence is not evidence.

Verification stays external. In the validation pipeline the mathematics does the deciding — the domain checks, the stable-equilibrium requirement, the trajectory refitting, the forecast comparison. Bounding the LLM's role this way keeps its fluency from passing a wrong answer off as an accepted one. It is the ordinary requirement that claims be checked against reality, applied to a new and persuasive source of claims.

Remark 14.1 (Traditional methods still win on some tasks)**.** Even within the literature advancing LLM-based discovery, one finds the acknowledgment that on certain tasks classical methods still outperform the language-model approach. A controlled study like the running project maps where the deterministic machinery suffices and the language model adds little, on data where the truth is known.

14.4 Summary

Language models are strong tools for the linguistic and combinatorial work of science — proposing, explaining, drafting, ranking — and using them in those roles is sensible and increasingly standard. They do not remove the mathematical difficulty of inference from data, and treating them as oracles that "discover" laws invites the confident errors described in the preceding chapters. An LLM is best handled like any fitted model: use it where it helps, know what it actually does, and verify its output against data it has not seen.

The two kinds of system this text has treated — the data-driven models of Parts I through IV and the language models of this chapter — are both fitted models, and the same practice governs each: know the model, distrust the fit, test on the unseen.

Exercises

Exercise 14.1 (*An LLM as a fitted model*). Describe a large language model in terms of the concepts of earlier chapters — model, parameters, training, loss. Why does this description help in reasoning about its limitations?

Exercise 14.2 (*Attention, conceptually*). In one or two sentences, describe what the attention mechanism lets a transformer do that a plain feedforward network does not. Why does a transformer nonetheless differ from a multilayer perceptron "in architecture, scale, and training data, not in kind"?

Exercise 14.3 (*Legitimate roles*). List three roles an LLM can usefully play in the scientific modeling pipeline, and for each explain why it is "advisory and verifiable" rather than authoritative.

Exercise 14.4 (*Plausible but wrong*). Explain why a model trained to produce plausible text might be especially likely to generate plausible-looking but incorrect equations. Connect this to the $\ln(C - V)$ surrogate of Chapter 12.

Exercise 14.5 (*What LLMs do not fix*). A colleague claims that feeding sparse noisy tumor data to a sufficiently advanced language model will yield the correct growth law. Using the difficulties of Chapter 11, give a reasoned rebuttal, including the "would a classical method have determined the answer?" test.

Exercise 14.6 (*Testable prior*). Phrase "an LLM is a useful plausibility filter for candidate laws" as a hypothesis that could be tested on synthetic data where the truth is known. What would you measure, and what comparison would settle it?

Exercise 14.7 (*Guiding a search*). Explain the role in which an LLM proposes *where to look* in a search space rather than supplying the answer. Why does this role require only that the model be better than random, with correctness established downstream?

Exercise 14.8 (*Your assessment*). In a short paragraph, state your own position on the role of language models in scientific discovery, citing at least two specific ideas from earlier chapters that inform it.

Exercise 14.9 (*The two kinds of AI*). In what sense are the data-driven models of Parts I–IV and the language models of this chapter both fitted models governed by the same practice? State that practice in one sentence.

Chapter 15

Model Selection and Validation

This chapter is about deciding which model to believe, and how much. Its subject is *model selection* — choosing among competing models — and *validation* — earning the right to trust the choice. The theme that has recurred since Chapter 3, that goodness of fit is not goodness of model, reaches its conclusion here as a working methodology for sound inference.

15.1 The core problem: fit improves with flexibility

The difficulty underlying all model selection is one we have met repeatedly: a more flexible model fits the training data at least as well as a less flexible one, always. Adding parameters can only lower the training error. Therefore training error *cannot* be used to choose model flexibility — it always votes for more. A degree-$(N-1)$ polynomial fits N points exactly, with zero training error and, typically, terrible predictions (Section 3.4). We need criteria that reward fit but penalize flexibility, or that estimate generalization directly. These are the two broad strategies of this chapter: *penalized criteria*, which add an explicit complexity penalty, and *resampling methods*, which measure generalization by holding data out. The latter are more trustworthy; the former are cheaper and sometimes all one can afford.

15.2 Penalized criteria: AIC and BIC

One family of methods adds an explicit penalty for model complexity to the training error. The two best-known are the *Akaike Information Criterion* (AIC) and the *Bayesian Information Criterion* (BIC). Both take the form "goodness of fit plus a penalty for the number of parameters," so that adding a parameter is worthwhile only if it improves the fit by more than the penalty. Schematically, for a model with p parameters fit to N data points,

$$\text{AIC} \sim (\text{misfit}) + 2p, \qquad \text{BIC} \sim (\text{misfit}) + p \ln N,$$

where the misfit term is built from the maximized likelihood (for Gaussian noise, essentially the training sum of squares, specifically $N \ln(\text{SSE}/N)$ up to constants). Lower is better. BIC's penalty grows with the data size N, so it favors simpler models more strongly as data accumulate; AIC's fixed penalty makes it more permissive. The two encode different philosophies — AIC targets predictive accuracy, BIC targets identifying a "true" model assumed to be among the candidates — and they do not always agree.

Where the penalties come from. The forms are not arbitrary. AIC arises from estimating the expected prediction error of a fitted model and correcting the training error's optimism; the correction, to leading order, is proportional to the number of parameters, giving the $2p$ term. BIC arises from a Bayesian approximation to the probability that each candidate model generated the data; carrying out that approximation produces the $p \ln N$ penalty, with the $\ln N$ reflecting how the evidence concentrates as data accumulate. One need not reproduce these derivations to use the criteria, but knowing they are principled — one an estimate of prediction error, the other an approximate model probability — explains both why they take the form they do and why they can disagree: they are answering different questions.

Remark 15.1 (Use these criteria with caution)**.** Penalized criteria are convenient but rest on assumptions — correct noise model, large-sample approximations, and crucially that the candidate set *contains* an adequate model — that the misspecified, small-data settings of this course routinely violate. Under misspecification (Chapter 11) the likelihood-based misfit can rank a wrong model above a right one, and no parameter-counting penalty repairs that, since the problem is the *form* of the model, not its parameter count. AIC and BIC are useful summaries, not substitutes for the direct test of generalization we turn to next. Treat a model-selection number as evidence, not proof.

15.3 Cross-validation and held-out forecasting

The most direct and trustworthy approach estimates generalization by *measuring* it: hold out data, fit on the rest, and test on what was held out. *Cross-validation* systematizes this by rotating which subset is held out and averaging the results, so that every point serves once as a test point.

Several variants trade off differently. In *k-fold* cross-validation the data are split into k groups; each group is held out in turn while the model is fit on the other $k-1$, and the k held-out errors are averaged. *Leave-one-out* cross-validation is the extreme $k = N$, omitting a single point each time — maximally data-efficient but requiring N refits. *Time-series* cross-validation respects temporal order, always training on earlier data and testing on later, which is the right choice when the goal is forecasting and when using the future to predict the past would be cheating. The general principle is the same across variants: the held-out error estimates performance on data not used in fitting, which is what we actually care about.

For the time-series problems of this course the most meaningful version is *held-out forecasting*, used throughout the applied chapters: fit on an early window, predict a later window hidden during fitting, and score the forecast. This directly measures the quantity of scientific interest — the ability to predict the future — and, as Chapters 11 and 13 showed, it succeeds at exposing both poor identifiability and misspecification precisely where training error and even penalized criteria fail. A misspecified model that fit the past beautifully forecasts the future poorly, and the held-out test sees it plainly. Among the diagnostics in this chapter, held-out forecasting is the one to reach for first.

The bootstrap. A related resampling idea, the *bootstrap*, estimates the variability of a fitted quantity by refitting on many resampled versions of the data (drawn, with replacement, from the original sample) and observing how much the estimate varies. Applied to the running example, a bootstrap over noise realizations is exactly what reveals the identifiability ridge: the spread and the tight anticorrelation of the resampled $(\hat{a}, \hat{K})$ estimates (Section 11.3) are a bootstrap diagnosis of how poorly the data determine the parameters. Resampling, in this sense, is the empirical counterpart of the sensitivity analysis of Section 11.4: both quantify how much the data actually constrain the answer.

15.4 The noise floor: a model can fit too well

A complementary check uses the noise level as a yardstick. If the measurements carry noise of variance σ^2_{noise}, then a model that has captured the signal — and nothing more — should leave residuals of about that size; its mean squared error should sit near the noise floor, not far below it. An MSE *far below* the noise floor is not a triumph but a warning: the model has fit the noise itself, which a correct model of the signal cannot do (Chapter 3). In the running project this "too good to be true" diagnostic complemented forecasting: a fit whose training error fell well under the noise floor was overfitting, however flattering its appearance. Comparing achieved error against the known noise level turns an abstract worry into a quantitative check, and it is available whenever the noise level can be estimated — from replicate measurements, from instrument specifications, or from the residuals of a trusted fit.

15.5 Reproducibility as part of correctness

A result that cannot be reproduced is not yet a result. The methodological spine of the case study — fixing the random seed, generating canonical datasets once, loading the same data for every method (Chapter 13) — is not bureaucratic tidiness but part of what makes a comparison *mean* anything. When methods are compared on different random data, differences between methods are confounded with differences between datasets, and no conclusion is warranted. Reproducibility extends to reporting: stating the seeds, the data sizes, the noise levels, and the number of restarts, so that another person — or oneself, months later — can regenerate every number. Good science is reproducible, and reproducibility is a property one designs in from the start, not bolts on at the end.

15.6 The capstone: a methodology for sound inference

We can now state the methodology as a consolidated practice. Confronted with the task of modeling a system from limited data:

1. **Be explicit about the model and its assumptions.** Know whether it is mechanistic or phenomenological, what it commits to, and how it could be wrong.

2. **Fit responsibly.** Use an appropriate optimizer; for nonconvex problems, multi-start. Remember that convergence is not correctness.

3. **Never trust training error.** It measures fit, not generalization, and always rewards flexibility.

4. **Validate on held-out data, especially by forecasting.** This is the primary diagnostic for identifiability failure and misspecification alike.

5. **Probe stability.** Refit on perturbed or resampled data; if the parameters scatter wildly or trace a ridge, they are not determined.

6. **Compare against the noise floor.** A fit far below it is overfitting; a fit far above it is underfitting.

7. **Compare alternatives fairly.** Several models, judged by forecast rather than fit; if the data do not select among them, say so.

8. **Make it reproducible.** Fix seeds, freeze datasets, report enough to regenerate every result.
9. **Claim only what you have shown.** Distinguish what the controlled experiment demonstrates from what you hope is true, and resist the inflation of either classical or AI methods beyond the evidence.

This is less a recipe to be applied mechanically than a working stance: skeptical of one's own fits, attentive to what the data can and cannot support, and explicit about uncertainty. The optimizers, networks, discovery pipelines, and language models of the preceding chapters are tools, and a tool produces a reliable result only under the discipline that checks it.

Exercises

Exercise 15.1 (*Why not training error*). Explain in one paragraph why training error cannot be used to select model flexibility, and give a concrete example where minimizing it would lead you badly astray.

Exercise 15.2 (*AIC versus BIC*). State the schematic form of each criterion and explain how their penalties differ. In what circumstance do they tend to disagree, and why might BIC favor a simpler model than AIC as data accumulate?

Exercise 15.3 (*Where the penalties come from*). Explain, without reproducing the derivations, what question AIC answers and what question BIC answers, and how this difference accounts for their different penalty terms and their occasional disagreement.

Exercise 15.4 (*Designing cross-validation*). For tumor data measured at days 2, 5, 8, 11, 14, 17, 20, 30, and 40, describe a held-out forecasting scheme and explain why it measures something training error cannot. Why is time-series cross-validation more appropriate here than a random split?

Exercise 15.5 (*Variants of cross-validation*). Contrast k-fold, leave-one-out, and time-series cross-validation. For a small forecasting problem, which would you choose and why?

Exercise 15.6 (*Reading the noise floor*). Measurements have noise variance 1600. Model A achieves training MSE 1700; model B achieves 150. Interpret each in relation to the noise floor, and say which you would be suspicious of and why.

Exercise 15.7 (*The bootstrap and the ridge*). Explain how a bootstrap over noise realizations would reveal the identifiability ridge of the running example. What feature of the resampled $(\hat{a}, \hat{K})$ estimates is the diagnosis?

Exercise 15.8 (*Misspecification and AIC*). Explain why AIC and BIC can fail to detect a misspecified model, and why held-out forecasting succeeds where they fail. *(Refer to Chapter 11.)*

Exercise 15.9 (*Reproducibility*). List four specific things one should fix or report so that a model comparison can be reproduced, and explain why comparing methods on differently-generated data confounds the comparison.

Exercise 15.10 (*The write-up*). You have fit three growth laws to a sparse dataset; their forecasts disagree and the data do not clearly favor one. Drawing on the capstone methodology, describe what you would report, what you would refuse to claim, and what additional data or experiment would most improve the situation.

Appendix A

Solution Sketches

These sketches indicate the intended approach and answer for each exercise. They are deliberately concise; a full written solution should expand the reasoning and show intermediate steps. Discursive exercises (those asking for explanations or examples) are answered with the key points an adequate response should contain.

Chapter 1

1.1. (a) Forward: known weather model, computing tomorrow's temperature. (b) Inverse: data are radiation measurements, estimating the decay constant. (c) Forward: computing a trajectory from known initial conditions. (d) Inverse: data are stretch–weight pairs, estimating the spring constant.

1.2. As $V \to K$, the factor $(1 - V/K) \to 0$, so $dV/dt \to 0$: growth halts and the curve levels off. Above K the rate is negative, pushing V back down, so K is a stable ceiling.

1.3. The five points may lie along an identifiability ridge: different parameter pairs produce nearly identical curves over the observed range, so both students fit well while disagreeing on parameters — a preview of poor practical identifiability (Chapter 11).

1.4. No. Although $\ln(K/V) \to \infty$ as $V \to 0^+$, the factor $V \to 0$ faster: $V \ln(K/V) \to 0$. (Set $V = Ke^{-s}$; then $Kse^{-s} \to 0$ as $s \to \infty$.) The growth rate vanishes at $V = 0$, so the singularity is removable and the model is well behaved near zero.

1.5. Open-ended. Strong answers name a specific domain: sparse (few patients in a trial), noisy (sensor readings), confidently-wrong (forecasting under a wrong transmission model).

1.6. Solving, $x_2 = (y_2 - y_1)/0.001$ and $x_1 = y_1 - x_2$. An error of $+0.001$ in y_2 changes x_2 by 1 and x_1 by -1. The combination $x_1 + x_2 = y_1$ is well determined; $x_1 - x_2$ is not. This is the linear prototype of the ridge.

1.7. The forward map evaluates a smooth rule, so it is single-valued and continuous. The inverse map can be many-to-one (distinct parameters giving identical observations) and discontinuous (tiny data changes forcing large parameter changes), as in Example 1.2.

1.8. Open-ended. For a commute estimate: the model is a rule mapping conditions to travel time; the data are past commutes; the fitting procedure is the mental averaging that tunes the rule.

Chapter 2

2.1. $\nabla L = (2(\theta_1 - 3),\ 8(\theta_2 + 1))$, zero at $(3, -1)$. Hessian $\mathrm{diag}(2, 8)$, positive definite, eigenvalues 2 and 8, condition number 4.

2.2. With $\bar{x} = 1$, $\bar{y} = 7/3$: slope $= \sum(x_i - \bar{x})(y_i - \bar{y}) / \sum(x_i - \bar{x})^2 = 2/2 = 1$; intercept $= 7/3 - 1 = 4/3$. Line $y = 4/3 + x$.

2.3. $\nabla f = (2\theta_1 + 3\theta_2,\, 3\theta_1)$; at $(1, 1)$ this is $(5, 3)$. Steepest-ascent unit direction $(5, 3)/\sqrt{34}$; rate of increase $\sqrt{34} \approx 5.83$.

2.4. Differentiate (2.5) via the chain rule to recover $dV/dt = aV \ln(K/V)$; and $V(0) = K \exp(\ln(V_0/K)) = V_0$.

2.5. Gompertz $V^\star = K$; Richards $V^\star = K$; von Bertalanffy $\alpha V^{2/3} = \beta V$ gives $V^\star = (\alpha/\beta)^3$. With $\alpha = 4, \beta = 0.4$: $(10)^3 = 1000$.

2.6. With $V = K(1-s)$, $g \approx Ks - \frac{1}{2}Ks^2$; to first order $g \approx Ks = (K - V)$, the linear relaxation, with quadratic correction $-\frac{1}{2}Ks^2$.

2.7. $\mathrm{Var}(y_i \mid V(t_i)) = \mathrm{Var}(V(t_i)\sigma\varepsilon_i) = \sigma^2 V(t_i)^2$. Absolute uncertainty $\sigma V(t_i)$ grows with size; relative uncertainty σ is constant.

2.8. $g'(V) = a(\ln(K/V) - 1) = 0$ gives $V = K/e \approx 0.368K$. For $K = 1000$ this is $V \approx 368$, reached in the middle of the rise (early-middle, since the trajectory starts at $V_0 = 10$).

2.9. $\mathrm{Var}(\frac{1}{2}(X + Y)) = \frac{1}{4}(2\sigma^2) = \sigma^2/2$; $\mathrm{Var}(X - Y) = 2\sigma^2$. The average has the smaller variance, which is why averaging repeated measurements reduces noise.

2.10. $V'(10) = 0.4 \cdot 10 \cdot (1 - 10/1000) = 3.96$, so $V(1) \approx 10 + 3.96 = 13.96$.

Chapter 3

3.1. Open-ended; e.g. a high-degree polynomial through a few points, where the "noise" is measurement error it interpolates.

3.2. Residuals $(0.1, -0.1, 0.2, -0.2)$; SSE $= 0.10$; MSE $= 0.025$.

3.3. MAE $= 0.15$, max error $= 0.2$. With an outlier (20 for 8), the MSE inflates most (squares the large residual), the max error jumps to the outlier's residual, and the MAE rises least — so MAE is most robust.

3.4. A degree-$(N-1)$ polynomial has N coefficients and interpolates N points exactly (the Vandermonde system is invertible for distinct inputs), giving zero training error for *any* data — so training error says nothing about the right degree.

3.5. (a) bias (too rigid); (b) variance (too flexible for the data); (c) bias (wrong family, systematic error); (d) variance (over-flexible on little data).

3.6. Train on days ≤ 20, forecast 30 and 40. It tests genuine extrapolation into the future — the quantity of interest — on points not used in fitting.

3.7. MSE 200 is far below the noise floor 1600: the model is fitting noise, since a correct model of the signal cannot beat the noise variance. Suspicion, not satisfaction, is warranted.

3.8. (a) mechanistic (good for interpretation/extrapolation when correct); (b) phenomenological (flexible, poor extrapolation); (c) mechanistic; (d) phenomenological.

3.9. The validation set is used to choose the model, so the chosen model is the one that happened to do best on it — making validation error optimistic. An untouched test set gives an unbiased final number. Repeatedly tuning against the test set lets the model adapt to it, so it too becomes optimistic.

3.10. The irreducible error is the 10% multiplicative measurement noise itself. No growth model predicts the random draw ε_i, so no model can drive the error below the noise variance.

Chapter 4

4.1. Update $\theta_{k+1} = \theta_k - 2\eta(\theta_k - 5)$. With $\eta = 0.1$: $\theta_1 = 1.0$, $\theta_2 = 1.8$, $\theta_3 = 2.44$. Converges to 5.

4.2. $\theta_{k+1} - 5 = (1 - 2\eta)(\theta_k - 5)$; converges iff $|1 - 2\eta| < 1$, i.e. $0 < \eta < 1$. At $\eta = 1$ the factor is -1 (persistent oscillation, no decay); beyond it, divergence.

4.3. $\nabla L = (2\theta_1, 200\theta_2)$, Hessian $\mathrm{diag}(2, 200)$, condition number 100. The steep θ_2 direction forces $\eta < 2/200 = 0.01$, which makes the θ_1 direction converge slowly.

4.4. (a) convex; (b) convex; (c) not convex; (d) convex; (e) not convex (a saddle, Hessian indefinite).

4.5. E.g. $f = \theta_1^2 - \theta_2^2$, saddle at the origin. Stochastic gradient noise can knock the iterate off the saddle's unstable direction; a strict local minimum offers no descent direction to exploit.

4.6. Newton on θ^4: step $= -L'/L'' = -(4\theta^3)/(12\theta^2) = -\theta/3$, so $\theta_{k+1} = \frac{2}{3}\theta_k$; from $1 \to 0.667$. One GD step with $\eta = 0.1$: $1 - 0.1 \cdot 4 = 0.6$. Both decrease θ toward the minimum at 0.

4.7. A line search chooses the step length by (approximately) minimizing the loss along the descent direction, or accepting any step that decreases it sufficiently; it removes the need to guess a global learning rate. The cost is extra loss evaluations per step — negligible for small problems, potentially prohibitive when each evaluation is expensive.

4.8. Optimization error is the gap to the loss minimizer (reduced by better/longer optimization); statistical error is the gap from the minimizer to the truth (set by data, irreducible by optimization). Diagnostic: if many starts reach the same loss but wrong parameters, the error is statistical; if different starts reach different losses, optimization is not yet solved.

4.9. The second ($\kappa = 10^4$) is far harder, by a factor of order $\kappa = 10^4$ in iterations. Both are in $\mathbb{R}^2$, so dimension is equal; conditioning, not dimension, governs the difficulty.

Chapter 5

5.1. Φ has rows $(1, x_i, x_i^2)$ for $x = -1, 0, 1, 2$; then solve $\Phi^\top \Phi\, \theta = \Phi^\top y$.

5.2. The condition $\Phi^\top(\Phi\theta - y) = 0$ says the residual is orthogonal to every column of Φ, hence to the model's subspace. So $\Phi\hat{\theta}$ is the orthogonal projection of y onto that subspace, and the residual is the perpendicular from y to it.

5.3. $\log p = -\frac{1}{2\sigma^2}\sum(y_i - m_i)^2 + \text{const}$; maximizing drops the constant and the positive factor, leaving minimization of $\sum(y_i - m_i)^2$. The variance scales the objective but not its minimizer.

5.4. Weighting by inverse variance gives $\sum_i (y_i - m_i)^2 / V(t_i)^2$; noisier (larger) points count less. This is the maximum-likelihood objective under multiplicative noise.

5.5. Absolute-error loss corresponds to Laplace (double-exponential) noise, which is heavier-tailed than Gaussian; it is the wiser choice when the data contain outliers, since it down-weights large residuals.

5.6. Single parameter: $\theta_{k+1} = \theta_k - \frac{\sum_i r_i r_i'}{\sum_i (r_i')^2}$. For $m = \theta x_i$, $r_i' = x_i$, and one step lands on the linear least-squares solution $\theta = \sum x_i y_i / \sum x_i^2$.

5.7. As $\mu \to 0$, the step $\to$ Gauss–Newton (large, curvature-aware); as $\mu \to \infty$, $\to$ small gradient descent. Increasing μ after a failed step shrinks the trust region toward the safe gradient direction.

5.8. Nonconvexity means LM lands in a basin set by the start; a single start may find a bad basin. Many starts sample many basins. For a convex problem there is one basin, so multi-start is unnecessary.

5.9. Initialize K near the largest observed volume and a from the early slope; working in $\log a, \log K$ keeps them positive and spans their ranges evenly, improving conditioning and robustness.

5.10. Near a good fit the residuals r_i are small, so the dropped term $2\sum_i r_i \nabla^2 r_i$ is negligible; far from a fit the residuals are large and the approximation poor, but the LM damping μI keeps the step conservative until the fit improves.

Chapter 6

6.1. $\mathbb{E}[\widehat{\nabla L}] = \frac{1}{|B|}\sum_{i\in B}\mathbb{E}[\nabla \ell_i] = \frac{1}{N}\sum_i \nabla \ell_i = \nabla L$ for a uniform batch. Unbiasedness means the steps are correct on average, so descent works despite per-step noise.

6.2. Variance falls like $1/|B|$, so the standard deviation falls like $1/\sqrt{|B|}$ (independent terms, variances add). Larger batches give smoother but costlier steps; smaller batches, noisier but cheaper steps.

6.3. Momentum accumulates the consistent along-valley gradient component and cancels the alternating cross-valley one, so the iterate accelerates along the shallow eigen-direction while oscillation along the steep one is damped.

6.4. The large-gradient parameter is divided by a large $\sqrt{\hat{v}}$, shrinking its step; the small-gradient one by a small $\sqrt{\hat{v}}$, enlarging its step. Both end up well scaled.

6.5. m and v start at zero, so early averages are biased toward zero; dividing by $1-\beta^k$ (near zero at first) inflates them to compensate. As k grows the factors approach one and the correction fades.

6.6. Levenberg–Marquardt: two parameters, few data, cheap full gradient, and the limit is data information, not optimization speed. Adam is for many parameters and large data.

6.7. A constant step keeps injecting gradient noise, so the iterate jitters around the minimum; decreasing the step shrinks the jitter so it can settle. Decreasing too fast stalls progress before the minimum is reached.

6.8. Newton divides by curvature (the Hessian); Adam divides by $\sqrt{\hat{v}}$, the observed gradient scale — a diagonal, data-driven stand-in for curvature. It is far cheaper (no Hessian) and cruder (diagonal only, and based on gradient magnitude rather than true second derivatives).

6.9. $10{,}000/100 = 100$ updates per epoch; 50 epochs give 5000 updates. Progress tracks the number of *updates* (parameter steps), not epochs, since each update is one move of the optimizer.

Chapter 7

7.1. $z = 2 - 3 + 0.5 = -0.5$; output $\tanh(-0.5) \approx -0.462$.

7.2. Composing affine maps is affine: $W_2(W_1x + b_1) + b_2 = (W_2W_1)x + (W_2b_1 + b_2)$. Any depth collapses to one affine map, so nonlinearity is essential to flexibility.

7.3. On $x = 0$: hidden pre-activations $z = (0, 1)$, activations $(\tanh 0, \tanh 1) = (0, 0.762)$, output $0 + 0.762 = 0.762$. The bias in the second neuron is what makes the two differ.

7.4. Layer 1: $16 \cdot 1 + 16 = 32$. Layer 2: $16 \cdot 16 + 16 = 272$. Output: $1 \cdot 16 + 1 = 17$. Total 321.

7.5. Softplus output is positive, so $V > 0$ and $\ln(K/V)$ is defined, and it avoids the $V = 0$ singularity. A linear output could go negative, making the dynamics undefined and the prediction unphysical.

7.6. The physics loss contains $d\hat{V}/dt$; a kinked activation (ReLU) gives a discontinuous derivative, which makes the residual misbehave, whereas smooth tanh gives a smooth derivative.

7.7. Each hidden neuron contributes one adaptive bump/ramp; the output layer combines them. Unlike a fixed polynomial basis, the network *learns* the basis (positions and scales), which adds flexibility and the capacity to overfit.

7.8. (i) It is an existence result, not a construction; (ii) the required network may be enormous, and training is nonconvex, so the good weights may not be found even though they exist.

7.9. The theorem bounds error only on the bounded set where data live; it says nothing outside it. A network can fit the training window and predict anything beyond it, which is why forecasting (extrapolation) is the demanding test.

7.10. With 10,000 parameters and six points the network interpolates the noise (severe overfitting) and forecasts nonsense. Trustworthiness requires strong regularization — ideally physical constraints (Chapter 10) — or far more data.

Chapter 8

8.1. Error shrinks from $h = 10^{-1}$ to about 10^{-8}, then grows at 10^{-12} as floating-point cancellation dominates (the two function values agree to too many digits). This is the finite-difference dilemma.

8.2. $\partial f/\partial x_1 = x_2 \cos(x_1 x_2) + 2x_1$; $\partial f/\partial x_2 = x_1 \cos(x_1 x_2)$. At $(1, 0)$: $(2, 1)$.

8.3. Forward: $f = 9 \cdot 2 + 2 = 20$. Backward: $\partial f/\partial x_1 = 2x_1 x_2 = 12$, $\partial f/\partial x_2 = x_1^2 + 1 = 10$.

8.4. Forward: ~ 1000 passes (one per input). Reverse: ~ 1 sweep for all inputs. Reversed shape (1 input, 1000 outputs): forward ~ 1, reverse ~ 1000.

8.5. Input dual $(2, 1)$. With $f = x \ln x$, $f' = \ln x + 1$, so the output dual is $(2 \ln 2, \ln 2 + 1) \approx (1.386, 1.693)$.

8.6. AD applies exact chain-rule derivatives to elementary operations, propagating exact numerical values — no limit, no step size, no truncation. Hence it cannot produce a step-size-dependent artifact.

8.7. Backpropagation *is* reverse-mode AD specialized to a network; the layered structure lets the backward sweep reuse each layer's stored forward quantities efficiently.

8.8. It yields exact derivatives of the numerically computed trajectory with respect to the parameters — the Jacobian LM needs — even when no closed-form solution exists.

8.9. (i) The derivative of the output with respect to time gives dV/dt for the physics residual; (ii) the derivative of the loss with respect to the parameters drives training. Both are needed and both come from AD.

8.10. Reverse mode stores all intermediate values from the forward pass for the backward pass, so its memory grows with the computation's length; forward mode carries only one derivative alongside each value and needs no such store. We still prefer reverse mode for training because the gradient over many parameters comes in a single sweep.

Chapter 9

9.1. Causes: learning rate far too small; a disconnected/frozen parameter (no gradient reaching it); a bug making the loss constant. Checks: vary the learning rate; inspect gradient magnitudes; verify the loss depends on the parameter.

9.2. A good rate: steady decrease then plateau. Too large: divergence or oscillation. Too small: near-flat slow descent. To detect overfitting one must also watch the held-out (validation) curve, which rises while the training curve keeps falling — invisible to the training curve alone.

9.3. Overfitting. Interventions: more data, or regularization/reduced flexibility. Running the optimizer longer does not help — it lowers training error further, worsening the gap.

9.4. The parameter enters the loss but receives no update (e.g. it was not registered among the optimizer's variables). Verify by checking whether its computed gradient is nonzero and whether the optimizer includes it.

9.5. Identical initial weights make all neurons in a layer compute the same function and receive identical gradients, so they never differentiate (symmetry). Random initialization breaks the symmetry; otherwise the layer behaves like a single neuron.

9.6. Weight decay adds a size penalty, preferring smaller, smoother weights; early stopping halts before the noise is fit; dropout randomly disables neurons so none is over-relied upon, averaging over subnetworks.

9.7. A weight penalty encodes only generic smoothness; a physics constraint encodes specific, correct structure — far stronger *if correct*. It becomes harmful under misspecification, pulling the model toward the wrong law.

9.8. As $\lambda \to 0$ the network ignores the physics and may overfit; as $\lambda \to \infty$ it ignores the data and merely solves the (possibly wrong) ODE. An intermediate value balances fit and physical regularity.

9.9. Initialize K near the largest observed volume, a from the early slope (in logarithms for positivity). It matters more than the network weights because these few parameters are interpretable and are the actual object of interest.

9.10. An ℓ_2 penalty shrinks all weights smoothly toward zero; an ℓ_1 penalty drives many weights exactly to zero, producing a sparse model — the same sparsity principle behind library-based equation discovery (Chapter 12).

Chapter 10

10.1. $L = \frac{1}{N}\sum_i (\hat{V}(t_i) - y_i)^2 + \lambda \frac{1}{M}\sum_j \big(\hat{V}'(\tau_j) - r\hat{V}(\tau_j)(1 - \hat{V}(\tau_j)/K)\big)^2$.

10.2. The physics loss tests the *equation*, which can be evaluated anywhere; the data loss compares to *measurements*, which exist only at sampled times. So physics constrains the network between data points.

10.3. Classical collocation posits a trial function (e.g. a polynomial) and forces it to satisfy the equation at chosen points, solving for its coefficients. In a PINN the trial function is a neural network and the solver is gradient-based optimization.

10.4. Data residual: mm^3, squared $\to \mathrm{mm}^6$. Physics residual: dV/dt in mm^3/day, squared $\to \mathrm{mm}^6/\mathrm{day}^2$. A bare λ multiplies quantities of different scale, so its effective weight depends on units.

10.5. Logarithms enforce positivity for free (any real $\log a$ gives $a > 0$) and improve conditioning: $K \approx 1000$ and $a \approx 0.4$ differ by orders of magnitude, so the raw loss surface is highly elongated while the log surface is rounder.

10.6. Symptom: the physical parameters stay at their initial values while the loss decreases. Check that gradients reach them and that the optimizer updates them.

10.7. A larger λ enforces the wrong (Gompertz) equation more strictly, pulling the network further from the true von Bertalanffy dynamics, so beyond some point recovery worsens.

10.8. Dividing each term by its characteristic magnitude makes both $O(1)$, so λ is a genuine relative weight independent of units; the previously tiny optimal λ was compensating for scale and becomes $O(1)$ after normalization.

10.9. Increasing λ reduces variance (stiffening the fit). When the law is correct this adds little bias and helps; when wrong it adds bias that grows with λ, eventually outweighing the variance reduction.

10.10. An adaptive scheme rescales λ during training to keep the two loss terms' gradient contributions comparable, because the ideal balance early (network far from fitting) differs from that late (refining a good fit).

Chapter 11

11.1. Existence, uniqueness, continuous dependence. Violations: multiple parameter sets fit equally (uniqueness); noise causes large parameter swings (continuity); no parameters fit at all (existence, rare in least squares).

11.2. Structural: distinct parameters give distinct outputs under ideal data. Practical: finite noisy data actually pin parameters down. Example: Gompertz is structurally identifiable but, from five points near saturation, practically unidentifiable in a.

11.3. Individually a and K are poorly determined (they vary along the line); the combination transverse to the line is well determined. The line is the identifiability ridge.

11.4. A small Hessian eigenvalue means the loss barely rises along the corresponding eigenvector, so that parameter combination is weakly constrained. The condition number (largest over smallest eigenvalue) measures the elongation of the loss ellipse, which sets the strength of the anticorrelation between estimates.

11.5. Fix one parameter at various values; at each, re-optimize over the others and record the best loss. A steep profile means the parameter is well determined; a flat profile means it is not — the signature of a ridge.

11.6. The penalty curves the flat floor, giving a unique stable minimizer (reduced variance), but pulls the estimate toward θ_0 (bias). The strength γ sets the bias–variance balance: more γ, less variance and more bias.

11.7. By Example 2.5, each law $\to c(V^\star - V)$ near $V^\star$ ($c = a$ for Gompertz, $r\nu$ for Richards, $\beta/3$ for von Bertalanffy); they differ only at higher order, so data near saturation cannot distinguish them.

11.8. Lower training error reflects goodness of fit, which is blind to misspecification; B may exploit a coincidental match or fit noise. Diagnostic: held-out forecasting, or comparing forecasts across families.

11.9. Early measurements sample the fast-growth region, where $\partial V/\partial a$ is largest, orienting the data across the ridge; evenly-spaced points cluster near saturation (along the ridge), carrying little information about a.

11.10. A wrong family is systematically off, so its bias does not vanish as $N \to \infty$. On a single finite sample, that bias can be smaller than the noise-driven error of a correct model, so training error — seeing only one sample — fails to reveal it.

Chapter 12

12.1. Fitting $V \mapsto dV/dt$ recovers the autonomous law f; fitting $t \mapsto V$ recovers only the particular trajectory. The law generalizes across initial conditions; the trajectory does not.

12.2. $\mathrm{Var}((V_2 - V_1)/h) = 2s^2/h^2$, so $\mathrm{SD} = s\sqrt{2}/h$. Small h inflates the noise even as it reduces truncation bias — the instability of numerical differentiation.

12.3. Genetic programming searches all expressions (in an operator set) by evolution: general but stochastic, no optimality guarantee. Sparse regression selects a sparse combination from a fixed library: fast and reliable *if* the true terms are in the library. Prefer the former when the form is unknown, the latter when good candidate terms are known.

12.4. Sparse regression can only combine library terms, so it is reliable when the truth is expressible in them and useless otherwise — making the library choice decisive.

12.5. Too little smoothing: the spline chases noise, wild derivative. Too much: it over-flattens, missing real features. Tying the parameter to the noise level lets it miss points by about the noise

magnitude — neither chasing nor erasing signal.

12.6. A degree-2 parabola fit to a saturating curve must peak and decrease (it has nowhere else to go), giving $dV/dt < 0$ past the peak — a false negative rate that corrupts the pairs.

12.7. $\ln(C - V)$ is undefined for $V \geq C$ (invalid domain), and $\ln(C - 0) = \ln C \neq 0$, so growth does not vanish at $V = 0$. Both violate biological plausibility.

12.8. Choosing the functional form is the hard nonconvex, combinatorial search (the regressor); fitting the constants of a fixed form to trajectory data is the reliable (nonlinear) least-squares fit. Stage 3 keeps the constants out of the noisy derivative space.

12.9. A surrogate can match the (mostly near-saturation) derivative pairs well yet, integrated over the full range, forecast poorly, while a correctly-shaped law that fit the pairs slightly worse forecasts better. Hence rank by trajectory/forecast error.

12.10. Acceptable claim: symbolic regression recovers the correct law on dense, clean data and proposes useful candidates under noise and sparsity, but cannot, unaided on sparse noisy data, be trusted to output the single correct equation without plausibility filtering and trajectory validation.

Chapter 13

13.1. Finding: a misspecified Gompertz fit matches the data (even with lower training error) yet forecasts the future wrong. Diagnostic: held-out forecasting. It succeeds because it probes the systematic (bias) component that training error, seeing only the fitted window, cannot.

13.2. Known law and parameters let the project grade *correctness* (parameter and forecast error against truth), impossible with real data. It gives up realism, an acceptable trade for a methodological study whose point is to show how inference fails when the truth is known.

13.3. Fresh random data per run confounds method differences with dataset differences, so no comparison is valid; a fixed seed makes all methods face identical data. The principle is controlled comparison.

13.4. The data and physics losses had vastly different scales, so the effective weight was λ times their magnitude ratio; only a tiny λ balanced them. Normalizing the terms restores an interpretable order-one weight (units argument of Chapter 10).

13.5. The two parameters trade off along the ridge, so no single scalar weight can constrain both at once — a structural identifiability limitation, not a tuning failure.

13.6. The misspecified model's bias is systematic and does not vanish with data, but on one noisy sample it can be smaller than the correct model's noise-driven error; training error, seeing only that sample, cannot separate the two.

13.7. Testing on a known-answer (dense, noise-free) case verifies the machinery before trusting it on hard cases. Failure there would mean the pipeline is broken, not merely challenged — a bug to fix before proceeding.

13.8. (1) Generate candidates with the regressor; (2) filter by plausibility ($\ln(C - V)$ fails the domain and $V \to 0$ checks); (3) refit by trajectory integration; (4) optionally rank/explain with an LLM. The filter and the trajectory refit together reject the surrogate.

13.9. Enriched early sampling places points where $\partial V/\partial a$ is large, improving rate recovery, with no change to the fitting method. The caveat: it can degrade the fit elsewhere — no free lunch. The effect is about where the informative data lie on the curve.

13.10. Fit, then distrust the fit; validate on held-out data (especially forecasts); perturb and refit to expose fragility; compare alternatives fairly; claim only what the controlled experiment shows — applicable to any modeling problem.

Chapter 14

14.1. An LLM is a large parametric function (model) whose weights (parameters) are fit by minimizing a next-token loss on text (training). The framing makes its limits intelligible: like any fitted model it interpolates its training distribution and extrapolates unreliably.

14.2. Attention lets each part of the input draw on every other part, weighting relevance dynamically, capturing long-range structure. It is still a parametric function fit by gradient descent — differing from an MLP in architecture, scale, and data, not in the fundamental nature of a fitted model.

14.3. Proposing/explaining, acting as a structured prior, and guiding a search. Each is advisory and verifiable because deterministic checks and held-out data make the final decision, not the model.

14.4. Plausibility is exactly what such a model optimizes, so it readily produces plausible-but-wrong mathematics — as $\ln(C - V)$ looked reasonable until the domain and $V \to 0$ checks failed it.

14.5. The hardness (ill-posedness, identifiability, misspecification) lives in the data and problem, not the algorithm; fluency does not add information. Test: would a classical method have determined the answer from the same data? If not, the LLM has only supplied a prior, to be defended as such.

14.6. Hypothesis: LLM-filtered shortlists contain the true law more often than parsimony-filtered ones. Measure the overlap of each shortlist with the known truth on synthetic data; compare the two filtering strategies.

14.7. The model proposes where to look (which forms, which library terms, which experiments); correctness is established downstream by the deterministic machinery. This requires only that the suggestions beat random, not that they be right.

14.8. Open-ended; a strong answer uses, e.g., the held-out-forecasting discipline and the plausibility-is-not-correctness point to argue for using LLMs in advisory roles with external verification.

14.9. Both data-driven models and language models are parametric functions fit to data, powerful within their training distribution and unreliable beyond it. The shared discipline: know the model, distrust the fit, test on the unseen.

Chapter 15

15.1. A more flexible model cannot have larger training error, so training error always favors more flexibility; e.g. a degree-$(N-1)$ polynomial achieves zero training error on N points yet forecasts terribly.

15.2. AIC $\sim$ misfit $+2p$; BIC $\sim$ misfit $+p \ln N$. BIC's penalty grows with N, favoring simpler models as data accumulate; AIC's is fixed and more permissive. They disagree most at large N, where BIC penalizes extra parameters more heavily.

15.3. AIC estimates expected prediction error (correcting the training error's optimism), giving the $2p$ term; BIC approximates the posterior probability that a model generated the data, giving $p \ln N$. Different questions, hence different penalties and occasional disagreement.

15.4. Train on early days, forecast later ones (e.g. train ≤ 20, test $30, 40$). It measures extrapolation into the future, which training error cannot. A random split is wrong because using later points to predict earlier ones is not the forecasting task and leaks future information.

15.5. k-fold: split into k groups, hold each out in turn. Leave-one-out: $k = N$, maximally data-efficient but N refits. Time-series: always train on earlier, test on later. For a small forecasting problem, time-series CV is most appropriate because it respects causality.

15.6. Model A (MSE 1700) sits just above the noise floor 1600 — it has captured the signal. Model B (MSE 150) is far below the floor — overfitting the noise. Be suspicious of B.

15.7. Resample the noise many times, refit, and observe the spread of $(\hat{a}, \hat{K})$. The wide scatter along a tight downward line (strong anticorrelation) is the bootstrap diagnosis of the ridge.

15.8. AIC and BIC use a likelihood-based misfit that can rank a wrong model above a right one under misspecification, and a parameter-count penalty cannot fix a wrong functional form. Held-out forecasting probes the systematic error directly and so exposes it.

15.9. Fix/report the random seed, the dataset (sizes and noise level), the number of restarts, and the train/test split. Comparing methods on differently-generated data confounds method differences with dataset differences.

15.10. Report all three forecasts and their disagreement; state that the data do not select among the families, so no single law or its parameters can be trusted; refuse to claim a discovered mechanism. Recommend more data, especially earlier (more informative) measurements, to break the tie.

Further Reading

These notes are self-contained, but a reader who wishes to go deeper will find the following pointers useful. They are grouped by theme rather than listed exhaustively, and emphasize accessible entry points over comprehensive surveys.

Foundations and historical background

The use of least squares to recover an orbit from sparse observations originates with Gauss's recovery of Ceres in 1801; the episode is a vivid historical anchor for the inverse-problem viewpoint and is recounted in many histories of statistics and astronomy. For the mathematical foundations of optimization at an undergraduate level, any standard text on numerical optimization develops gradient descent, Newton's method, and the Gauss–Newton and Levenberg–Marquardt algorithms of Chapter 5 with full rigor.

Growth models in biology

The three growth laws of the running example each have a substantial literature. The Gompertz model dates to the nineteenth century and remains widely used in tumor modeling; the Richards (generalized logistic) and von Bertalanffy models extend it with additional shape flexibility. A reader interested in the biological modeling context — how these laws are fit to real tumor data and compared — will find review articles on tumor growth modeling a good starting point, including comparative studies that fit multiple growth laws to experimental data and assess their predictive accuracy. A theme of this course — that such laws are easily confused on limited data — echoes concerns raised in that applied literature.

Identifiability

The distinction between structural and practical identifiability (Chapter 11) is developed carefully in the systems-biology literature on parameter estimation. Readers seeking the precise definitions and the profile-likelihood methods used to diagnose practical identifiability in practice should consult the methodological papers on identifiability analysis for dynamical systems models; these formalize the ridge phenomenon we treated geometrically.

Neural networks and automatic differentiation

For neural networks (Chapter 7) and their training, any modern textbook on deep learning covers the multilayer perceptron, universal approximation, initialization, and the Adam optimizer. The

universal approximation theorem has several forms; the original results establish density of single-hidden-layer networks with suitable activations. Automatic differentiation (Chapter 8) is treated in depth in the specialized literature, which carefully distinguishes forward and reverse modes and explains why backpropagation is reverse-mode AD; a survey of automatic differentiation in machine learning is the natural entry point.

Physics-informed neural networks

Physics-informed neural networks (Chapter 10) were popularized in their modern form in the scientific-computing literature, where the composite data-plus-physics loss and the use of automatic differentiation for the physics residual are developed in detail. The loss-balancing difficulty we discussed — the sensitivity to the weight on the physics term and the role of scaling — is an active topic, with several proposed schemes for adaptively weighting the loss components.

Symbolic regression and equation discovery

Equation discovery (Chapter 12) has two prominent modern strands. Sparse-regression methods identify governing equations by selecting from a library of candidate terms, emphasizing sparsity for interpretability. Genetic-programming-based symbolic regression searches the space of expressions directly and underlies the candidate-generation step in our pipeline. Both bodies of work discuss the central obstacle of estimating derivatives from noisy data and the resulting need for careful smoothing, exactly the bottleneck the case study encountered.

Language models in science

The role of large language models in scientific discovery (Chapter 14) is a fast-moving area. Recent work explores using language models to propose, rank, and explain candidate models, including hybrid pipelines that combine symbolic methods with language-model priors. The measured stance of this course — that such models are useful in advisory, verifiable roles but should not be credited with discovering laws unaided — is consistent with the more careful voices in that literature, which note that traditional methods still outperform language-model approaches on a range of quantitative tasks. The most enthusiastic claims are best approached with skepticism, and any reported discovery checked for held-out validation.

A closing note

A useful exercise: generate a dataset from any of the growth laws here, then try to recover the known truth under noise and sparsity using the methods of these notes. Identifiability, misspecification, and the limits of discovery become concrete when a confident fit forecasts the wrong future on data one generated oneself.