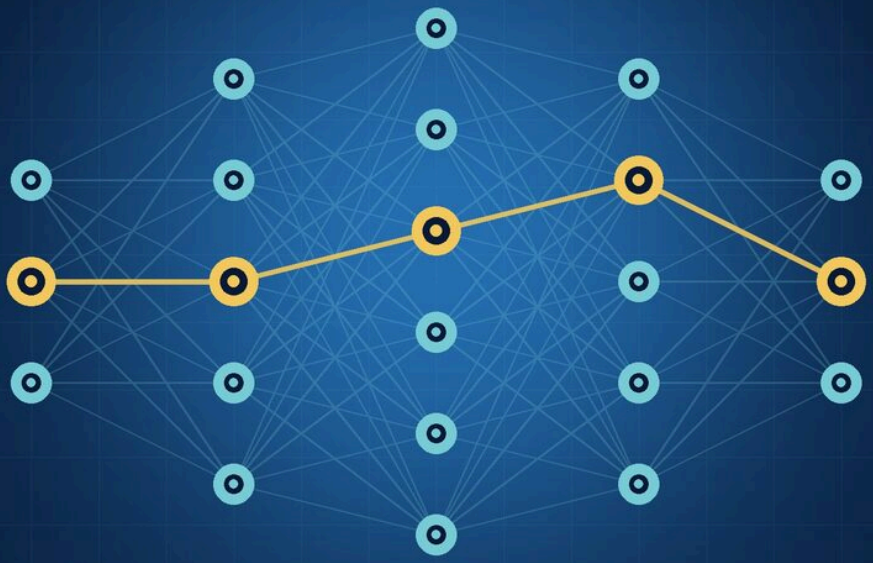


∇L ∂ Σ

MATHEMATICS FOR AI & Machine Learning

Foundations, Theory, and Practice with Python

From linear algebra to large language models

λ

Mathematics for AI and Machine Learning

Foundations, Theory, and Practice with Python

Choral Software

Choral Software

© 2026 Choral Software. All rights reserved.

Mathematics for AI and Machine Learning

Foundations, Theory, and Practice with Python

Choral Software

Copyright

Mathematics for AI and Machine Learning: Foundations, Theory, and Practice with Python

Copyright © 2026 Choral Software

All rights reserved. No part of this publication may be reproduced, distributed, or transmitted in any form or by any means, including photocopying, recording, or other electronic or mechanical methods, without the prior written permission of the publisher, except in the case of brief quotations embodied in critical reviews and certain other noncommercial uses permitted by copyright law.

All source code in this book is original work released for the reader's own use under the terms stated in the accompanying code repository. Third-party software referenced in this book — including NumPy, SciPy, Matplotlib, PyTorch, and scikit-learn — is the copyrighted work of its respective authors and is used here under its own open-source license (BSD, MIT, or Apache 2.0, as applicable). No proprietary or closed-source software, model weights, or datasets are used or required to reproduce any example in this book.

Every dataset used in this book's worked examples and Computational Companion listings is generated synthetically, in code, at runtime — drawn from explicit, seeded random distributions (Gaussian mixtures, synthetic Markov decision processes, synthetic preference and reward data, and similar) rather than loaded from any external corpus, image set, or citation graph. This is a deliberate choice, not an incidental one: it guarantees every example is fully reproducible from the code alone, requires no dataset downloads or licensing review, and involves no third-party data of any kind. See Appendix D for the complete per-chapter listing of what each Computational Companion generates and how.

This book is an independent work. It is not affiliated with, endorsed by, or sponsored by any of the organizations whose open-source software or public datasets are discussed or used herein. Product and company names mentioned in this book may be trademarks of their respective owners and are used for identification purposes only.

The information in this book is provided on an “as is” basis. While every effort has been made to ensure the mathematical derivations and code examples are correct, the author and publisher make no warranties, express or implied, and assume no liability for errors, omissions, or any damages arising from the use of the information contained herein.

First edition.

Published by Choral Software.

Preface

Most people learn the mathematics of machine learning backwards. They learn to call `model.fit()`, get a result, and only later — usually while debugging a training run that refuses to converge, or a confidence interval that turns out to be nonsense — go looking for the math that explains what actually happened. This book is written for that moment, and for the moment before it: for readers who would rather understand the machinery before it breaks.

Three kinds of failures motivate the approach taken here, and each recurs throughout the book as a worked example rather than an anecdote. A gradient descent run diverges or crawls not because the implementation is wrong but because the input features are so poorly scaled that the loss surface's condition number is enormous — a fact you can compute in one line once you know what a condition number is (Chapter 1). A model reports 95% confidence in a prediction it should not be confident about, because the likelihood function implicitly assumed a noise model the data does not satisfy — a fact you can diagnose once you understand maximum likelihood estimation as a modeling choice, not a black box (Chapter 4). A transformer's attention weights behave strangely at long context lengths because of how the softmax's gradient interacts with an unscaled dot product — a fact that falls directly out of a derivative you can write down (Chapter 17). In each case, the fix is easy once the mathematics is not a mystery.

This book therefore treats mathematics and code as one continuous object of study rather than two. Every chapter builds the relevant theory — definitions, theorems, and, where it is pedagogically valuable, proofs — and then immediately implements it: first from scratch in NumPy or SciPy, so that nothing is hidden inside a library call, and then again using the corresponding function in an open-source framework such as PyTorch or scikit-learn, so you can see exactly which line of the library is doing the work you just did by hand. Every code example in this book runs on

open-source tooling only; none requires a proprietary API key or a paid service, and none depends on downloading a third party's pretrained model weights.

The book assumes you have taken a first course in linear algebra, a first course in multivariable calculus, and a first course in probability, and that you can read and write Python comfortably. It does not assume you remember any of that coursework in detail — Part I rebuilds the necessary foundations from the ground up, but does so through the lens of the objects that show up in machine learning (design matrices, loss surfaces, likelihoods, gradients under noise) rather than as an abstract review.

Two audiences shaped every decision in this book. Graduate students will find the proofs and theorem statements they need to read primary-source papers without taking the mathematics on faith. Industry practitioners will find that every piece of theory is anchored to a "Where this shows up in practice" note connecting it to a specific model family, training pathology, or debugging technique. Neither audience is well served by a book that only does one of these things, so this book tries to do both, in every chapter.

How to Read This Book

Three tracks through the material. The chapters are written to be read start to finish, but three shorter tracks work if your time or background is limited.

Theory-first track. If your goal is to be able to read and evaluate machine learning papers, focus on the numbered definitions, theorems, and proofs in each chapter, and treat the Computational Companion sections as optional confirmation rather than required reading. Part I, Part II, and Chapters 8–10 are the highest-yield chapters for this track.

Code-first track. If your goal is to build better intuition for systems you already operate day to day, read each chapter's opening paragraph and its "Where this shows up in practice" note, then go directly to the Computational Companion and run the code. Come back for the derivations when a specific implementation choice (why this loss function, why this initialization, why this learning-rate schedule) stops making sense.

Reference track. Each chapter is self-contained enough to be read out of order once you have Part I under your belt; the dependency map below shows which chapters lean on which. Appendix B (notation) and Appendix C (extended proofs) are designed for lookup rather than linear reading.

Chapter dependency map.

Part I (Chapters 1–4) is a prerequisite for everything that follows and should be read first, or skimmed chapter-by-chapter as a refresher if the material is already familiar.

Part II (Chapters 5–8) depends on Part I throughout, particularly Chapter 1's matrix calculus and Chapter 2's convexity discussion.

Part III (Chapters 9–11) depends on Chapter 3 (probability) and, for Chapter 10's variational inference section, on Chapter 5's optimization machinery.

Part IV (Chapters 12–16) depends on Chapter 1 (linear algebra), Chapter 2 (calculus, particularly the chain rule), and Chapter 6 (stochastic optimization). Chapter 12's automatic differentiation is itself a dependency for every later chapter that trains a model.

Part V (Chapters 17–23) is the most interconnected part of the book: Chapter 17 depends on Chapter 12; Chapter 18 depends on Chapter 10's ELBO derivation; Chapter 19 depends on Chapter 5's convexity and Chapter 6's stochastic optimization; Chapter 21 depends on Chapter 9 (information theory) and Chapter 17; and Chapter 22 depends directly on Chapter 19's policy gradient theorem. Reading Part V out of order without Part IV is not recommended.

Running the code. Every Computational Companion listing in this book is written to run end to end on a standard CPU in well under a minute, using only five open-source libraries throughout: NumPy, SciPy, Matplotlib, scikit-learn, and PyTorch (introduced in Chapter 6 for optimizer cross-checks against `torch.optim`, with full coverage of PyTorch's automatic differentiation and neural-network-specific cross-checks beginning in Chapter 12). No other dependency — not JAX, not a tokenizer library, not any downloaded dataset — is required anywhere in the book; even Chapter 21's byte-pair-encoding tokenizer is implemented entirely from scratch in the chapter itself, since watching the algorithm work is more instructive than importing it. Appendix A covers environment setup in full; the short version is `pip install numpy scipy matplotlib scikit-learn torch`. Every listing in this book has been executed and its output verified before publication.

Exercises. Each chapter ends with a mix of derivation exercises (marked **T** for theory) and implementation exercises (marked **C** for code). Exercises are not answered in the text; Appendix D gives hints, not solutions, on the view that working through a proof or a bug is where the understanding actually happens.

Notation and Conventions

Scalars are written in lowercase italic (x, α), vectors in lowercase bold ($\mathbf{x}$, $\mathbf{w}$), matrices in uppercase bold ($\mathbf{A}$, $\mathbf{X}$), and random variables in uppercase italic (X, Y) unless a field's own convention (e.g., θ for parameters, $\mathcal{L}$ for a loss) is so standard that breaking it would be more confusing than keeping it. Vectors are column vectors by default; $\mathbf{x}^\top$ denotes a row vector.

Symbol	Meaning
$\mathbb{R}^n$	n-dimensional real coordinate space
$\mathbf{x}, \mathbf{w}$	column vectors
$\mathbf{A}, \mathbf{X}$	matrices
$\mathbf{A}^\top$	matrix transpose
$\mathbf{A}^{-1}$	matrix inverse
$\lVert \mathbf{x} \rVert_p$	ℓ_p norm of $\mathbf{x}$
$\langle \mathbf{x}, \mathbf{y} \rangle$	inner product
$\nabla f(\mathbf{x})$	gradient of f at $\mathbf{x}$
$\nabla^2 f(\mathbf{x})$	Hessian of f at $\mathbf{x}$
$\mathbf{J}_f$	Jacobian of a vector-valued function f
$\mathbb{E}[X]$	expectation of random variable X
$\operatorname{Var}(X)$	variance of X
$p(x), p(x \mid y)$	probability density/mass, conditional density/mass
$D_{\mathrm{KL}}(p \parallel q)$	Kullback–Leibler divergence from q to p

Symbol	Meaning
$\mathcal{N}\left(\mu, \sigma^2\right)$	Gaussian distribution with mean μ, variance σ^2
θ	model parameters
$\mathcal{L}\left(\theta\right)$	loss / objective function
$\mathcal{D}$	a dataset
$\mathcal{O}\left(\cdot\right)$	asymptotic (Big-O) growth rate
$\mathbf{1}\left[\cdot\right]$	indicator function
$\odot$	elementwise (Hadamard) product

A full symbol glossary organized by chapter appears in Appendix B. Code listings use standard Python 3 conventions throughout: `snake_case` for variables and functions, and an unambiguous suffix (such as `_np`, `_torch`, `_ours`, or similar) on each array/tensor variable whenever a listing mixes NumPy arrays and PyTorch tensors and disambiguation is needed.

Chapter 1 — Linear Algebra for Learning Systems

> **Technical scope and assumptions**
>
> The results and examples in this chapter use the following qualifications.
>
> - Condition-number perturbation estimates in this chapter use the stated operator norm and describe small relative perturbations; bounds differ depending on whether the matrix, right-hand side, or both are perturbed.
> - Eckart–Young identifies an optimal approximation to a fixed matrix. Its connection to LoRA is motivation, not a proof that a learned low-rank update preserves task performance.
> - A rank-k truncated SVD is unique only when the singular-value cutoff is separated; ties at the boundary can produce multiple minimizers.

Every learning algorithm in this book eventually reduces to operations on vectors and matrices: a dataset is a matrix, a model's parameters are a vector, a prediction is a matrix-vector product, and training is a search over a vector space guided by a gradient. This chapter builds the linear algebra needed to reason about that search precisely — not as a review of an undergraduate course, but organized around the two questions that actually determine whether a learning algorithm works: *how is information distributed across the directions of a matrix* (rank, eigenstructure, the SVD), and *how sensitive is a computation to small changes in its input* (norms, condition numbers). The chapter closes with a matrix calculus primer that every later chapter relies on.

1.1 Vector Spaces, Norms, and Inner Products

Definition 1.1 (Vector space). A vector space V over $\mathbb{R}$ is a set closed under addition and scalar multiplication, satisfying the usual associativity, commutativity, distributivity, and identity axioms. Throughout this book $V = \mathbb{R}^n$: an n-dimensional feature vector, a weight vector, a gradient — all live in some $\mathbb{R}^n$.

A **norm** $\| \cdot \| : V \to \mathbb{R}_{\geq 0}$ measures the "size" of a vector and must satisfy: (i) $\| \mathbf{x} \| = 0 \Leftrightarrow \mathbf{x} = \mathbf{0}$; (ii) $\| \alpha \mathbf{x} \| = |\alpha| \, \| \mathbf{x} \|$; (iii) the triangle inequality $\| \mathbf{x} + \mathbf{y} \| \leq \| \mathbf{x} \| + \| \mathbf{y} \|$.

Definition 1.2 (ℓ_p norms). For $\mathbf{x} \in \mathbb{R}^n$ and $p \geq 1$,

$$\| \mathbf{x} \|_p = \left(\sum_{i=1}^{n} |x_i|^p \right)^{1/p}, \qquad \| \mathbf{x} \|_\infty = \max_i |x_i|.$$

Three special cases recur constantly in ML: $\| \mathbf{x} \|_1 = \sum_i |x_i|$ (drives sparsity when used as a regularizer — Chapter 8), $\| \mathbf{x} \|_2 = \sqrt{\sum_i x_i^2}$ (the Euclidean norm, the default unless stated otherwise), and $\| \mathbf{x} \|_\infty$ (worst-case coordinate error, used in robustness analysis).

Definition 1.3 (Inner product). $\langle \mathbf{x}, \mathbf{y} \rangle = \mathbf{x}^\top \mathbf{y} = \sum_i x_i y_i$ on $\mathbb{R}^n$. The Euclidean norm is the norm *induced* by this inner product: $\| \mathbf{x} \|_2 = \sqrt{\langle \mathbf{x}, \mathbf{x} \rangle}$. Not every norm is induced by an inner product — ℓ_1 and ℓ_∞ are not — and this distinction matters: inner-product spaces have a notion of *angle* and *orthogonality* that general normed spaces lack.

Theorem 1.1 (Cauchy–Schwarz inequality). For all $\mathbf{x}, \mathbf{y} \in \mathbb{R}^n$, $|\langle \mathbf{x}, \mathbf{y} \rangle| \leq \| \mathbf{x} \|_2 \, \| \mathbf{y} \|_2$, with equality iff $\mathbf{x}$ and $\mathbf{y}$ are linearly dependent.

Proof. If $\mathbf{y} = \mathbf{0}$ the claim is trivial. Otherwise, for any $t \in \mathbb{R}$, $0 \leq \| \mathbf{x} - t\mathbf{y} \|_2^2 = \| \mathbf{x} \|_2^2 - 2t\langle \mathbf{x}, \mathbf{y} \rangle + t^2 \| \mathbf{y} \|_2^2$. This is a quadratic in t that is nonnegative everywhere, so its discriminant is ≤ 0: $4\langle \mathbf{x}, \mathbf{y} \rangle^2 - 4 \| \mathbf{x} \|_2^2 \, \| \mathbf{y} \|_2^2 \leq 0$, which rearranges to the claim. Equality holds exactly when the discriminant is zero, i.e., when $\| \mathbf{x} - t\mathbf{y} \|_2 = 0$ for some t. ■

Cauchy–Schwarz is the reason cosine similarity is bounded in $[-1, 1]$, and it is the key lemma behind the convergence proofs in Chapter 6.

1.2 Matrices as Linear Maps: Rank and the Four Fundamental Subspaces

A matrix $\mathbf{A} \in \mathbb{R}^{m \times n}$ is simultaneously a table of numbers and a linear map $\mathbf{A}: \mathbb{R}^n \to \mathbb{R}^m$. Everything in this section is about understanding that map: what it can reach (its range) and what it destroys (its null space).

Definition 1.4. The **column space** $\mathrm{Col}(\mathbf{A}) = \{\mathbf{A}\mathbf{x}: \mathbf{x} \in \mathbb{R}^n\} \subseteq \mathbb{R}^m$ is the set of all outputs $\mathbf{A}$ can produce. The **null space** $\mathrm{Null}(\mathbf{A}) = \{\mathbf{x} \in \mathbb{R}^n: \mathbf{A}\mathbf{x} = \mathbf{0}\}$ is the set of inputs $\mathbf{A}$ collapses to zero. The **rank** of $\mathbf{A}$, $\mathrm{rank}(\mathbf{A})$, is $\dim(\mathrm{Col}(\mathbf{A}))$, equivalently the number of linearly independent columns (equivalently rows — a nontrivial but standard fact).

Theorem 1.2 (Rank–nullity theorem). For $\mathbf{A} \in \mathbb{R}^{m \times n}$, $\mathrm{rank}(\mathbf{A}) + \dim(\mathrm{Null}(\mathbf{A})) = n$.

Proof sketch. Let $\{\mathbf{v}_1, \ldots, \mathbf{v}_k\}$ be a basis for $\mathrm{Null}(\mathbf{A})$ and extend it to a basis $\{\mathbf{v}_1, \ldots, \mathbf{v}_k, \mathbf{v}_{k+1}, \ldots, \mathbf{v}_n\}$ of $\mathbb{R}^n$. One shows $\{\mathbf{A}\mathbf{v}_{k+1}, \ldots, \mathbf{A}\mathbf{v}_n\}$ is a basis for $\mathrm{Col}(\mathbf{A})$: it spans $\mathrm{Col}(\mathbf{A})$ because any $\mathbf{A}\mathbf{x}$ can be written using this basis after discarding the null-space components of $\mathbf{x}$, and it is linearly independent because a nontrivial dependence among the $\mathbf{A}\mathbf{v}_i$ ($i > k$) would produce a nonzero vector in the span of $\mathbf{v}_{k+1}, \ldots, \mathbf{v}_n$ that also lies in $\mathrm{Null}(\mathbf{A})$ — contradicting that this span intersects $\mathrm{Null}(\mathbf{A})$ only at $\mathbf{0}$. Hence $\mathrm{rank}(\mathbf{A}) = n - k$. ∎

This theorem is the reason an underdetermined least-squares problem (n parameters, fewer than n independent constraints) has infinitely many solutions: $\mathrm{Null}(\mathbf{A})$ is nontrivial, and any particular solution can be shifted by any null-space vector without changing $\mathbf{A}\mathbf{x}$. This is exactly the situation regularization (Chapter 8) is designed to resolve.

The four fundamental subspaces. $\mathrm{Col}(\mathbf{A})$, $\mathrm{Null}(\mathbf{A})$, $\mathrm{Col}\left(\mathbf{A}^\top\right)$ (the row space), and $\mathrm{Null}\left(\mathbf{A}^\top\right)$ together decompose the input and output spaces:

$\mathbb{R}^n = \mathrm{Col}\left(\mathbf{A}^\top\right) \oplus \mathrm{Null}\left(\mathbf{A}\right)$ and $\mathbb{R}^m = \mathrm{Col}\left(\mathbf{A}\right) \oplus \mathrm{Null}\left(\mathbf{A}^\top\right)$, with each pair orthogonal. This decomposition is what makes the SVD (Section 1.6) so informative: its singular vectors are orthonormal bases for exactly these four subspaces.

1.3 Orthogonality and Projections

Definition 1.5. Vectors $\mathbf{x}, \mathbf{y}$ are **orthogonal** if $\langle \mathbf{x}, \mathbf{y} \rangle = 0$. A set of vectors is **orthonormal** if pairwise orthogonal and each has unit norm.

Theorem 1.3 (Orthogonal projection). Given a subspace $S \subseteq \mathbb{R}^n$ with orthonormal basis $\{\mathbf{u}_1, \ldots, \mathbf{u}_k\}$, the vector in S closest to a given $\mathbf{x} \in \mathbb{R}^n$ (in Euclidean distance) is

$$\mathrm{proj}_S\left(\mathbf{x}\right) = \sum_{i=1}^{k} \langle \mathbf{x}, \mathbf{u}_i \rangle \mathbf{u}_i,$$

and the residual $\mathbf{x} - \mathrm{proj}_S\left(\mathbf{x}\right)$ is orthogonal to every vector in S.

Proof. Write any $\mathbf{s} \in S$ as $\mathbf{s} = \sum_i c_i \mathbf{u}_i$. Then $\| \mathbf{x} - \mathbf{s} \|_2^2 = \| \mathbf{x} \|_2^2 - 2\sum_i c_i \langle \mathbf{x}, \mathbf{u}_i \rangle + \sum_i c_i^2$ (using orthonormality to kill cross terms), which is a separable sum of quadratics in each c_i, minimized at $c_i = \langle \mathbf{x}, \mathbf{u}_i \rangle$. Orthogonality of the residual follows by direct computation: $\langle \mathbf{x} - \mathrm{proj}_S\left(\mathbf{x}\right), \mathbf{u}_j \rangle = \langle \mathbf{x}, \mathbf{u}_j \rangle - \langle \mathbf{x}, \mathbf{u}_j \rangle = 0$. ∎

This is precisely the geometry of ordinary least squares: fitting a linear model projects the target vector $\mathbf{y}$ onto the column space of the design matrix, and the residual is orthogonal to every predictor — the algebraic reason residual plots that show a correlation with a feature indicate a misspecified model.

Gram–Schmidt. Given linearly independent $\mathbf{v}_1, \ldots, \mathbf{v}_k$, an orthonormal basis for their span is built iteratively: $\mathbf{u}_1 = \mathbf{v}_1 / \| \mathbf{v}_1 \|_2$, and for $i > 1$, subtract off the projection onto the span of the previous $\mathbf{u}_j$'s before normalizing:

$$\tilde{\mathbf{u}}_i = \mathbf{v}_i - \sum_{j<i} \langle \mathbf{v}_i, \mathbf{u}_j \rangle \mathbf{u}_j, \qquad \mathbf{u}_i = \tilde{\mathbf{u}}_i / \| \tilde{\mathbf{u}}_i \|_2 .$$

This is exactly the QR decomposition of Chapter 7, and it is numerically fragile in its naive form — a point we return to there.

1.4 Eigenvalues, Eigenvectors, and the Spectral Theorem

Definition 1.6. $\lambda \in \mathbb{C}$ is an **eigenvalue** of $\mathbf{A} \in \mathbb{R}^{n \times n}$ with **eigenvector** $\mathbf{v} \neq \mathbf{0}$ if $\mathbf{A}\mathbf{v} = \lambda \mathbf{v}$: the map $\mathbf{A}$ acts on $\mathbf{v}$ by pure scaling. The eigenvalues are the roots of the characteristic polynomial $\det\left(\mathbf{A} - \lambda \mathbf{I}\right) = 0$.

Theorem 1.4 (Spectral theorem for symmetric matrices). If $\mathbf{A} \in \mathbb{R}^{n \times n}$ is symmetric ($\mathbf{A} = \mathbf{A}^{\top}$), then $\mathbf{A}$ has n real eigenvalues (counted with multiplicity) and an orthonormal basis of eigenvectors: $\mathbf{A} = \mathbf{Q}\mathbf{\Lambda}\mathbf{Q}^{\top}$ where $\mathbf{Q}$ is orthogonal ($\mathbf{Q}^{\top}\mathbf{Q} = \mathbf{I}$) and $\mathbf{\Lambda} = \operatorname{diag}\left(\lambda_1, ..., \lambda_n\right)$.

Proof sketch. Realness of the eigenvalues: if $\mathbf{A}\mathbf{v} = \lambda \mathbf{v}$ with $\mathbf{v} \neq \mathbf{0}$ possibly complex, then $\bar{\mathbf{v}}^{\top}\mathbf{A}\mathbf{v} = \lambda \bar{\mathbf{v}}^{\top}\mathbf{v}$; taking the conjugate transpose of the whole equation and using $\mathbf{A} = \mathbf{A}^{\top}$ real shows $\bar{\lambda}\bar{\mathbf{v}}^{\top}\mathbf{v} = \bar{\mathbf{v}}^{\top}\mathbf{A}\mathbf{v}$ as well, so $\lambda = \bar{\lambda}$. Existence of the orthonormal eigenbasis follows by induction on n: take a (real, unit-norm) eigenvector $\mathbf{v}_1$ for λ_1, extend it to an orthonormal basis of $\mathbb{R}^n$, note that $\mathbf{A}$ restricted to $\mathbf{v}_1^{\perp}$ is again symmetric (because $\mathbf{A}$ is symmetric and $\mathbf{v}_1^{\perp}$ is $\mathbf{A}$-invariant when $\mathbf{v}_1$ is an eigenvector), and apply the inductive hypothesis to that $(n-1)$-dimensional restriction. ■

Definition 1.7 (Positive (semi-)definiteness). A symmetric $\mathbf{A}$ is **positive semidefinite** (PSD) if $\mathbf{x}^{\top}\mathbf{A}\mathbf{x} \geq 0$ for all $\mathbf{x}$, and **positive definite** (PD) if this is strict for all $\mathbf{x} \neq \mathbf{0}$. By the spectral theorem, $\mathbf{A}$ is PSD iff all its eigenvalues are ≥ 0, and PD iff all are > 0 — a fact used constantly to check whether a critical point of a loss function is a minimum (Chapter 2) or whether a covariance matrix is valid.

1.5 Condition Numbers and Numerical Sensitivity

Definition 1.8. For an invertible $\mathbf{A}$, the **condition number** (in the 2-norm) is

$$\kappa\left(\mathbf{A}\right) = \left\|\mathbf{A}\right\|_2 \left\|\mathbf{A}^{-1}\right\|_2 = \frac{\sigma_{\max}\left(\mathbf{A}\right)}{\sigma_{\min}\left(\mathbf{A}\right)},$$

the ratio of the largest to smallest singular value (Section 1.6). $\kappa\left(\mathbf{A}\right) \geq 1$ always, with $\kappa\left(\mathbf{A}\right) = 1$ only for scaled orthogonal matrices.

The condition number answers a precise question: if $\mathbf{b}$ in $\mathbf{A}\mathbf{x} = \mathbf{b}$ is perturbed by a small relative amount δ, how much can the relative error in the solution $\mathbf{x}$ grow? The answer is bounded by $\kappa(\mathbf{A}) \cdot \delta$. In ML, $\mathbf{A}$ is often (an approximation to) the Hessian of a loss function, and $\kappa(\mathbf{A})$ governs how many gradient descent steps are needed: for a quadratic $f(\mathbf{x}) = \frac{1}{2}\mathbf{x}^\top \mathbf{A}\mathbf{x}$ with $\mathbf{A} \succ 0$, gradient descent's convergence rate is governed by $\left(\frac{\kappa(\mathbf{A})-1}{\kappa(\mathbf{A})+1}\right)^2$ per step (derived in Chapter 6) — a large condition number means slow, zig-zagging convergence, which is exactly why feature standardization (Chapter 6) and second-order methods (Chapter 14) both exist.

1.6 The Singular Value Decomposition

Theorem 1.5 (Existence of the SVD). Every $\mathbf{A} \in \mathbb{R}^{m \times n}$ can be written as $\mathbf{A} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^\top$, where $\mathbf{U} \in \mathbb{R}^{m \times m}$ and $\mathbf{V} \in \mathbb{R}^{n \times n}$ are orthogonal and $\mathbf{\Sigma} \in \mathbb{R}^{m \times n}$ is diagonal with nonnegative entries $\sigma_1 \geq \sigma_2 \geq \cdots \geq 0$ (the **singular values**).

Proof sketch. $\mathbf{A}^\top \mathbf{A}$ is symmetric PSD, so by Theorem 1.4 it has an orthonormal eigenbasis $\mathbf{v}_1, \ldots, \mathbf{v}_n$ with eigenvalues $\lambda_1 \geq \cdots \geq \lambda_n \geq 0$. Define $\sigma_i = \sqrt{\lambda_i}$ and, for $\sigma_i > 0$, $\mathbf{u}_i = \mathbf{A}\mathbf{v}_i / \sigma_i$. One checks directly that the $\mathbf{u}_i$ are orthonormal: $\langle \mathbf{u}_i, \mathbf{u}_j \rangle = \frac{\mathbf{v}_i^\top \mathbf{A}^\top \mathbf{A} \mathbf{v}_j}{\sigma_i \sigma_j} = \frac{\lambda_j \langle \mathbf{v}_i, \mathbf{v}_j \rangle}{\sigma_i \sigma_j} = \delta_{ij}$. Extend $\{\mathbf{u}_i\}$ to a full orthonormal basis of $\mathbb{R}^m$ if $m > \operatorname{rank}(\mathbf{A})$. By construction $\mathbf{A}\mathbf{v}_i = \sigma_i \mathbf{u}_i$ for every i, which is exactly $\mathbf{A}\mathbf{V} = \mathbf{U}\mathbf{\Sigma}$, i.e., $\mathbf{A} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^\top$. $\blacksquare$

Geometrically, every linear map is a rotation ($\mathbf{V}^\top$), followed by axis-aligned scaling ($\mathbf{\Sigma}$), followed by another rotation ($\mathbf{U}$). The columns of $\mathbf{V}$ corresponding to nonzero σ_i span $\operatorname{Col}\left(\mathbf{A}^\top\right)$; the remaining columns of $\mathbf{V}$ span $\operatorname{Null}(\mathbf{A})$ — the SVD makes Section 1.2's four fundamental subspaces explicit and orthonormal.

Theorem 1.6 (Eckart–Young). Among all rank-k matrices, the one closest to $\mathbf{A}$ in Frobenius norm is $\mathbf{A}_k = \sum_{i=1}^{k} \sigma_i \mathbf{u}_i \mathbf{v}_i^\top$ (truncating the SVD to its top k singular values), and $\|\mathbf{A} - \mathbf{A}_k\|_F = \sqrt{\sum_{i>k} \sigma_i^2}$.

This theorem is the mathematical justification for using PCA and low-rank approximation for compression and denoising: the SVD is not merely *a* good rank-k

approximation, it is provably the *best* one in Frobenius norm.

1.7 Matrix Calculus Primer

Training a model requires differentiating scalar loss functions with respect to vector- or matrix-valued parameters. Two identities, both derivable directly from the definition of the gradient (Chapter 2), account for the overwhelming majority of derivatives you will need in this book.

Gradient of a linear form. For $f(\mathbf{x}) = \mathbf{a}^\top \mathbf{x}$, $\nabla_{\mathbf{x}} f(\mathbf{x}) = \mathbf{a}$, since f is already linear in each x_i with coefficient a_i.

Gradient of a quadratic form. For $f(\mathbf{x}) = \mathbf{x}^\top \mathbf{A} \mathbf{x}$,

$$\nabla_{\mathbf{x}} f(\mathbf{x}) = \left(\mathbf{A} + \mathbf{A}^\top\right)\mathbf{x},$$

which simplifies to $2\mathbf{A}\mathbf{x}$ when $\mathbf{A}$ is symmetric.

Derivation. Write $f(\mathbf{x}) = \sum_{i,j} A_{ij} x_i x_j$. Then $\frac{\partial f}{\partial x_k} = \sum_j A_{kj} x_j + \sum_i A_{ik} x_i$ (product rule, differentiating the two occurrences of x_k separately), which is exactly the k-th entry of $\mathbf{A}\mathbf{x} + \mathbf{A}^\top \mathbf{x}$. $\blacksquare$

This single identity gives the normal equations for least squares in one line: minimizing $\|\mathbf{X}\mathbf{w} - \mathbf{y}\|_2^2 = \mathbf{w}^\top\left(\mathbf{X}^\top\mathbf{X}\right)\mathbf{w} - 2\mathbf{y}^\top\mathbf{X}\mathbf{w} + \mathbf{y}^\top\mathbf{y}$ over $\mathbf{w}$, setting the gradient $2\mathbf{X}^\top\mathbf{X}\mathbf{w} - 2\mathbf{X}^\top\mathbf{y}$ to zero gives $\mathbf{X}^\top\mathbf{X}\mathbf{w} = \mathbf{X}^\top\mathbf{y}$.

1.8 Computational Companion

The listing below implements PCA two ways — via eigendecomposition of the covariance matrix, and via the SVD of the centered data matrix directly — and confirms they agree, per Theorem 1.6. It then builds a small visualizer showing how the condition number of a quadratic's Hessian predicts gradient descent's convergence speed, anticipating Chapter 6.

```
"""
Chapter 1 Computational Companion
Requires: numpy, scipy, matplotlib, scikit-learn
"""
```

```python
import numpy as np
from numpy.linalg import eigh, svd, cond
from sklearn.decomposition import PCA

rng = np.random.default_rng(seed=1)


def pca_via_eigendecomposition(X: np.ndarray, k: int):
    """PCA by eigendecomposing the covariance matrix (Theorem 1.4)."""
    X_centered = X - X.mean(axis=0, keepdims=True)
    cov = (X_centered.T @ X_centered) / (X.shape[0] - 1)
    eigvals, eigvecs = eigh(cov)          # ascending order
    order = np.argsort(eigvals)[::-1]     # descending
    top_k_vecs = eigvecs[:, order[:k]]
    scores = X_centered @ top_k_vecs
    return scores, top_k_vecs, eigvals[order[:k]]


def pca_via_svd(X: np.ndarray, k: int):
    """PCA by SVD of the centered data matrix directly (Theorem 1.5/1.6)."""
    X_centered = X - X.mean(axis=0, keepdims=True)
    U, S, Vt = svd(X_centered, full_matrices=False)
    top_k_vecs = Vt[:k].T
    scores = X_centered @ top_k_vecs
    explained_var = (S[:k] ** 2) / (X.shape[0] - 1)
    return scores, top_k_vecs, explained_var


def test_pca_equivalence():
    X = rng.normal(size=(200, 5)) @ rng.normal(size=(5, 5))  # correlated
features
    k = 2
    scores_eig, vecs_eig, var_eig = pca_via_eigendecomposition(X, k)
    scores_svd, vecs_svd, var_svd = pca_via_svd(X, k)

    # Eigenvectors/scores can differ by a sign flip; compare up to sign.
    for j in range(k):
        sign = np.sign(np.dot(vecs_eig[:, j], vecs_svd[:, j])) or 1.0
        assert np.allclose(vecs_eig[:, j], sign * vecs_svd[:, j], atol=1e-8)
        assert np.allclose(scores_eig[:, j], sign * scores_svd[:, j], atol=1e-6)
    assert np.allclose(var_eig, var_svd, atol=1e-6)

    # Cross-check against scikit-learn's production implementation.
    skl = PCA(n_components=k).fit(X)
    for j in range(k):
        sign = np.sign(np.dot(vecs_eig[:, j], skl.components_[j])) or 1.0
        assert np.allclose(vecs_eig[:, j], sign * skl.components_[j], atol=1e-6)
    print("PCA via eigendecomposition, SVD, and scikit-learn all agree.")


def condition_number_vs_gradient_descent():
    """
    Show empirically that a quadratic's convergence rate under gradient
    descent tracks its Hessian condition number kappa, as Section 1.5 predicts.

    To make runs comparable across different kappa, we fix the rotation Q
    once and start every run from a point with *equal weight along every
```

```
            eigenvector* (x0 = Q @ (1,...,1)/sqrt(n)). This isolates the effect of
            kappa itself: a random initial point can, by chance, put little weight
            on the slow-converging (smallest-eigenvalue) direction, which makes
            finite-horizon comparisons across kappa noisy and occasionally
            non-monotonic even though the asymptotic rate result still holds. We
            also compare *relative* loss (loss / initial loss) rather than raw
            loss, since raw loss magnitude depends on the arbitrary scale of x0.
            """
            n = 20
            Q, _ = np.linalg.qr(rng.normal(size=(n, n)))          # shared rotation
            x0_shared = Q @ (np.ones(n) / np.sqrt(n))              # equal-weight start

            def make_quadratic(kappa: float):
                eigvals = np.linspace(1.0, kappa, n)
                return Q @ np.diag(eigvals) @ Q.T

            def run_gd(A: np.ndarray, n_steps: int = 200):
                x = x0_shared.copy()
                step = 1.0 / np.max(np.linalg.eigvalsh(A))  # stable step size
                losses = []
                for _ in range(n_steps):
                    losses.append(0.5 * x @ A @ x)
                    grad = A @ x
                    x = x - step * grad
                return np.array(losses)

            results = {}
            for kappa in (1.0, 10.0, 100.0, 1000.0):
                A = make_quadratic(kappa)
                assert np.isclose(cond(A), kappa, rtol=1e-3)
                losses = run_gd(A)
                relative_final = losses[-1] / losses[0]
                results[kappa] = relative_final
                print(f"kappa={kappa:>7.1f}  relative loss after 200 steps:
{relative_final:.3e}")

            # Sanity check: higher condition number should mean slower relative
            # convergence, i.e. a larger *fraction* of the initial loss remaining
            # after a fixed number of steps.
            kappas_sorted = sorted(results)
            relative_losses = [results[k] for k in kappas_sorted]
            assert all(x <= y + 1e-9 for x, y in zip(relative_losses,
relative_losses[1:])), \
                "expected monotonically worse relative convergence as kappa grows"
            print("Confirmed: relative convergence speed degrades monotonically with
kappa.")


        if __name__ == "__main__":
            test_pca_equivalence()
            condition_number_vs_gradient_descent()
```

Running this module prints confirmation that the two from-scratch PCA implementations agree with each other and with scikit-learn to numerical precision,

and that gradient descent's convergence measurably slows as the condition number of the problem grows from 1 to 1000 — the empirical face of Section 1.5's theory.

1.9 Exercises

T1.1. Prove that $\| \mathbf{x} \|_\infty \leq \| \mathbf{x} \|_2 \leq \| \mathbf{x} \|_1$ for any $\mathbf{x} \in \mathbb{R}^n$, and identify the vectors for which each inequality is tight.

T1.2. Prove that for any $\mathbf{A} \in \mathbb{R}^{m \times n}$, $\text{Null}\left(\mathbf{A}^\top \mathbf{A} \right) = \text{Null}\left(\mathbf{A} \right)$. (This fact is why $\mathbf{A}^\top \mathbf{A}$ is invertible exactly when $\mathbf{A}$'s columns are linearly independent — the condition for the normal equations to have a unique solution.)

T1.3. Show that $\kappa\left(\mathbf{A}^\top \mathbf{A} \right) = \kappa\left(\mathbf{A} \right)^2$. What does this imply about forming the normal equations $\mathbf{X}^\top \mathbf{X} \mathbf{w} = \mathbf{X}^\top \mathbf{y}$ explicitly versus solving the least-squares problem via a QR- or SVD-based solver (Chapter 7)?

C1.1. Extend `condition_number_vs_gradient_descent` to also run momentum-accelerated gradient descent (you may look ahead to Chapter 6 for the update rule) and confirm empirically that momentum narrows, but does not eliminate, the gap between $\kappa = 10$ and $\kappa = 1000$.

C1.2. Using `pca_via_svd`, implement image compression: load a small grayscale image as a matrix, compute its rank-k SVD approximation for several values of k, and plot reconstruction error against k. Confirm your empirical error curve matches the $\sqrt{\sum_{i>k} \sigma_i^2}$ prediction of Theorem 1.6.

1.10 Where This Shows Up in Practice

Feature standardization before training a linear model or a neural network is a direct application of Section 1.5: unstandardized features with wildly different scales produce a loss Hessian with a large condition number, and gradient-based optimizers slow to a crawl. Low-rank adaptation methods for fine-tuning large models (an idea you will see formalized once Chapter 21 introduces LLM pretraining) rest on Theorem 1.6's guarantee that a low-rank update is often nearly

as good as a full-rank one whenever a matrix's singular values decay quickly. And whenever a numerical linear algebra routine — Cholesky, matrix inversion, eigendecomposition — silently produces `nan` or wildly inaccurate output on real data, the first thing worth checking is the condition number of the matrix you handed it.