CALCULUS

FROM BEGINNER TO EXPERT

A complete, visual, application-driven course — limits and derivatives through multivariable optimization and series, with a capstone on how calculus powers Machine Learning, Deep Learning, and AI.

What you will be able to do

Differentiate and integrate fluently *and* reason about why it works; extend calculus to many variables; derive gradient descent, backpropagation, and the gradients of standard loss functions; and explain precisely why calculus is the engine that lets AI learn.

Format: self-paced reference & course | **Prerequisites:** high-school algebra; a little trigonometry

Part of the Comprehensive Machine Learning & Deep Learning Curriculum.

Preface: how to use this book

If linear algebra is the language of machine learning, calculus is its *engine of change*. Calculus is the mathematics of things that vary — how a quantity responds to a small nudge in another (the derivative), and how countless tiny pieces accumulate into a whole (the integral). Every time a model "learns," it is computing derivatives and stepping downhill; every time it reasons about probability, it is computing integrals. This book builds that machinery from scratch and carries it all the way to the gradients behind modern AI.

Who this is for

This book starts from the very beginning — what a function is and what a limit means — and climbs to multivariable optimization, infinite series, and the matrix calculus of backpropagation. It is written for the self-learner who wants both **computational fluency** ("I can compute the derivative and the integral") and **conceptual understanding** ("I know what they mean and why they matter"). Every important result is motivated with a picture, a worked example, and — wherever possible — a forward link to how it is used in machine learning.

The two big ideas

Almost all of calculus orbits two problems and the stunning theorem that links them:

- **The tangent problem** (differential calculus): given a curve, find its instantaneous slope — the rate of change.
- **The area problem** (integral calculus): given a curve, find the area beneath it — the total accumulation.
- **The Fundamental Theorem of Calculus**: these two problems are inverses of each other. Differentiation and integration undo one another.

The structure

- **Part I — Foundations & Limits** (beginner): functions and the idea of calculus; limits and continuity.
- **Part II — Differential Calculus** (core): the derivative, the rules, and their applications.
- **Part III — Integral Calculus** (core): the integral and the Fundamental Theorem, techniques, applications.
- **Part IV — Series & Multivariable** (advanced): sequences and Taylor series; partial derivatives and the gradient; multiple integrals; optimization and differential equations.
- **Part V — Applications**: a capstone chapter on how calculus powers machine learning, deep learning, and AI.

How to read it

Read with a pen. When you meet a *Definition* box, restate it in your own words; when you meet an *Example*, cover the solution and try it first; when you meet a *Try it in code* box, actually run it. The colored boxes recur throughout:

- **Key idea** — the one sentence to remember.
- **Intuition** — the geometric or physical picture.

- **Common pitfall** — mistakes to avoid.
- **How this powers ML / AI** — the forward link to applications.
- **Try it in code** — a hands-on check in Python/NumPy.

Key idea

Two mental images carry the whole subject. The **derivative** is the slope of the tangent line — the best local *linear* approximation to a function, the answer to "if I nudge the input, how fast does the output move?" The **integral** is the area under the curve — the limit of adding up many thin slices. Keep both pictures in view and calculus stops being a bag of rules and becomes a single, coherent idea.

Contents

PART I

Foundations and Limits

1

Functions and the Idea of Calculus

Level: Beginner

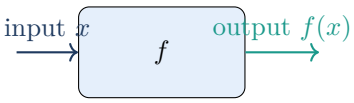
Calculus studies how things change and how things accumulate. Before we can speak of change, we need the object that changes: the **function**. This chapter sets up functions and the handful of "essential" ones that appear everywhere, then introduces the two questions — the slope of a curve and the area under it — that the entire subject is built to answer.

1.1 What is a function?

Definition 1.1. A **function** f is a rule that assigns to each input x in a set called the **domain** exactly one output $f(x)$. The set of outputs is the **range**. We write $f : \mathbb{R} \to \mathbb{R}$ for a function taking a real number to a real number.

Intuition

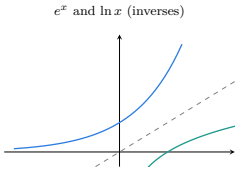
Think of a function as a machine: feed in x, get out $f(x)$ — and crucially, the same input always yields the same output (that is the "exactly one" clause). A function can be described four ways: by a *formula* ($f(x) = x^2$), a *graph*, a *table* of values, or a *verbal rule*. Fluency means moving between these views; calculus mostly reasons with the graph and the formula.

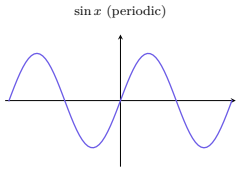


1.2 A catalog of essential functions

A small library of functions covers almost everything in machine learning:

- **Linear** $f(x) = mx + b$: constant slope m; the simplest model and the local picture of *every* differentiable function.
- **Power** $f(x) = x^n$ and **polynomials**: sums of powers.
- **Exponential** $f(x) = e^x$: the function equal to its own derivative; models growth, decay, and (through e^{-x}) the tails of probability distributions.
- **Logarithm** $f(x) = \ln x$: the inverse of e^x; turns products into sums and powers into products, which is why log-likelihoods are everywhere in ML.
- **Trigonometric** $\sin x, \cos x$: periodic; the basis of Fourier analysis and positional encodings.





1.2.1 Building new functions from old

Functions combine by addition, multiplication, and especially **composition**: $(f \circ g)(x) = f(g(x))$, "do g, then f." Composition is the structural backbone of deep learning — a neural network is a deep composition of simple functions — and the *chain rule* (Chapter 4) exists precisely to differentiate compositions.

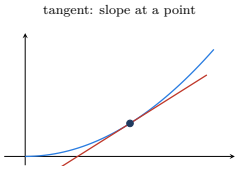
Definition 1.2. A function is **one-to-one** if different inputs give different outputs; only then does it have an **inverse** f^{-1} that undoes it: $f^{-1}(f(x)) = x$. Graphically, f^{-1} is the reflection of f across the line $y = x$ (as with e^x and $\ln x$ above).

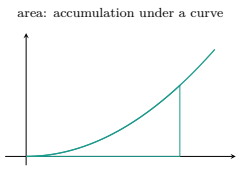
1.3 The two big problems

Calculus was invented to solve two geometric problems that turned out to be inverse to each other.

Key idea

The tangent problem. Given the graph of f, what is its *slope* at a point — the instantaneous rate of change? A straight line has an obvious slope; a curve does not, because the slope changes everywhere. Resolving this gives the **derivative** (differential calculus).
The area problem. Given the graph of f, what is the *area* between it and the x-axis over an interval — the total accumulation? Rectangles approximate it; making them infinitely thin gives the exact answer. Resolving this gives the **integral** (integral calculus).





Intuition

The two problems are secretly the same problem viewed from opposite ends. If $f(t)$ is your position at time t, the *slope* of the position graph is your velocity (a derivative), and the *area* under the velocity graph is the distance you travelled (an integral). Differentiation and integration undo each other — a fact so central it is called the **Fundamental Theorem of Calculus** (Chapter 6).

1.4 Rate of change: the average that becomes instantaneous

The slope of f between two points is the **average rate of change**:

$$\frac{\Delta y}{\Delta x} = \frac{f(x+h) - f(x)}{h}.$$

This is the slope of the *secant* line through the two points. The whole trick of differential calculus is to let $h \to 0$: the secant lines approach the *tangent* line, and the average rate becomes the **instantaneous** rate. Making that limit precise is the job of Chapter 2.

How this powers ML / AI

Functions are the models of machine learning, and composition is their architecture. A neural network is a function $f_{\boldsymbol{\theta}}$ with millions of parameters $\boldsymbol{\theta}$, built by composing linear maps with simple nonlinearities (**activation functions**) such as the sigmoid $\sigma(x) = 1/(1+e^{-x})$, tanh, and ReLU $\max(0, x)$ — all from the catalog above. Training adjusts $\boldsymbol{\theta}$ so the function fits data, and that adjustment is driven by *rates of change* (derivatives) of a loss with respect to each parameter. The exponential and logarithm in particular run the softmax and cross-entropy that almost every classifier ends with.

Try it in NumPy

```
import numpy as np
f = lambda x: 0.5*x**2
# average rate of change over [1.5, 1.5+h] approaches the slope 1.5 as h->0
for h in [1, 0.1, 0.01, 1e-4]:
    print(h, (f(1.5+h) - f(1.5)) / h)     # -> 1.5
sigmoid = lambda x: 1/(1+np.exp(-x))       # a key ML activation function
print(sigmoid(np.array([-2.0, 0.0, 2.0])))
```

Chapter summary

- A **function** assigns one output to each input; describe it by formula, graph, table, or words.
- Know the essential functions: linear, power/polynomial, e^x, $\ln x$, $\sin$ / $\cos$ — and **composition** $f(g(x))$.
- Calculus answers two inverse questions: the **tangent** (slope/derivative) and the **area** (integral).
- The derivative is the limit of average rates of change $\frac{f(x+h)-f(x)}{h}$ as $h \to 0$.
- ML models are functions built by composition; activations come straight from this catalog.

2

Limits and Continuity

Level: Beginner

The limit is the idea that makes calculus rigorous. "Instantaneous" rate of change and "infinitely thin" rectangles both hide a limit: a quantity we approach but may never reach. This chapter builds limits intuitively and precisely, the laws for computing them, and the notion of **continuity** — an "unbroken" function — which guarantees the well-behaved landscapes that optimization relies on.

2.1 The limit, intuitively

Definition 2.1. We write $\lim_{x \to a} f(x) = L$ to mean: as x gets arbitrarily close to a (from either side, but not equal to a), the value $f(x)$ gets arbitrarily close to L. The limit describes where the function is *heading*, regardless of what — if anything — happens exactly at a.

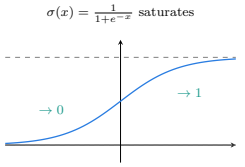
Intuition

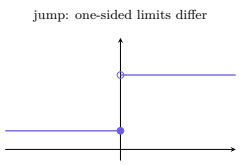
The phrase "but not equal to a" is the whole point. A limit asks about the *neighborhood* of a point, not the point itself. This is exactly what lets us make sense of $\frac{f(x+h)-f(x)}{h}$ as $h \to 0$: we never divide by zero; we watch where the ratio heads as h shrinks. The function may even be undefined at a and still have a perfectly good limit there.

Example 2.2. Consider $f(x) = \dfrac{x^2 - 1}{x - 1}$. At $x = 1$ this is $\frac{0}{0}$, undefined. But for $x \neq 1$, $f(x) = \dfrac{(x-1)(x+1)}{x-1} = x + 1$, so $\lim_{x \to 1} f(x) = 2$. The graph is the line $y = x + 1$ with a single hole at $(1, 2)$ — the limit fills in the hole.

2.2 One-sided and infinite limits

Sometimes the approach direction matters. The **left limit** $\lim_{x \to a^-} f(x)$ and **right limit** $\lim_{x \to a^+} f(x)$ consider $x < a$ and $x > a$ separately; the two-sided limit exists *only when both agree*. Limits can also be infinite (a vertical asymptote, $f \to \pm\infty$) or taken *at* infinity ($x \to \pm\infty$, describing end behavior). Limits at infinity are how we describe the tails of distributions and the saturation of activation functions.





2.3 Computing limits: the limit laws

For most functions, limits are computed by the obvious algebra, provided no division by zero occurs.

Proposition 2.3 (Limit laws)**.** If $\lim_{x\to a} f$ and $\lim_{x\to a} g$ exist, then limits distribute over sums, products, and quotients (denominator limit nonzero):

$$\lim(f \pm g) = \lim f \pm \lim g, \quad \lim(fg) = \lim f \cdot \lim g, \quad \lim \frac{f}{g} = \frac{\lim f}{\lim g}.$$

For a continuous function, $\lim_{x\to a} f(x) = f(a)$ — you may simply "plug in."

When plugging in gives an **indeterminate form** like $\frac{0}{0}$ or $\frac{\infty}{\infty}$, the limit is not automatic: it must be resolved by algebra (as above), by the squeeze theorem, or later by L'Hôpital's rule (Chapter 5).

Proposition 2.4 (Squeeze theorem)**.** If $g(x) \le f(x) \le h(x)$ near a and $\lim_{x\to a} g = \lim_{x\to a} h = L$, then $\lim_{x\to a} f = L$. Trapping a function between two others that meet forces it to the same limit; this is how one proves the foundational $\lim_{x\to 0} \dfrac{\sin x}{x} = 1$.

2.4 The precise definition (epsilon–delta)

The intuitive "arbitrarily close" is made exact by the ε–δ definition — the logical bedrock of analysis.

Definition 2.5 (ε–δ)**.** $\lim\limits_{x\to a} f(x) = L$ means: for every tolerance $\varepsilon > 0$ there is a $\delta > 0$ such that whenever $0 < |x - a| < \delta$, we have $|f(x) - L| < \varepsilon$.

Intuition

Read it as a challenge–response game. A skeptic demands that $f(x)$ land within ε of L; you must produce a radius δ around a guaranteeing it. If you can answer *every* ε, the limit truly is L. You will rarely compute with ε–δ directly, but it is what makes every later theorem trustworthy.

2.5 Continuity

Definition 2.6. A function f is **continuous at** a if $\lim\limits_{x\to a} f(x) = f(a)$ — the limit exists, the function is defined there, and the two agree. A function continuous at every point of an interval is **continuous** on it: its graph has no holes, jumps, or breaks — you can draw it without lifting your pen.

Sums, products, quotients (nonzero denominator), and compositions of continuous functions are continuous, so polynomials, exponentials, logs (on their domain), and sines/cosines are all continuous — which is why "plug in" usually works.

Theorem 2.7 (Intermediate Value Theorem)**.** If f is continuous on $[a, b]$ and N lies between $f(a)$ and $f(b)$, then $f(c) = N$ for some $c \in [a, b]$. A continuous function cannot skip values: to get from one height to another it must pass through every height in between.

The IVT guarantees that root-finding methods like bisection work, and the related **Extreme Value Theorem** (a continuous function on a closed interval attains a maximum and minimum) is what makes optimization well-posed.

Common pitfall

A limit existing does not require the function to be *defined* at the point, and a function being defined does not guarantee continuity. Watch for removable holes ($\frac{0}{0}$ that cancels), jumps (one-sided limits disagree), and blow-ups (vertical asymptotes). In ML, a subtler issue is functions that are continuous but *not differentiable* at a point — ReLU $\max(0, x)$ has a corner at 0 — which is fine for forward passes but needs care (subgradients) when differentiating.

How this powers ML / AI

Continuity is the unspoken assumption that makes learning possible. Gradient-based training implicitly assumes the loss surface is continuous (and almost-everywhere differentiable) so that a small parameter change produces a small, predictable change in loss — letting us follow slopes downhill. Limits at infinity describe **saturation**: the sigmoid flattens to 0 and 1, which both enables probabilistic outputs and causes *vanishing gradients* when units saturate. The limit defining the derivative, next chapter, is the literal foundation of backpropagation.

Try it in NumPy

```python
import numpy as np
# sin(x)/x -> 1 as x -> 0 (a squeeze-theorem classic)
for x in [1, 0.1, 0.01, 1e-6]:
    print(x, np.sin(x)/x)
# bisection relies on the Intermediate Value Theorem
f = lambda x: x**2 - 2          # root at sqrt(2)
lo, hi = 1.0, 2.0
for _ in range(40):
    mid = (lo+hi)/2
    lo, hi = (mid, hi) if f(mid) < 0 else (lo, mid)
print((lo+hi)/2)                # ~ 1.41421356
```

Chapter summary

- A **limit** $\lim_{x \to a} f(x) = L$ describes where f heads near a, ignoring the value at a itself.
- Two-sided limits need agreeing one-sided limits; limits can be infinite or taken at infinity (saturation, tails).
- Use the **limit laws**; resolve indeterminate forms by algebra, the **squeeze theorem**, or L'Hôpital later.
- The ε–δ definition makes "arbitrarily close" precise and underpins every theorem.
- **Continuity** ($\lim_{x \to a} f = f(a)$) gives unbroken graphs; the IVT and EVT make root-finding and optimization well-posed.

PART II

Differential Calculus

3

The Derivative

Level: Core

The derivative is the central object of differential calculus and the single most important concept for machine learning. It answers the tangent problem: the instantaneous rate at which a function changes. Once we have it, "which way is downhill?" — the question every learning algorithm asks — has a precise, computable answer.

3.1 Definition: the limit of a difference quotient

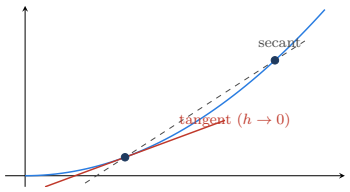
Definition 3.1. The **derivative** of f at x is

$$f'(x) = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h},$$

provided the limit exists. The quotient inside is the slope of the secant line through $(x, f(x))$ and $(x+h, f(x+h))$; the limit is the slope of the **tangent line** at x.

Intuition

As h shrinks, the secant line pivots toward the tangent line, and its slope approaches the derivative. The derivative is the **instantaneous rate of change** — the speedometer reading rather than the trip's average speed — and equivalently the slope of the curve at that point. It is also the best *linear* approximation: near x, $f(x+h) \approx f(x) + f'(x)\,h$.



Example 3.2 (Derivative from the definition)**.** For $f(x) = x^2$:

$$f'(x) = \lim_{h \to 0} \frac{(x+h)^2 - x^2}{h} = \lim_{h \to 0} \frac{2xh + h^2}{h} = \lim_{h \to 0} (2x + h) = 2x.$$

The slope of $y = x^2$ at x is $2x$: zero at the bottom ($x = 0$), increasingly steep as $|x|$ grows.

3.2 Notation

The same idea wears several notations, each handy in context:

$$f'(x) \quad = \quad \frac{\mathrm{d}y}{\mathrm{d}x} \quad = \quad \frac{\mathrm{d}}{\mathrm{d}x}f(x) \quad = \quad Df(x) \quad = \quad \dot{y} \text{ (for time).}$$

The Leibniz form $\frac{\mathrm{d}y}{\mathrm{d}x}$ is suggestive: a ratio of an infinitesimal output change $\mathrm{d}y$ to an input change $\mathrm{d}x$. It is not literally a fraction, but it behaves like one in ways that make the chain rule and substitution feel natural.

3.3 Differentiability and continuity

Theorem 3.3. If f is differentiable at x, then f is continuous at x. The converse is false: continuity does not imply differentiability.

Intuition

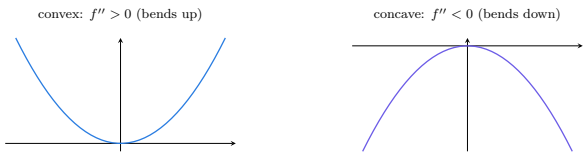
Differentiable means *locally straight*: zoom in far enough and the graph is indistinguishable from its tangent line. A function can be continuous (unbroken) yet fail this at sharp **corners** or vertical tangents. The absolute value $|x|$ is continuous everywhere but has no derivative at 0 — the left slope is -1, the right slope is $+1$, and they disagree.

Common pitfall

The ML-critical example is **ReLU**, $\max(0, x)$: continuous, differentiable everywhere except the corner at 0. Frameworks simply pick a *subgradient* (usually 0) at the kink, which works fine in practice. The lesson: differentiability can fail at isolated points without breaking gradient-based learning, but you should know where the kinks are.

3.4 Higher-order derivatives

Differentiating the derivative gives the **second derivative** $f''(x) = \dfrac{\mathrm{d}^2y}{\mathrm{d}x^2}$, the rate of change of the slope — i.e. **curvature/concavity**. If $f(t)$ is position, f' is velocity and f'' is acceleration. The sign of f'' tells you whether the graph bends up (convex, $f'' > 0$) or down (concave, $f'' < 0$), which is exactly the second-derivative test for minima and maxima (Chapter 5) and, in many variables, the curvature of the loss surface (the Hessian).



3.5 The derivative as a function and as a rate

Computing $f'(x)$ at every x produces a new function, the **derivative function**. Its readings are rates of change with units of "output per input": a velocity (m/s), a marginal cost (\$/unit), a reaction rate, or — in ML — the sensitivity of a loss to a parameter. That last reading is the one that matters most to us.

How this powers ML / AI

Training a model means minimizing a loss $L(\boldsymbol{\theta})$. The derivative $\partial L/\partial\theta_i$ answers: "if I increase parameter θ_i a little, does the loss go up or down, and how fast?" Collect these into the **gradient** (Chapter 10) and you know the steepest-descent direction; step against it and the model improves. This is **gradient descent**, and it is nothing but the derivative of Chapter 3 applied to millions of parameters at once. The limit definition is also how *numerical* gradient checks are computed to verify backpropagation.

Try it in NumPy

```python
import numpy as np
f  = lambda x: x**2
df = lambda x: 2*x                    # exact derivative

# central finite difference approximates the derivative
def numeric_deriv(f, x, h=1e-5):
    return (f(x+h) - f(x-h)) / (2*h)

print(numeric_deriv(f, 3.0), df(3.0))   # ~6.0, 6.0
# corner of |x|: left and right slopes disagree at 0 (not differentiable)
g = np.abs
print(numeric_deriv(g, 0.0))            # ~0 by symmetry, but slope undefined at 0
```

Chapter summary

- The **derivative** $f'(x) = \lim_{h\to 0} \frac{f(x+h)-f(x)}{h}$ is the tangent slope / instantaneous rate of change.
- It is the best local linear approximation: $f(x+h) \approx f(x) + f'(x)h$.
- Notations: f', $\frac{\mathrm{d}y}{\mathrm{d}x}$, Df; differentiable $\Rightarrow$ continuous, but not conversely (corners like $|x|$, ReLU).
- The **second derivative** measures curvature/concavity (convex vs concave); it drives the minimum/maximum tests.
- In ML the derivative is the sensitivity of the loss to a parameter — the basis of gradient descent.

4

Differentiation Rules

Level: Core

Computing every derivative from the limit definition would be exhausting. Fortunately, a small set of rules lets us differentiate almost any function mechanically. The most important of these — the **chain rule** — is the mathematical heart of backpropagation, so this chapter is worth real fluency.

4.1 The building-block derivatives

Memorize these; everything else is built from them.

$f(x)$	$f'(x)$	$f(x)$	$f'(x)$
c (constant)	0	$\sin x$	$\cos x$
x^n	nx^{n-1}	$\cos x$	$-\sin x$
e^x	e^x	$\tan x$	$\sec^2 x$
a^x	$a^x \ln a$	$\ln x$	$1/x$
$\sqrt{x}$	$\frac{1}{2\sqrt{x}}$	$\log_a x$	$1/(x \ln a)$

Key idea

Two facts deserve a star. First, $\frac{\mathrm{d}}{\mathrm{d}x} e^x = e^x$: the exponential is its own derivative, the only function (up to scaling) with this property — which is why it governs growth, decay, and the math of probability. Second, $\frac{\mathrm{d}}{\mathrm{d}x} \ln x = \frac{1}{x}$: the logarithm converts the multiplicative into the additive, the reason log-likelihoods turn products of probabilities into sums we can differentiate term by term.

4.2 The algebraic rules

Proposition 4.1. For differentiable f, g and constant c:

$$(cf)' = cf', \qquad (f \pm g)' = f' \pm g',$$

$$\textbf{Product: } (fg)' = f'g + fg', \qquad \textbf{Quotient: } \left(\frac{f}{g}\right)' = \frac{f'g - fg'}{g^2}.$$

Common pitfall

The derivative is linear, so sums split cleanly — but products and quotients do *not*. A frequent error is writing $(fg)' = f'g'$; it is not. Use the product rule "derivative of first times second, plus first times derivative of second." Sign errors in the quotient rule (numerator order matters) are the other classic slip.

Example 4.2. $\frac{\mathrm{d}}{\mathrm{d}x}\big(x^2e^x\big) = (2x)e^x + x^2e^x = e^x(x^2+2x)$, by the product rule.

4.3 The chain rule

The chain rule differentiates **compositions** — and since deep networks are deep compositions, it is the rule that makes deep learning possible.

Theorem 4.3 (Chain rule)**.** If $y = f(u)$ and $u = g(x)$, then

$$\frac{\mathrm{d}y}{\mathrm{d}x} = \frac{\mathrm{d}y}{\mathrm{d}u}\cdot\frac{\mathrm{d}u}{\mathrm{d}x} = f'(g(x))\,g'(x).$$

"Differentiate the outer function (leaving the inside alone), then multiply by the derivative of the inside."

Intuition

The Leibniz notation makes it look like the $\mathrm{d}u$'s cancel, and that is exactly the right mental model: rates of change *multiply* along a chain. If y changes $3\times$ as fast as u, and u changes $2\times$ as fast as x, then y changes $6\times$ as fast as x. Stack a hundred such links and you get a product of a hundred derivatives — the structure of backpropagation, and the origin of vanishing/exploding gradients when those factors are persistently < 1 or > 1.

Example 4.4. $\frac{\mathrm{d}}{\mathrm{d}x}\,e^{-x^2}$: outer e^u with $u = -x^2$, so derivative $= e^{-x^2}\cdot(-2x) = -2x\,e^{-x^2}$.

Example 4.5 (The sigmoid's tidy derivative)**.** For $\sigma(x) = \dfrac{1}{1+e^{-x}}$, the chain and quotient rules give the famously clean

$$\sigma'(x) = \sigma(x)\big(1-\sigma(x)\big).$$

The derivative is expressed entirely through the output — so a network can reuse the forward-pass value during the backward pass, saving computation. This identity is a small reason sigmoids were historically popular.

4.4 Implicit and logarithmic differentiation

When y is tangled with x in an equation (e.g. $x^2 + y^2 = 1$), **implicit differentiation** differentiates both sides with respect to x, treating y as a function of x and applying the chain rule to y-terms, then solves for $\frac{\mathrm{d}y}{\mathrm{d}x}$. **Logarithmic differentiation** — take ln of both sides first — tames products, quotients, and variable exponents like x^x, converting them to sums before differentiating. Both are routine once the chain rule is solid.

4.5 Derivatives of inverse functions

If f and f^{-1} are inverses, their slopes are reciprocals at corresponding points:

$$\big(f^{-1}\big)'(y) = \frac{1}{f'(x)} \quad \text{where } y = f(x).$$

This yields $\frac{\mathrm{d}}{\mathrm{d}x}\ln x = \frac{1}{x}$ from $\frac{\mathrm{d}}{\mathrm{d}x}e^x = e^x$, and the derivatives of inverse trig functions — and it is the calculus behind the change-of-variables and *normalizing-flow* densities mentioned later.

How this powers ML / AI

The chain rule *is* backpropagation. A neural network composes layers $f_L \circ \cdots \circ f_1$; the gradient of the scalar loss with respect to early parameters is a product of per-layer derivatives, computed efficiently from output back to input (Chapter 13). Activation derivatives from this chapter are used at every step: $\sigma' = \sigma(1-\sigma)$, $\tanh' = 1 - \tanh^2$, and $\mathrm{ReLU}'(x) = \mathbb{1}[x > 0]$. Persistent factors below 1 cause **vanishing gradients**; above 1, **exploding gradients** — both are direct consequences of the chain rule's multiplication.

Try it in NumPy

```python
import torch
x = torch.tensor(0.7, requires_grad=True)
y = torch.sigmoid(x)
y.backward()                                    # autodiff applies the chain rule
print(x.grad, (y*(1-y)).detach())               # match: sigma'(x) = sigma(1-sigma)

x2 = torch.tensor(1.3, requires_grad=True)
(torch.exp(-x2**2)).backward()
print(x2.grad, (-2*1.3*torch.exp(-torch.tensor(1.3)**2)))   # -2x e^{-x^2}
```

Chapter summary

- Memorize the building blocks; note $\frac{\mathrm{d}}{\mathrm{d}x}e^x = e^x$ and $\frac{\mathrm{d}}{\mathrm{d}x}\ln x = \frac{1}{x}$.
- Derivatives are linear; use the **product** $(fg)' = f'g + fg'$ and **quotient** rules (not $f'g'$!).
- The **chain rule** $\frac{\mathrm{d}y}{\mathrm{d}x} = f'(g(x))g'(x)$ multiplies rates along a composition — the basis of backprop.
- Implicit, logarithmic, and inverse-function differentiation extend the toolkit.
- Activation derivatives ($\sigma' = \sigma(1-\sigma)$, $\tanh' = 1 - \tanh^2$, $\mathrm{ReLU}' = \mathbb{1}[x > 0]$) and vanishing/-exploding gradients come straight from these rules.

5

Applications of the Derivative

Level: Intermediate

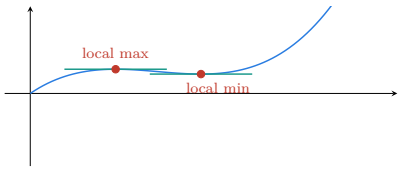
Now that we can compute derivatives, we put them to work. Derivatives find the highest and lowest points of a function (**optimization**), approximate complicated functions by simple ones (**linearization** and Newton's method), and resolve stubborn limits (L'Hôpital). Optimization in particular is the bridge to machine learning: training a model *is* minimizing a function.

5.1 Extrema and critical points

Definition 5.1. A function has a **local maximum** (or minimum) at c if $f(c)$ is the largest (smallest) value nearby. A **critical point** is where $f'(c) = 0$ or f' does not exist. By **Fermat's theorem**, every interior local extremum is a critical point.

Key idea

At a smooth peak or valley the tangent line is horizontal: $f'(c) = 0$. So the search for maxima and minima begins by solving $f'(x) = 0$. This is the one-variable seed of *all* optimization — and in many variables it becomes "set the gradient to zero," the condition every trained model approximately satisfies.



5.1.1 Classifying critical points

- **First-derivative test:** if f' changes $+ \to -$ at c, it is a local max; $- \to +$, a local min.
- **Second-derivative test:** if $f'(c) = 0$ and $f''(c) > 0$ (concave up, a bowl) it is a local min; $f''(c) < 0$ (concave down) a local max; $f''(c) = 0$ is inconclusive.

The second-derivative test is the one-variable ancestor of the **Hessian** test in many variables (Chapter 10), which distinguishes minima, maxima, and saddle points of a loss surface.

5.2 The Mean Value Theorem

Theorem 5.2 (Mean Value Theorem)**.** If f is continuous on $[a, b]$ and differentiable on

(a, b), then there is a point c where the instantaneous rate equals the average rate:

$$f'(c) = \frac{f(b) - f(a)}{b - a}.$$

Intuition

Over any trip, at some instant your speedometer reads exactly your average speed. The MVT is the theoretical workhorse behind many results: it implies that $f' > 0$ on an interval forces f to increase there, that a zero derivative everywhere means a constant function, and it underlies the error bounds for the approximation methods below.

5.3 Monotonicity, concavity, and curve sketching

The first derivative's sign tells you where f rises ($f' > 0$) or falls ($f' < 0$); the second derivative's sign tells you where it is convex ($f'' > 0$) or concave ($f'' < 0$), with **inflection points** where concavity flips. Together they let you sketch a curve's qualitative shape — and, more usefully for us, they describe the geometry of loss functions: convex regions have a single basin, while non-convex regions have multiple basins and saddles.

5.4 Optimization in practice

Real optimization problems ask for the largest or smallest value of some quantity subject to constraints. The recipe: write the objective as a function of one variable, differentiate, solve $f' = 0$, and check endpoints and the second derivative.

Example 5.3 (A classic)**.** Of all rectangles with perimeter 20, which has the largest area? With sides x and $10 - x$, area $A(x) = x(10 - x) = 10x - x^2$. Then $A'(x) = 10 - 2x = 0 \Rightarrow x = 5$, and $A'' = -2 < 0$ confirms a maximum. The square (5×5) wins. Optimization problems in ML are vastly higher-dimensional, but the logic — differentiate, find where the derivative vanishes — is identical.

5.5 Linear approximation and differentials

Because a differentiable function is locally straight, the tangent line is an excellent local stand-in:

$$f(x) \approx f(a) + f'(a)(x - a) \qquad \text{(linearization at } a\text{)}.$$

This is the order-1 Taylor approximation (Chapter 9). It is the basis of error propagation, of the "local linear model" view of a neuron, and — iterated — of Newton's method.

5.6 Newton's method

To solve $f(x) = 0$, follow the tangent line to where it crosses the axis and repeat:

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}.$$

Intuition

Newton's method replaces a hard equation by its tangent-line approximation at each step and solves that instead. When it works it is breathtakingly fast (the number of correct digits roughly doubles each iteration). Applied to $f' = 0$ instead of $f = 0$, it becomes *Newton's method for optimization*, $x_{n+1} = x_n - f'(x_n)/f''(x_n)$ — which in many variables uses the Hessian and is the ancestor of second-order optimizers (Chapter 13).

5.7 L'Hôpital's rule

For indeterminate forms $\frac{0}{0}$ or $\frac{\infty}{\infty}$, derivatives rescue the limit:

$$\lim_{x\to a}\frac{f(x)}{g(x)} = \lim_{x\to a}\frac{f'(x)}{g'(x)}$$

when the right-hand limit exists. For instance $\lim_{x\to 0}\frac{\sin x}{x} = \lim_{x\to 0}\frac{\cos x}{1} = 1$. It quantifies which of two competing quantities wins a race to 0 or ∞ — useful for analyzing asymptotic rates, including how fast algorithms converge.

How this powers ML / AI

This chapter is the conceptual core of model training. Learning is **optimization**: minimize a loss $L(\boldsymbol{\theta})$. The condition $\nabla L = \mathbf{0}$ (the many-variable "$f'(c) = 0$") characterizes the stationary points training seeks; the **second-derivative/Hessian** test classifies them as minima, maxima, or the saddle points that pervade deep networks. **Linearization** is the first-order model behind gradient descent's step; **Newton's method** is the second-order ideal that methods like L-BFGS and K-FAC approximate. Even the learning-rate intuition — small steps along the tangent — is the linear approximation of this chapter in disguise.

Try it in NumPy

```python
import numpy as np
# Newton's method to solve f(x)=0 for f(x)=x^2-2  (root sqrt(2))
f, df = lambda x: x**2-2, lambda x: 2*x
x = 1.0
for _ in range(6):
    x = x - f(x)/df(x)
print(x)                          # 1.4142135623730951

# 1-D gradient descent to MINIMIZE f(x)=(x-3)^2  (uses f', not f)
g = lambda x: 2*(x-3)
x, lr = 0.0, 0.1
for _ in range(60):
    x -= lr*g(x)
print(x)                          # -> 3.0
```

Chapter summary

- Interior extrema occur at **critical points** ($f' = 0$); classify with the first- or second-derivative test.
- The **MVT** links average and instantaneous rates and powers many proofs and error bounds.
- Sign of f' gives increase/decrease; sign of f'' gives convex/concave and inflection points.

- **Linearization** $f(x) \approx f(a) + f'(a)(x - a)$ and **Newton's method** approximate functions and find roots/optima fast.
- **L'Hôpital** resolves $\frac{0}{0}$, $\frac{\infty}{\infty}$; all of this is the one-variable seed of ML optimization.

PART III

Integral Calculus

6

The Integral and the Fundamental Theorem

Level: Core

Integration is the second pillar of calculus: the mathematics of *accumulation*. Where the derivative chops a curve into instantaneous rates, the integral sums infinitely many tiny pieces into a total — an area, a distance, a probability. The chapter's climax is the Fundamental Theorem of Calculus, which reveals that integration and differentiation are inverse operations, turning the hard area problem into easy antidifferentiation.

6.1 Antiderivatives

Definition 6.1. An **antiderivative** of f is a function F with $F' = f$. Since constants differentiate to zero, antiderivatives come in families: the **indefinite integral**

$$\int f(x)\,\mathrm{d}x = F(x) + C,$$

where C is an arbitrary constant. Reading the derivative table backwards gives the basic antiderivatives.

$f(x)$	$\int f\,\mathrm{d}x$	$f(x)$	$\int f\,\mathrm{d}x$
$x^n\ (n \neq -1)$	$\dfrac{x^{n+1}}{n+1} + C$	$\dfrac{1}{x}$	$\ln\lvert x\rvert + C$
e^x	$e^x + C$	$\cos x$	$\sin x + C$
$\sin x$	$-\cos x + C$	$\dfrac{1}{1+x^2}$	$\arctan x + C$

6.2 The definite integral as a limit of sums

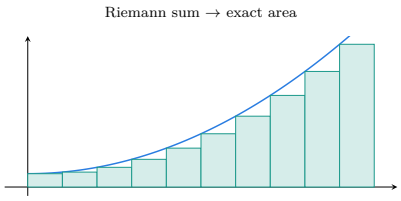
The area problem is solved by approximating with rectangles and refining without bound.

Definition 6.2. Partition $[a, b]$ into n subintervals of width $\Delta x = \frac{b-a}{n}$, pick a sample point x_i^* in each, and form the **Riemann sum** $\sum_{i=1}^{n} f(x_i^*)\,\Delta x$. The **definite integral** is its limit as the rectangles become infinitely thin:

$$\int_a^b f(x)\,\mathrm{d}x = \lim_{n\to\infty} \sum_{i=1}^{n} f(x_i^*)\,\Delta x.$$

Intuition

The integral sign $\int$ is a stretched "S" for "sum," and $\mathrm{d}x$ is the width of an infinitely thin slice: the integral literally adds up $f(x)\,\mathrm{d}x$, "height times width," over the whole interval. If f can be negative, the integral counts area below the axis as negative — it is a *signed* area. This "sum of tiny contributions" picture is exactly how an expected value sums value $\times$ probability in the next part.



6.2.1 Properties

The integral is linear ($\int(af+bg) = a\int f + b\int g$), additive over intervals ($\int_a^c = \int_a^b + \int_b^c$), and monotone ($f \le g \Rightarrow \int f \le \int g$). These mirror the limit/derivative laws and make integrals manipulable.

6.3 The Fundamental Theorem of Calculus

Here is the theorem that ties the whole subject together — arguably the most important in calculus.

Theorem 6.3 (Fundamental Theorem of Calculus)**.** Let f be continuous on $[a,b]$.
Part 1 (differentiation undoes integration). The accumulation function $G(x) = \displaystyle\int_a^x f(t)\,\mathrm{d}t$ is an antiderivative of f: $G'(x) = f(x)$.
Part 2 (integration via antiderivatives). If F is any antiderivative of f, then

$$\int_a^b f(x)\,\mathrm{d}x = F(b) - F(a).$$

Key idea

The FTC says **differentiation and integration are inverse operations**. This is monumental: instead of computing a hard limit of Riemann sums, you find an antiderivative and subtract two values. The area problem and the tangent problem — which looked unrelated in Chapter 1 — are revealed to be two sides of one coin. The rate of accumulation of area under f is exactly f itself.

Example 6.4. $\displaystyle\int_0^2 x^2\,\mathrm{d}x = \left[\frac{x^3}{3}\right]_0^2 = \frac{8}{3} - 0 = \frac{8}{3}$. No Riemann sum needed — just antidifferentiate and subtract.

Intuition

Part 1 is best felt physically. If $f(t)$ is the rate at which water flows into a tank, then $G(x) = \int_a^x f$ is the total water by time x, and its rate of change G' is — of course — the

current flow rate $f(x)$. Accumulating and then measuring the rate of accumulation returns what you started with.

Common pitfall

Three habits prevent most integration errors. (1) Never forget the $+C$ on an *indefinite* integral — losing it loses a whole family of solutions and corrupts differential-equation work. (2) The variable of integration is a dummy: $\int_a^b f(x)\,dx = \int_a^b f(t)\,dt$. (3) The integrand must be well-behaved; integrating across a vertical asymptote without treating it as an *improper* integral (Chapter 7) gives nonsense.

How this powers ML / AI

Integrals are how machine learning handles *continuous probability*. A probability density $p(x)$ must integrate to 1, $\int p(x)\,dx = 1$; probabilities are areas $\int_a^b p(x)\,dx$; and the all-important **expected value** $\mathbb{E}[X] = \int x\,p(x)\,dx$ is an integral (Chapter 8). Losses are typically expectations, hence integrals, which in practice we estimate by averaging samples (Monte Carlo) — a Riemann-sum-like approximation. The FTC's "accumulate a rate" picture also underlies continuous-time models (neural ODEs, diffusion) in the capstone.

Try it in NumPy

```python
import numpy as np
from scipy import integrate
# Riemann sum vs exact integral of x^2 on [0,2] (exact = 8/3)
x = np.linspace(0, 2, 100_000)
print(np.trapz(x**2, x))                    # ~2.6667
val, _ = integrate.quad(lambda t: t**2, 0, 2)
print(val)                                  # 2.6666666...
# a normal density integrates to 1
from math import pi
p = lambda t: np.exp(-t**2/2)/np.sqrt(2*pi)
print(integrate.quad(p, -np.inf, np.inf)[0])   # 1.0
```

Chapter summary

- An **antiderivative** reverses differentiation; the indefinite integral $\int f\,dx = F + C$.
- The **definite integral** is a limit of **Riemann sums** — a signed area / total accumulation.
- The **Fundamental Theorem** makes differentiation and integration inverses: $\int_a^b f = F(b) - F(a)$.
- Always keep $+C$; the variable of integration is a dummy; mind asymptotes (improper integrals).
- Densities, probabilities, and **expectations** are integrals — the continuous backbone of ML, estimated by sampling.

7

Techniques of Integration

Level: Intermediate

Differentiation is mechanical; integration is an art. There is no universal algorithm, so we build a toolkit of techniques — each one reversing a differentiation rule — plus numerical methods for the (many) integrals with no closed form, and **improper integrals** for infinite regions, which is exactly the setting of probability densities.

7.1 Substitution: reversing the chain rule

Theorem 7.1 (u-substitution)**.** If $u = g(x)$ then $\mathrm{d}u = g'(x)\,\mathrm{d}x$, and

$$\int f(g(x))\,g'(x)\;\mathrm{d}x = \int f(u)\;\mathrm{d}u.$$

Intuition

Substitution is the chain rule read backwards: spot an "inside function" $g(x)$ whose derivative also appears, and rename it u to collapse the integral into a basic one. For *definite* integrals, remember to convert the limits to u-values too. This is the single most-used technique.

Example 7.2. $\displaystyle\int 2x\,e^{x^2}\;\mathrm{d}x$: let $u = x^2$, $\mathrm{d}u = 2x\;\mathrm{d}x$, giving $\int e^u\;\mathrm{d}u = e^u + C = e^{x^2} + C$.

7.2 Integration by parts: reversing the product rule

Theorem 7.3 (By parts)**.**

$$\int u\;\mathrm{d}v = uv - \int v\;\mathrm{d}u.$$

Choose u to be the part that *simplifies* when differentiated, and $\mathrm{d}v$ the part you can integrate. The mnemonic **LIATE** (Logs, Inverse-trig, Algebraic, Trig, Exponential) suggests a good u.

Example 7.4. $\displaystyle\int x\,e^x\;\mathrm{d}x$: take $u = x$ ($\mathrm{d}u = \mathrm{d}x$), $\mathrm{d}v = e^x\,\mathrm{d}x$ ($v = e^x$). Then $\int xe^x\,\mathrm{d}x = xe^x - \int e^x\,\mathrm{d}x = xe^x - e^x + C = e^x(x-1) + C$.

7.3 Other standard techniques (overview)

- **Trigonometric integrals & substitutions:** use identities, or substitute $x = \sin\theta, \tan\theta, \sec\theta$ to clear $\sqrt{a^2 \pm x^2}$.
- **Partial fractions:** split a rational function $\frac{P(x)}{Q(x)}$ into simple pieces, each integrable to a log or arctangent. (This is also how one inverts the logistic ODE to derive the sigmoid.)
- **Tables / computer algebra:** for everything else, look it up or use `sympy`.

Common pitfall

Many perfectly innocent-looking integrals have *no* elementary antiderivative — most famously the Gaussian $\int e^{-x^2}\,dx$, which cannot be written with elementary functions (its definite version over $\mathbb{R}$ is the beautiful $\sqrt{\pi}$). This is not a failure of skill; such integrals are evaluated numerically or via special functions. In ML this is the norm, not the exception, which is why we lean on numerical integration and Monte Carlo.

7.4 Numerical integration

When no closed form exists, approximate the area directly. The **trapezoidal rule** replaces the curve by line segments; **Simpson's rule** uses parabolas and is far more accurate for smooth integrands:

$$\int_a^b f\,dx \approx \frac{\Delta x}{2}\left[f_0 + 2f_1 + \cdots + 2f_{n-1} + f_n\right] \text{ (trapezoid).}$$

In high dimensions even these fail (the "curse of dimensionality"), and **Monte Carlo integration** — averaging the integrand at random sample points — becomes the only practical option. That is precisely how expectations are estimated in modern ML.

7.5 Improper integrals

Definition 7.5. An **improper integral** has an infinite limit or an unbounded integrand; it is defined as a limit of proper integrals, e.g.

$$\int_a^\infty f(x)\,dx = \lim_{b\to\infty}\int_a^b f(x)\,dx,$$

and **converges** if that limit is finite, otherwise **diverges**.

Example 7.6. $\int_1^\infty \frac{1}{x^2}\,dx = \lim_{b\to\infty}\left[-\frac{1}{x}\right]_1^b = \lim_{b\to\infty}\left(1-\frac{1}{b}\right) = 1$ (converges), whereas $\int_1^\infty \frac{1}{x}\,dx$ diverges. Whether the tail decays fast enough decides convergence — the same question that decides whether a probability distribution has a finite mean or variance.

How this powers ML / AI

Improper integrals are the natural habitat of probability. Densities live on all of $\mathbb{R}$ (the Gaussian $\frac{1}{\sqrt{2\pi}}e^{-x^2/2}$), so normalization $\int_{-\infty}^{\infty} p = 1$, means $\int x\,p\,dx$, and entropies $-\int p\ln p\,dx$ are all improper integrals. Because most are not elementary (that Gaussian again), ML estimates them with **Monte Carlo**: replace $\mathbb{E}[f(X)] = \int f\,p\,dx$ by the sample average $\frac{1}{N}\sum_i f(x_i)$ with $x_i \sim p$. Mini-batch training, dropout, and variational inference are all Monte Carlo estimates of integrals.

Try it in NumPy

```python
import numpy as np
from scipy import integrate
import sympy as sp

x = sp.symbols('x')
print(sp.integrate(x*sp.exp(x), x))        # x*e^x - e^x  (by parts)
print(sp.integrate(sp.exp(-x**2), (x, -sp.oo, sp.oo)))   # sqrt(pi)

# Monte Carlo estimate of E[X^2] for X ~ N(0,1)  (true value = 1)
samples = np.random.randn(1_000_000)
print(np.mean(samples**2))                 # ~1.0
```

Chapter summary

- **Substitution** reverses the chain rule; **by parts** reverses the product rule ($\int u\,dv = uv - \int v\,du$).
- Trig substitution and partial fractions handle roots and rational functions; many integrals need tables/CAS.
- Some integrals (e.g. $\int e^{-x^2}$) have **no elementary** antiderivative — use numerical or special-function methods.
- **Numerical integration** (trapezoid/Simpson) and, in high dimensions, **Monte Carlo** approximate areas.
- **Improper integrals** (infinite range/integrand) are limits; they govern normalization, means, and entropies in ML.

8

Applications of Integration

Level: Intermediate

Integration is far more than "area under a curve." Any quantity built by accumulating infinitely many infinitesimal pieces is an integral: areas between curves, volumes of solids, lengths of arcs, averages, and — most important for us — the probabilities and expectations of continuous random variables. This chapter shows the pattern and then turns squarely toward the probability that machine learning runs on.

8.1 Area between curves

The area between $y = f(x)$ and $y = g(x)$ (with $f \geq g$) over $[a, b]$ is

$$A = \int_a^b \big[f(x) - g(x)\big]\, \mathrm{d}x.$$

The recipe generalizes: to measure something, slice it into infinitesimal pieces, write the size of one slice, and integrate. This "slice and sum" habit is the real content of applied integration.

8.2 Volumes

Slicing a solid into thin disks or shells gives its volume. A solid of revolution about the x-axis has

$$V = \int_a^b \pi \,[f(x)]^2 \, \mathrm{d}x \quad \text{(disks)},$$

since each slice is a disk of area $\pi r^2 = \pi f(x)^2$ and thickness $\mathrm{d}x$. Other geometries (shells, washers, cross-sections) follow the same slice-and-integrate logic.

8.3 Average value of a function

Definition 8.1. The **average value** of f over $[a, b]$ is

$$\bar{f} = \frac{1}{b-a} \int_a^b f(x)\, \mathrm{d}x.$$

This is the continuous analogue of averaging a list: sum (integrate) the values and divide by the size of the domain. It is the bridge to the expected value — the average of a function weighted by a probability density.

8.4 Probability densities and expectation

This is the application that matters most for AI, so we treat it carefully.

Definition 8.2. A **probability density function** (pdf) $p(x)$ satisfies $p(x) \geq 0$ and $\int_{-\infty}^{\infty} p(x)\,\mathrm{d}x = 1$. The probability that X lands in $[a, b]$ is the area $\mathbb{P}(a \leq X \leq b) = \int_a^b p(x)\,\mathrm{d}x$.

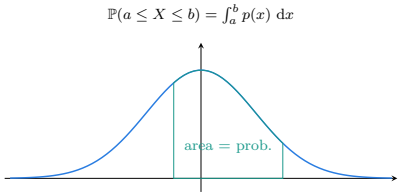
Definition 8.3. The **expected value** (mean) and **variance** of a continuous random variable are integrals:

$$\mathbb{E}[X] = \int_{-\infty}^{\infty} x\,p(x)\,\mathrm{d}x, \qquad \mathrm{Var}[X] = \int_{-\infty}^{\infty} (x - \mathbb{E}[X])^2\,p(x)\,\mathrm{d}x.$$

More generally $\mathbb{E}[g(X)] = \int g(x)\,p(x)\,\mathrm{d}x$ — value times probability, accumulated.

Key idea

An expectation is a *weighted average over a continuum*: each value x contributes $g(x)$, weighted by how likely it is, $p(x)\,\mathrm{d}x$. This single idea — "integrate value against density" — is the form of nearly every loss and objective in probabilistic machine learning. Because the integrals are usually intractable, we estimate them by sampling: $\mathbb{E}[g(X)] \approx \frac{1}{N}\sum_i g(x_i)$.



8.5 Entropy, cross-entropy, and KL divergence

Information theory — the source of classification losses — is built entirely from integrals (or sums, in the discrete case) of a density against its own logarithm:

$$H(p) = -\int p \ln p\,\mathrm{d}x, \quad H(p,q) = -\int p \ln q\,\mathrm{d}x, \quad \mathrm{KL}(p\,\|\,q) = \int p \ln \frac{p}{q}\,\mathrm{d}x.$$

Entropy $H(p)$ measures uncertainty; **cross-entropy** $H(p, q)$ measures the cost of using model q for true distribution p; their difference is the **KL divergence**, a (non-symmetric) "distance" between distributions with KL ≥ 0 and $= 0$ iff $p = q$. Minimizing cross-entropy — the standard classification loss — is minimizing KL between the data and the model.

How this powers ML / AI

The objects above *are* the loss functions of machine learning. Training a classifier minimizes **cross-entropy**; fitting a probabilistic model maximizes a log-likelihood, which is an expectation (integral) over the data; variational methods minimize **KL divergence**; the **ELBO** that trains VAEs is a difference of an expected log-likelihood and a KL term — all integrals (Chapter 13). Since these integrals rarely have closed forms, we replace them by averages over mini-batches and samples — the reason "expectation" and "sample mean" are used almost interchangeably in practice.

Try it in NumPy

```python
import numpy as np
# Expectation by Monte Carlo: E[X^2] for X ~ N(0,1) is 1 (= variance)
x = np.random.randn(1_000_000)
print(x.mean(), (x**2).mean())            # ~0, ~1

# KL divergence between two discrete distributions (sum form of the integral)
p = np.array([0.2, 0.5, 0.3])
q = np.array([0.1, 0.6, 0.3])
kl = np.sum(p * np.log(p / q))
print(kl)                                 # >= 0, zero only if p == q
```

Chapter summary

- Applied integration is "slice into infinitesimal pieces and sum": areas, volumes, arc length, averages.
- The **average value** $\frac{1}{b-a}\int_a^b f$ generalizes to the **expectation** $\mathbb{E}[g(X)] = \int g\,p\,\mathrm{d}x$.
- A **pdf** integrates to 1; probabilities are areas; mean and variance are integrals.
- **Entropy, cross-entropy, KL** are integrals of densities against logs — the basis of classification and variational losses.
- These integrals are usually estimated by **sampling**, linking continuous theory to mini-batch practice.

PART IV

Series and Multivariable Calculus

9

Sequences, Series, and Taylor Series

Level: Advanced

Can you add up infinitely many numbers and get a finite total? Sometimes — and when you can, the result is one of the most powerful tools in mathematics: the ability to represent complicated functions as infinite polynomials. **Taylor series** let us approximate any smooth function by polynomials, which is the basis of linearization, of numerical methods, and of the second-order analysis of optimization.

9.1 Sequences and their limits

A **sequence** $a_1, a_2, a_3, \dots$ is an ordered list; it **converges** to L if $a_n \to L$ as $n \to \infty$ (the limit idea of Chapter 2, on the integers). For example $a_n = 1/n \to 0$, while $a_n = (-1)^n$ does not converge. Convergence of sequences is the foundation for everything that follows — and for the convergence of training algorithms, whose iterates form a sequence we hope settles down.

9.2 Infinite series

Definition 9.1. A **series** $\sum_{n=1}^{\infty} a_n$ is the limit of its **partial sums** $s_N = \sum_{n=1}^{N} a_n$. If $s_N \to S$ (finite), the series **converges** to S; otherwise it **diverges**.

Example 9.2 (Geometric series)**.** The most important series of all:

$$\sum_{n=0}^{\infty} r^n = \frac{1}{1-r} \quad \text{for } |r| < 1, \qquad \text{diverges for } |r| \geq 1.$$

It models compound discounting — and the discounted sum of future rewards in reinforcement learning, $\sum_t \gamma^t r_t$, converges precisely because $0 < \gamma < 1$.

Proposition 9.3 (A few convergence tests)**.**

- ***n*-th term test:** if $a_n \not\to 0$, the series diverges.
- ***p*-series:** $\sum 1/n^p$ converges iff $p > 1$ (so $\sum 1/n$ diverges, $\sum 1/n^2$ converges).
- **Ratio test:** if $\lim |a_{n+1}/a_n| = L < 1$ it converges, > 1 diverges.

Common pitfall

The terms going to zero is *necessary but not sufficient* for convergence: the harmonic series $\sum 1/n$ has terms $\to 0$ yet diverges (to ∞). Convergence is about how *fast* the terms shrink. The same distinction matters for the discount factor and learning-rate schedules:

$\sum \eta_t = \infty$ but $\sum \eta_t^2 < \infty$ is the classic condition guaranteeing stochastic gradient descent converges.

9.3 Power series

A **power series** $\sum_{n=0}^{\infty} c_n (x-a)^n$ is a "polynomial of infinite degree." It converges for x within some **radius of convergence** R of the center a, and inside that interval it defines a function that can be differentiated and integrated term by term — as if it were an ordinary polynomial.

9.4 Taylor and Maclaurin series

Theorem 9.4 (Taylor series)**.** If f is infinitely differentiable near a, then near a

$$f(x) = \sum_{n=0}^{\infty} \frac{f^{(n)}(a)}{n!}(x-a)^n = f(a) + f'(a)(x-a) + \frac{f''(a)}{2}(x-a)^2 + \cdots .$$

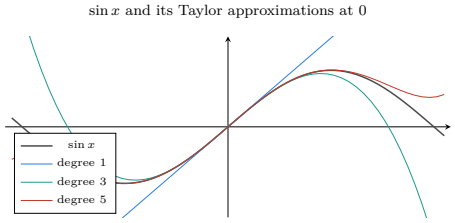
Centered at $a = 0$ it is the **Maclaurin series**.

Key idea

A Taylor series rebuilds a function from its derivatives at a single point. Truncating it gives polynomial approximations of increasing accuracy: the degree-1 truncation is the tangent-line **linearization** (Chapter 5); the degree-2 truncation adds curvature. This "replace a hard function by its low-order polynomial" move is everywhere — it is how calculators evaluate e^x and $\sin x$, how Newton's method and gradient descent are analyzed, and how the Laplace approximation models a posterior as a Gaussian.

Example 9.5 (Three series to know)**.**

$$e^x = \sum_{n=0}^{\infty} \frac{x^n}{n!}, \qquad \sin x = \sum_{n=0}^{\infty} \frac{(-1)^n x^{2n+1}}{(2n+1)!}, \qquad \frac{1}{1-x} = \sum_{n=0}^{\infty} x^n \ (|x| < 1).$$



Intuition

Watch the picture: degree 1 matches the slope at the origin, degree 3 hugs the curve over a wider stretch, degree 5 wider still. Each extra term cancels the next derivative's error. The **multivariable** Taylor expansion (Chapter 10) does the same with a gradient (linear term) and a Hessian (quadratic term) — and that quadratic model is exactly what second-order optimizers minimize at each step.

How this powers ML / AI

Taylor expansion is the analytic lens on optimization. The second-order Taylor model $f(\boldsymbol{\theta}+\Delta) \approx f + \nabla f^\top \Delta + \frac{1}{2}\Delta^\top \mathbf{H}\Delta$ explains gradient descent (use the linear term), Newton's method (minimize the quadratic exactly), and trust-region methods. Series also appear directly: the discounted return $\sum \gamma^t r_t$ in RL is geometric; the logistic and softmax expansions inform numerically stable implementations; positional encodings use sin / cos; and the convergence conditions $\sum \eta_t = \infty$, $\sum \eta_t^2 < \infty$ for SGD are series statements.

Try it in NumPy

```python
import numpy as np
from math import factorial
# Maclaurin series for e^x converges fast
def exp_series(x, terms=12):
    return sum(x**n/factorial(n) for n in range(terms))
print(exp_series(1.0), np.e)                # ~2.71828

# geometric series sum 1 + r + r^2 + ... = 1/(1-r) for |r|<1
r = 0.9
print(sum(r**n for n in range(1000)), 1/(1-r))   # ~10, 10
```

Chapter summary

- A **series** converges if its partial sums have a finite limit; the **geometric** series $\sum r^n = \frac{1}{1-r}$ ($|r| < 1$) is the prototype.
- Terms $\to 0$ is necessary, not sufficient ($\sum 1/n$ diverges); use p-series and ratio tests.
- **Taylor series** rebuild a function from its derivatives; truncations give polynomial approximations (degree 1 = linearization).
- Know $e^x, \sin x, \frac{1}{1-x}$ series; the multivariable version uses gradient + Hessian.
- Series underlie SGD convergence conditions, RL discounted returns, and second-order optimization.

10

Multivariable Calculus: Partial Derivatives and the Gradient

Level: Advanced

Machine-learning models depend on millions of parameters at once, so single-variable calculus is not enough. This chapter extends differentiation to functions of several variables. The star object is the **gradient** — the vector of partial derivatives that points uphill — because following its negative is exactly gradient descent. We also meet the multivariable chain rule, the Jacobian, and the Hessian, the three pillars of training and curvature analysis.

10.1 Functions of several variables

A function $f : \mathbb{R}^n \to \mathbb{R}$ takes a vector $\mathbf{x} = (x_1, \dots, x_n)$ and returns a number — for example a loss $L(\boldsymbol{\theta})$ of all the model's parameters. Its graph is a surface (for $n = 2$) or hypersurface; we visualize it with **contour lines** (level sets where f is constant), exactly like elevation lines on a topographic map.

10.2 Partial derivatives

Definition 10.1. The **partial derivative** $\frac{\partial f}{\partial x_i}$ is the ordinary derivative of f with respect to x_i, treating all other variables as constants:

$$\frac{\partial f}{\partial x_i} = \lim_{h \to 0} \frac{f(\dots, x_i + h, \dots) - f(\dots, x_i, \dots)}{h}.$$

It measures the rate of change of f as you move along the x_i-axis alone.

Example 10.2. For $f(x, y) = x^2 y + \sin y$: $\frac{\partial f}{\partial x} = 2xy$ (treat y as constant), and $\frac{\partial f}{\partial y} = x^2 + \cos y$ (treat x as constant).

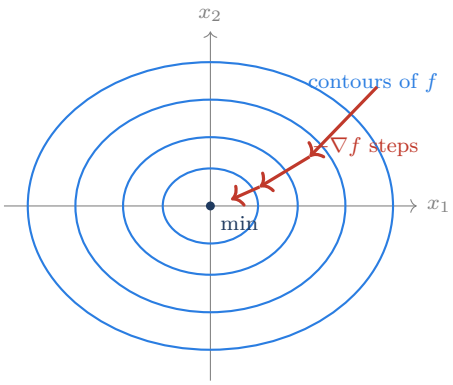
10.3 The gradient

Definition 10.3. The **gradient** of $f : \mathbb{R}^n \to \mathbb{R}$ collects all partial derivatives into a vector:

$$\nabla f = \left[\frac{\partial f}{\partial x_1}, \frac{\partial f}{\partial x_2}, \dots, \frac{\partial f}{\partial x_n} \right]^\top.$$

Key idea

The gradient points in the direction of **steepest ascent**, and its magnitude is the rate of climb in that direction. Therefore $-\nabla f$ points in the direction of steepest *descent*. That one fact is the engine of nearly all machine learning: to minimize a loss, repeatedly step a little in the direction $-\nabla L$. The gradient is perpendicular to the contour lines — always pointing "straight uphill" across them.



10.4 Directional derivatives

The rate of change of f in an arbitrary unit direction $\mathbf{u}$ is the **directional derivative**

$$D_{\mathbf{u}} f = \nabla f \cdot \mathbf{u} = \|\nabla f\| \cos\theta.$$

It is largest when $\mathbf{u}$ aligns with ∇f ($\theta = 0$) — confirming that the gradient is the steepest-ascent direction — and zero along the contour ($\theta = 90°$). The dot-product form ties this directly to the linear-algebra notion of alignment.

10.5 The tangent plane and linearization

Just as a curve has a tangent line, a surface has a **tangent plane**, giving the multivariable linear approximation:

$$f(\mathbf{x} + \Delta) \approx f(\mathbf{x}) + \nabla f(\mathbf{x}) \cdot \Delta.$$

This first-order model is the precise justification of a gradient-descent step: locally, the loss looks linear, and the steepest decrease is along $-\nabla f$.

10.6 The multivariable chain rule and the Jacobian

When f depends on variables that themselves depend on others, rates combine through the **multivariable chain rule**. For a vector-valued map $\mathbf{f} : \mathbb{R}^n \to \mathbb{R}^m$, all first-order information is packed into the **Jacobian** matrix:

$$\mathbf{J} = \frac{\partial \mathbf{f}}{\partial \mathbf{x}}, \qquad J_{ij} = \frac{\partial f_i}{\partial x_j} \quad (m \times n).$$

Intuition

The Jacobian is the best linear approximation of a vector-valued function: locally $\mathbf{f}(\mathbf{x}+\Delta) \approx \mathbf{f}(\mathbf{x}) + \mathbf{J}\Delta$. For a chain of maps the Jacobians *multiply*, exactly as one-variable derivatives did. Backpropagation is this chain rule applied to a deep composition — a product of layer

Jacobians, evaluated efficiently from the output backward (the gradient is one row, so we propagate vectors, never the huge full matrices).

10.7 Second derivatives: the Hessian

Definition 10.4. The **Hessian** of $f : \mathbb{R}^n \to \mathbb{R}$ is the symmetric matrix of second partial derivatives,

$$\mathbf{H}_{ij} = \frac{\partial^2 f}{\partial x_i \, \partial x_j}.$$

It encodes the **curvature** of f in every direction at once (assuming the mixed partials are continuous, $\mathbf{H}$ is symmetric — Clairaut's theorem).

Key idea

The Hessian is the multivariable second-derivative test. At a point where $\nabla f = \mathbf{0}$: if $\mathbf{H}$ is positive definite (all eigenvalues > 0) it is a local **minimum**; negative definite, a **maximum**; indefinite (mixed-sign eigenvalues), a **saddle point**. The eigenvalues of the Hessian are the curvatures along the principal directions, and their ratio — the condition number — governs how fast (or how painfully slowly) gradient descent converges.

The second-order Taylor expansion ties it together:

$$f(\mathbf{x} + \Delta) \approx f(\mathbf{x}) + \nabla f \cdot \Delta + \tfrac{1}{2} \Delta^{\top} \mathbf{H} \, \Delta.$$

Newton's method minimizes this quadratic model exactly, stepping $\Delta = -\mathbf{H}^{-1} \nabla f$.

How this powers ML / AI

This chapter is the mathematical core of training. The **gradient** $\nabla_{\boldsymbol{\theta}} L$ gives the descent direction; **gradient descent** is $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \eta \, \nabla_{\boldsymbol{\theta}} L$. The **Jacobian** and the multivariable chain rule are backpropagation. The **Hessian** explains why deep loss surfaces are hard: in high dimensions, critical points are overwhelmingly *saddles*, not local minima, and ill-conditioned Hessians cause the zig-zagging that momentum and Adam are designed to fix. Newton/quasi-Newton methods (L-BFGS, K-FAC) approximate $\mathbf{H}^{-1}$ because the true Hessian is far too large to form for billions of parameters. Chapter 13 makes all of this explicit.

Try it in NumPy

```
import torch
# gradient and Hessian of f(x,y) = x^2 + 1.6 y^2 at (1.6, 1.4)
def f(v): return v[0]**2 + 1.6*v[1]**2
v = torch.tensor([1.6, 1.4], requires_grad=True)
g = torch.autograd.functional.jacobian(f, v)     # gradient = [2x, 3.2y]
H = torch.autograd.functional.hessian(f, v)       # [[2,0],[0,3.2]]
print(g)                                           # tensor([3.2000, 4.4800])
print(H)                                           # diag(2, 3.2): eigenvalues = curvatures
```

Chapter summary

- **Partial derivatives** differentiate in one variable at a time; the **gradient** ∇f stacks them.
- ∇f points in the direction of **steepest ascent** and is $\perp$ to contours; $-\nabla f$ is steepest descent.
- The **directional derivative** is $\nabla f \cdot \mathbf{u}$; the tangent plane gives the first-order approximation.
- The **Jacobian** (first derivatives of a vector map) multiplies along chains — this is backpropagation.
- The **Hessian** (second derivatives) gives curvature; its definiteness classifies minima/maxima/saddles and sets convergence speed.

11

Multiple Integrals and a Glimpse of Vector Calculus

Level: Advanced

Just as differentiation extended to many variables, so does integration. **Multiple integrals** accumulate a quantity over a region of two, three, or more dimensions — the natural setting for probabilities over several variables. The **change-of-variables** formula introduces the Jacobian determinant, which reappears in normalizing flows; and a brief tour of vector calculus names the operators (gradient, divergence, curl) that close out a standard course.

11.1 Double and triple integrals

Definition 11.1. The **double integral** of $f(x,y)$ over a region R is the limit of sums over a grid of small rectangles:

$$\iint_R f(x,y)\,\mathrm{d}A = \lim \sum_{i,j} f(x_i, y_j)\,\Delta x\,\Delta y.$$

It measures the (signed) volume under the surface $z = f(x,y)$ over R.

By **Fubini's theorem**, a double integral over a rectangle can be evaluated as an **iterated integral** — integrate over x first (holding y fixed), then over y:

$$\iint_R f\,\mathrm{d}A = \int_c^d \left(\int_a^b f(x,y)\,\mathrm{d}x \right) \mathrm{d}y.$$

Triple integrals $\iiint_E f\,\mathrm{d}V$ extend this to solid regions, and the pattern continues to any dimension — which is exactly what a probability over many variables requires.

Example 11.2. $\displaystyle\int_0^1 \int_0^2 (x+y)\,\mathrm{d}x\,\mathrm{d}y = \int_0^1 \left[\tfrac{x^2}{2} + xy\right]_0^2 \mathrm{d}y = \int_0^1 (2+2y)\,\mathrm{d}y = [2y+y^2]_0^1 = 3.$

11.2 Change of variables and the Jacobian

Substitution in many variables rescales the volume element by the absolute value of a determinant.

Theorem 11.3 (Change of variables)**.** Under an invertible map $\mathbf{x} = \mathbf{T}(\mathbf{u})$,

$$\iint_R f(\mathbf{x})\, dA_{\mathbf{x}} = \iint_S f(\mathbf{T}(\mathbf{u}))\, \left|\det \mathbf{J}_{\mathbf{T}}(\mathbf{u})\right|\, dA_{\mathbf{u}},$$

where $\mathbf{J}_{\mathbf{T}}$ is the Jacobian matrix of the transformation. The factor $|\det \mathbf{J}|$ is the local volume-scaling of the map.

Key idea

The Jacobian *determinant* answers "when I bend space with $\mathbf{T}$, by how much does a tiny volume element stretch or shrink?" Polar, cylindrical, and spherical coordinates are the familiar special cases (e.g. $dA = r\, dr\, d\theta$, where r is the Jacobian factor). The very same formula governs how a *probability density* transforms when you change variables — the foundation of normalizing-flow generative models.

Example 11.4 (The Gaussian integral via polar coordinates)**.** The non-elementary $\int_{-\infty}^{\infty} e^{-x^2}\, dx$ is found by squaring it into a double integral and switching to polar coordinates (whose Jacobian is r):

$$\left(\int_{-\infty}^{\infty} e^{-x^2}\, dx\right)^2 = \iint_{\mathbb{R}^2} e^{-(x^2+y^2)}\, dA = \int_0^{2\pi}\int_0^{\infty} e^{-r^2} r\, dr\, d\theta = \pi,$$

so the integral is $\sqrt{\pi}$. This is the normalizing constant behind the Gaussian distribution.

11.3 A glimpse of vector calculus

Vector calculus studies **vector fields** $\mathbf{F} : \mathbb{R}^n \to \mathbb{R}^n$ (think: a velocity or force at every point) with three differential operators:

- **Gradient** ∇f: turns a scalar field into the vector field of steepest ascent (Chapter 10).
- **Divergence** $\operatorname{div} \mathbf{F} = \sum_i \frac{\partial F_i}{\partial x_i}$: the net "outflow" per unit volume — a source (> 0) or sink (< 0).
- **Curl** $\operatorname{curl} \mathbf{F}$: the local rotation of the field.

Intuition

The capstone theorems of a multivariable course — **Green's**, **Stokes'**, and the **Divergence theorem** — all say the same profound thing: the total of a derivative over a region equals a boundary measurement. They are the higher-dimensional siblings of the Fundamental Theorem of Calculus ($\int_a^b F' = F(b) - F(a)$ is exactly "interior derivative = boundary values"). You will not need their full machinery for most of ML, but the divergence shows up in continuous-time generative models, where the change in log-density along a flow is governed by div of the velocity field.

How this powers ML / AI

Multivariable integration is how ML reasons about *joint* distributions. A density over many variables normalizes via a high-dimensional integral; marginalizing a variable integrates it out; and a Bayesian posterior's evidence $p(\text{data}) = \int p(\text{data} \mid \boldsymbol{\theta}) p(\boldsymbol{\theta})\, d\boldsymbol{\theta}$ is a multidimensional integral — usually intractable, hence variational inference and MCMC. The **change-of-variables / Jacobian-determinant** formula is the mathematical core

of **normalizing flows**: transform a simple density through an invertible network and track $\log|\det \mathbf{J}|$. The **divergence** term appears in continuous normalizing flows and the probability-flow ODE of diffusion models (Chapter 13).

Try it in NumPy

```python
import numpy as np
from scipy import integrate
# iterated integral of (x + y) over [0,2] x [0,1]  -> 3
val, _ = integrate.dblquad(lambda y, x: x + y, 0, 2, lambda x: 0,
    lambda x: 1)
print(val)                                      # 3.0
# Gaussian integral over R: sqrt(pi)
print(integrate.quad(lambda x: np.exp(-x**2), -np.inf, np.inf)[0], np
    .sqrt(np.pi))
```

Chapter summary

- **Multiple integrals** accumulate over 2-D/3-D/n-D regions; evaluate by **iterated** (Fubini) integration.
- **Change of variables** multiplies by $|\det \mathbf{J}|$, the local volume-scaling — the Jacobian determinant.
- Polar coordinates give the Gaussian integral $\int e^{-x^2} = \sqrt{\pi}$; the same formula transforms probability densities.
- **Vector calculus** (gradient, divergence, curl) and the Green/Stokes/Divergence theorems generalize the FTC.
- Joint densities, marginalization, Bayesian evidence, and normalizing flows are all multidimensional integration.

12

Optimization and Differential Equations

Level: Expert

This chapter gathers the calculus that most directly becomes machine learning: how to find minima of functions of many variables (with and without constraints), the gradient-descent dynamics that training uses, and the differential equations that describe how systems — and increasingly, generative models — evolve in continuous time. It is the launch pad for the capstone.

12.1 Unconstrained optimization

To minimize $f : \mathbb{R}^n \to \mathbb{R}$, the multivariable analogue of $f' = 0$ is

$$\nabla f(\mathbf{x}^\star) = \mathbf{0} \quad \text{(first-order/stationarity condition)},$$

and the Hessian classifies the stationary point: positive definite ⇒ minimum, negative definite ⇒ maximum, indefinite ⇒ saddle (Chapter 10). Solving $\nabla f = \mathbf{0}$ in closed form is usually impossible for the functions ML cares about, so we descend iteratively instead.

12.2 Gradient descent

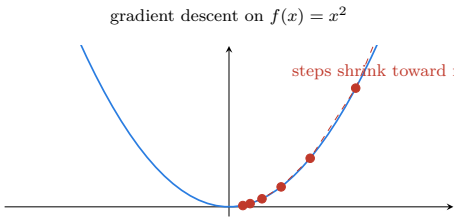
Definition 12.1. Gradient descent minimizes f by repeatedly stepping against the gradient:

$$\mathbf{x}_{t+1} = \mathbf{x}_t - \eta \nabla f(\mathbf{x}_t),$$

where $\eta > 0$ is the **learning rate** (step size).

Intuition

Picture a ball rolling downhill on the surface f. The gradient points uphill, so $-\nabla f$ points to the locally fastest decrease; a small step that way lowers f. The learning rate sets the step length: too small and progress crawls; too large and you overshoot and may diverge. The right scale is tied to curvature — specifically the largest eigenvalue of the Hessian — which is why ill-conditioned problems (long, narrow valleys) make plain gradient descent zig-zag.



12.2.1 Convexity

Definition 12.2. f is **convex** if its graph lies below every chord; equivalently (when smooth) its Hessian is positive semidefinite everywhere. For a convex function, *every* local minimum is global, and $\nabla f = \mathbf{0}$ pinpoints it.

Key idea

Convexity is the dividing line between "easy" and "hard" optimization. Convex losses (linear/logistic regression, SVMs) have a single basin: gradient descent provably reaches the global minimum. Deep networks are *non-convex* — many basins, many saddles — so we settle for good local minima. Remarkably, in very high dimensions most stationary points are saddles rather than bad local minima, and momentum/adaptive methods escape them well enough to train enormous models.

12.3 Constrained optimization: Lagrange multipliers

To optimize $f(\mathbf{x})$ subject to a constraint $g(\mathbf{x}) = 0$, introduce a multiplier λ and solve

$$\nabla f = \lambda \nabla g, \qquad g(\mathbf{x}) = 0.$$

Intuition

At a constrained optimum, you cannot improve f without violating the constraint — which happens exactly when the contour of f is *tangent* to the constraint surface, i.e. their gradients are parallel. The multiplier λ measures how hard the constraint "pushes back" (the shadow price). Inequality constraints generalize this to the KKT conditions, the backbone of support vector machines and constrained ML.

Example 12.3. Maximize entropy $-\sum_i p_i \ln p_i$ subject to $\sum_i p_i = 1$: the Lagrange condition gives $p_i = \text{const}$, i.e. the *uniform* distribution — the maximum-entropy principle. Adding a mean constraint yields the exponential/softmax family.

12.4 Differential equations

A **differential equation** relates a function to its own derivatives — the language of change over time.

Definition 12.4. An **ordinary differential equation** (ODE) has the form $\frac{\mathrm{d}y}{\mathrm{d}t} = F(t, y)$. A **separable** equation $\frac{\mathrm{d}y}{\mathrm{d}t} = g(t)h(y)$ is solved by separating and integrating: $\int \frac{\mathrm{d}y}{h(y)} = \int g(t)\,\mathrm{d}t$.

Example 12.5 (Exponential growth and the logistic curve)**.** $\frac{\mathrm{d}y}{\mathrm{d}t} = ky$ gives $y = y_0 e^{kt}$ (unbounded growth/decay). Adding a saturation term, the **logistic** ODE $\frac{\mathrm{d}y}{\mathrm{d}t} = ky(1-y)$ integrates (via partial fractions) to the sigmoid $y = \frac{1}{1+e^{-kt}}$ — the same S-curve used as a neural activation and as a probability.

When an ODE has no closed-form solution, **Euler's method** steps along the tangent, $y_{t+1} = y_t + \Delta t\, F(t, y_t)$ — structurally identical to a gradient-descent update, and the simplest of the numerical ODE solvers used inside neural ODEs and diffusion samplers.

12.5 A glimpse of the calculus of variations

Where ordinary optimization finds the best *point*, the **calculus of variations** finds the best *function* — the curve minimizing an integral functional, via the Euler–Lagrange equation. It underlies optimal control, the continuous-time view of residual networks, and the optimal-control perspective on diffusion-based generative models.

How this powers ML / AI

This chapter is essentially "the math of training," one level before the capstone. **Gradient descent** and its variants (momentum, RMSProp, **Adam**) are exactly the iteration above with smarter step rules; **convexity** explains which models train reliably; **Lagrange multipliers/KKT** give SVMs and constrained objectives, and the max-entropy derivation yields softmax. **ODEs** appear as neural ODEs (a network defines $\frac{\mathrm{d}\mathbf{h}}{\mathrm{d}t}$ and an ODE solver produces the output) and as the continuous-time formulation of **diffusion models**, whose samplers are ODE/SDE integrators. Euler's method and gradient descent are the same tangent-following idea.

Try it in NumPy

```python
import numpy as np
# Gradient descent on a 2-D quadratic f(x) = x0^2 + 10 x1^2 (ill-conditioned)
grad = lambda x: np.array([2*x[0], 20*x[1]])
x, eta = np.array([5.0, 1.0]), 0.08
for _ in range(50):
    x = x - eta*grad(x)
print(x)                                    # -> near [0, 0]

# Euler's method for dy/dt = k y(1-y): recovers the logistic/sigmoid curve
y, k, dt = 0.01, 1.0, 0.05
ys = [y]
for _ in range(300):
    y = y + dt*(k*y*(1-y)); ys.append(y)
print(round(ys[-1], 4))                     # -> ~1.0 (saturates)
```

Chapter summary

- Minima satisfy $\nabla f = \mathbf{0}$ with positive-(semi)definite Hessian; solve iteratively by **gradient descent** $\mathbf{x}_{t+1} = \mathbf{x}_t - \eta \nabla f$.
- **Convex** problems have one global basin; deep learning is non-convex (saddles dominate in high dimensions).
- **Lagrange multipliers** ($\nabla f = \lambda \nabla g$) solve constrained problems; max-entropy yields the softmax family.
- **ODEs** model continuous change; separable equations integrate directly, and Euler's method steps along tangents.
- These are the direct ancestors of Adam, SVMs, neural ODEs, and diffusion samplers.

PART V

Applications

13

Calculus in Machine Learning, Deep Learning, and AI

Level: Capstone synthesis

Here is the payoff. Linear algebra gives AI its *vocabulary* — vectors, matrices, transformations. Calculus gives it the ability to *learn*. At the heart of almost every modern machine-learning system is one computational pattern: compute a scalar loss by composing many simple functions (the forward pass), take its gradient with respect to billions of parameters (the backward pass), and nudge the parameters downhill (gradient descent). The composing, the differentiating, and the nudging are the three subjects of this book, executed at industrial scale. This chapter assembles them.

13.1 Learning is minimizing a function

A model $f_{\boldsymbol{\theta}}$ has parameters $\boldsymbol{\theta}$; a **loss** $L(\boldsymbol{\theta})$ measures how badly it fits the data. Training solves

$$\boldsymbol{\theta}^\star = \arg\min_{\boldsymbol{\theta}} L(\boldsymbol{\theta}).$$

Because L is a scalar function of many variables, this is precisely the multivariable optimization of Chapter 12: find where the gradient vanishes, approached iteratively by gradient descent. Every concept below serves this single goal.

13.2 The gradient is the learning signal

The gradient $\nabla_{\boldsymbol{\theta}} L$ assembles the loss's sensitivity to each parameter (Chapter 10). It points uphill, so the update

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \eta \, \nabla_{\boldsymbol{\theta}} L$$

moves the model in the direction that decreases the loss fastest. This is the most important equation in machine learning, and it is just the derivative of Chapter 3 applied in millions of dimensions.

13.3 Backpropagation is the chain rule in reverse

A deep network is a composition $L = \ell \circ f_L \circ \cdots \circ f_1$. The multivariable chain rule (Chapter 4, 10) says its gradient is a product of per-layer Jacobians:

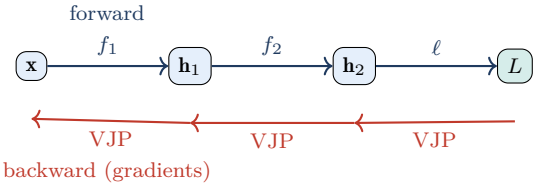
$$\frac{\partial L}{\partial \boldsymbol{\theta}} = \frac{\partial L}{\partial \mathbf{h}_L} \frac{\partial \mathbf{h}_L}{\partial \mathbf{h}_{L-1}} \cdots \frac{\partial \mathbf{h}_1}{\partial \boldsymbol{\theta}}.$$

Key idea

Backpropagation is the chain rule traversed in reverse on a computational graph. Because the loss is a single scalar, the leftmost factor is a row vector; multiplying by Jacobians from the left keeps a *vector* flowing backward, so we never build the enormous full Jacobians — each step is a **vector–Jacobian product** (VJP). This **reverse-mode automatic differentiation** computes the gradient with respect to *all* parameters in roughly the cost of one forward pass, which is exactly why it — and not forward mode — powers deep learning.

Forward vs. reverse mode

Forward-mode AD propagates derivatives input-to-output and is efficient when there are few inputs; reverse-mode propagates output-to-input and is efficient when there is one output (the loss) and many inputs (the parameters). Deep learning is the extreme "many parameters, one scalar loss" regime, so reverse mode wins overwhelmingly. Frameworks build the graph on the forward pass and replay it backward when you call `loss.backward()`.



13.4 Gradients you will derive again and again

A handful of gradients recur across models (consistent with Chapter 10 and the matrix-calculus appendix of the companion volume):

$$\nabla_{\mathbf{x}}(\mathbf{a}^\top \mathbf{x}) = \mathbf{a}, \qquad \nabla_{\mathbf{x}}(\mathbf{x}^\top \mathbf{A}\mathbf{x}) = (\mathbf{A} + \mathbf{A}^\top)\mathbf{x}, \qquad \nabla_{\mathbf{x}} \|\mathbf{x}\|_2^2 = 2\mathbf{x}.$$

Two deserve special attention because nearly every model ends with them.

Example 13.1 (Mean-squared error)**.** For $L = \frac{1}{2}\|\mathbf{X}\boldsymbol{\theta} - \mathbf{y}\|_2^2$ (linear regression), the chain rule gives $\nabla_{\boldsymbol{\theta}} L = \mathbf{X}^\top(\mathbf{X}\boldsymbol{\theta} - \mathbf{y})$. Setting it to zero recovers the normal equations — calculus and linear algebra agreeing exactly.

Example 13.2 (Softmax + cross-entropy)**.** The canonical classification head composes the softmax $p_i = \dfrac{e^{z_i}}{\sum_j e^{z_j}}$ with cross-entropy loss $L = -\sum_i y_i \ln p_i$. Despite the intimidating composition, the gradient with respect to the logits collapses to the beautifully simple

$$\frac{\partial L}{\partial z_i} = p_i - y_i.$$

"Predicted minus true." This clean form — a small calculus miracle — is why softmax + cross-entropy is the default classifier loss: the backward pass is trivial and numerically stable.

13.5 Curvature: Hessians, Newton, and why deep nets are hard

The second-order Taylor model (Chapters 9, 10) $L(\boldsymbol{\theta}+\Delta) \approx L+\nabla L^\top \Delta+\frac{1}{2}\Delta^\top \mathbf{H}\Delta$ explains optimization behavior. The Hessian's eigenvalues are curvatures; their ratio (condition number) sets gradient descent's zig-zag and the stable learning rate. Newton's method steps $\Delta = -\mathbf{H}^{-1}\nabla L$ and converges in few iterations on well-behaved problems — but $\mathbf{H}$ is $n \times n$ for n in the billions, impossible to form or invert, so deep learning uses first-order methods plus curvature approximations (L-BFGS, K-FAC, Adam's diagonal preconditioning).

Key idea

In high dimensions, **saddle points** — where the Hessian has both positive and negative eigenvalues — vastly outnumber poor local minima. This reframed the old fear of "getting stuck in local minima": the real obstacle is plateaus and saddles, which momentum and adaptive methods are designed to escape. Curvature analysis (the Hessian spectrum) is also why normalization and good initialization help: they improve conditioning so gradients descend efficiently.

13.6 Optimizers: smarter ways down the slope

Plain gradient descent is rarely used as-is. The standard variants are all calculus-motivated refinements:

- **Stochastic gradient descent (SGD):** estimate ∇L on a mini-batch — a Monte Carlo estimate (Chapter 8) of the full-data gradient. Convergence needs $\sum \eta_t = \infty$, $\sum \eta_t^2 < \infty$ (Chapter 9).
- **Momentum:** accumulate an exponentially weighted average of past gradients to power through ravines and damp zig-zag.
- **Adam/RMSProp:** scale each coordinate by a running estimate of its gradient magnitude — a cheap per-parameter curvature proxy.

13.7 The integral side: probability, entropy, and generative models

Calculus's integral half runs the probabilistic part of AI (Chapters 8, 11).

- **Losses as expectations.** Risk is $\mathbb{E}[\text{loss}] = \int \text{loss} \cdot p \, dx$, estimated by averaging over data — mini-batch training is Monte Carlo integration.
- **Cross-entropy and KL.** Classification minimizes cross-entropy; variational methods minimize $\mathrm{KL}(q \,\|\, p)$ — integrals of densities against logs.
- **The ELBO and the reparameterization trick.** VAEs maximize the evidence lower bound $\mathbb{E}_q[\ln p(x \mid z)] - \mathrm{KL}(q(z \mid x) \,\|\, p(z))$. To differentiate through the sampling of z, write $z = \mu + \sigma\,\epsilon$ with $\epsilon \sim \mathcal{N}(0, I)$ (the **reparameterization trick**), moving the randomness off the path of differentiation so gradients flow.
- **Continuous-time models.** Neural ODEs define $\frac{d\mathbf{h}}{dt} = f_{\boldsymbol{\theta}}(\mathbf{h}, t)$ and integrate with an ODE solver; **diffusion models** are described by stochastic differential equations whose reverse-time dynamics involve the score $\nabla_{\mathbf{x}} \log p$, and whose log-likelihood uses a divergence term (Chapter 11). Training again reduces to gradients of an integral objective.

13.8 The grand summary table

Calculus concept	Ch.	Where it powers AI
Limits & continuity	2	well-posed losses; saturation; numerical gradient checks
Derivative (rate of change)	3	sensitivity of loss to a parameter
Chain rule	4	backpropagation; vanishing/exploding gradients
Activation derivatives	4	$\sigma' = \sigma(1-\sigma)$, $\tanh'$, ReLU$'$ in every backward pass
Optimization / critical points	5	training objective $\nabla L = \mathbf{0}$; Newton's method
Taylor expansion	9	linearization; 2nd-order optimization; Laplace approx.
Series convergence	9	SGD step-size conditions; RL discounted returns
Integral & expectation	6,8	risk as $\mathbb{E}$[loss]; densities; Monte Carlo
Entropy / cross-entropy / KL	8	classification loss; variational objectives
Multiple integrals / Jacobian det	11	joint densities; normalizing flows; evidence
Gradient	10	gradient descent: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \eta \nabla L$
Jacobian / VJP	10	reverse-mode autodiff (backprop)
Hessian / curvature	10	saddles, conditioning, Adam, L-BFGS, K-FAC
Lagrange multipliers	12	SVMs, constraints, max-entropy $\Rightarrow$ softmax
Differential equations	12	neural ODEs; diffusion samplers; Euler $\approx$ GD

The one-paragraph thesis

Modern AI learns by differentiating. A model composes simple functions into a scalar loss; reverse-mode automatic differentiation — the chain rule on a graph — computes the gradient of that loss with respect to every parameter; an optimizer nudges the parameters downhill; and integrals (expectations, entropies, ELBOs) define the probabilistic objectives being optimized. Derivatives say *which way to move*, integrals say *what to optimize*, and the Fundamental Theorem ties the continuous-time generative models together. Master the concepts in the table and the training loop of any model — from logistic regression to a diffusion transformer — becomes a single, legible idea: calculus at scale.

Where to go next

Pair this with the companion *Linear Algebra* volume (vectors, matrices, the SVD) and you have the full mathematical core of machine learning. The natural next steps are probability and statistics (estimation, Bayes, the Gaussian), then optimization theory, and then the deep-learning curriculum proper — where these derivatives and integrals come alive in convolutional networks, Transformers, and generative models. You now understand the engine; the rest is engineering.

Chapter summary

- Learning is minimizing a loss; the **gradient** is the learning signal and gradient descent the update.
- **Backpropagation** is reverse-mode autodiff — the chain rule as vector–Jacobian products on a graph.
- Key gradients (MSE $\to \mathbf{X}^\top(\mathbf{X}\boldsymbol{\theta} - \mathbf{y})$; softmax+cross-entropy $\to p - y$) recur everywhere.
- The **Hessian** explains saddles, conditioning, and second-order/adaptive optimizers.
- The **integral** side — expectations, entropy/KL, ELBO, ODEs/diffusion — defines and trains probabilistic and generative models.

A

Notation and Formula Reference

A compact reference for the symbols, rules, and formulas used throughout the book.

A.1 Notation

Symbol	Meaning
$\lim_{x \to a} f(x)$	limit of f as $x \to a$
$f'(x)$, $\frac{\mathrm{d}y}{\mathrm{d}x}$, Df	derivative
$f''(x)$, $\frac{\mathrm{d}^2 y}{\mathrm{d}x^2}$	second derivative (curvature)
$\int f \, \mathrm{d}x$	indefinite integral (antiderivative $+C$)
$\int_a^b f \, \mathrm{d}x$	definite integral (signed area)
$\frac{\partial f}{\partial x_i}$	partial derivative
∇f	gradient (vector of partials)
$\mathbf{J}$, $\mathbf{H}$	Jacobian, Hessian
$\mathbb{E}[X]$, $\mathrm{Var}[X]$	expectation, variance (integrals)
$H(p)$, $\mathrm{KL}(p\|q)$	entropy, KL divergence
$\sum_n a_n$	infinite series

A.2 Differentiation rules

$$(cf)' = cf', \quad (f \pm g)' = f' \pm g', \quad (fg)' = f'g + fg', \quad \left(\frac{f}{g}\right)' = \frac{f'g - fg'}{g^2}.$$

Chain rule: $\dfrac{\mathrm{d}}{\mathrm{d}x} f(g(x)) = f'(g(x))\, g'(x).$

A.3 Derivative & integral table

f	f'	f	$\int f \, \mathrm{d}x$
x^n	nx^{n-1}	$x^n \, (n \neq -1)$	$\frac{x^{n+1}}{n+1} + C$
e^x	e^x	e^x	$e^x + C$
$\ln x$	$1/x$	$1/x$	$\ln \lvert x \rvert + C$
$\sin x$	$\cos x$	$\sin x$	$-\cos x + C$
$\cos x$	$-\sin x$	$\cos x$	$\sin x + C$
$\tan x$	$\sec^2 x$	$\frac{1}{1+x^2}$	$\arctan x + C$
$\sigma(x)$	$\sigma(x)(1-\sigma(x))$	$\tanh x$	—

A.4 Integration techniques

$$\textbf{Substitution: } \int f(g(x))g'(x)\,\mathrm{d}x = \int f(u)\,\mathrm{d}u, \qquad \textbf{By parts: } \int u\,\mathrm{d}v = uv - \int v\,\mathrm{d}u.$$

$$\textbf{FTC: } \int_a^b f(x)\,\mathrm{d}x = F(b) - F(a) \ \text{ where } F' = f; \qquad \frac{\mathrm{d}}{\mathrm{d}x}\int_a^x f(t)\,\mathrm{d}t = f(x).$$

A.5 Series

$$\sum_{n=0}^{\infty} r^n = \frac{1}{1-r} \ (|r|<1), \quad e^x = \sum_{n=0}^{\infty} \frac{x^n}{n!}, \quad f(x) = \sum_{n=0}^{\infty} \frac{f^{(n)}(a)}{n!}(x-a)^n.$$

A.6 Multivariable & ML-facing formulas

$$\nabla f = \Big[\frac{\partial f}{\partial x_1}, \ldots, \frac{\partial f}{\partial x_n}\Big]^\top, \qquad \mathbf{H}_{ij} = \frac{\partial^2 f}{\partial x_i \partial x_j}, \qquad f(\mathbf{x}+\Delta) \approx f + \nabla f^\top \Delta + \tfrac{1}{2}\Delta^\top \mathbf{H} \Delta.$$

$$\textbf{Gradient descent: } \boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \eta \nabla_{\boldsymbol{\theta}} L, \qquad \textbf{Newton: } \boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \mathbf{H}^{-1}\nabla_{\boldsymbol{\theta}} L.$$

$$\nabla_{\mathbf{x}}(\mathbf{a}^\top \mathbf{x}) = \mathbf{a}, \quad \nabla_{\mathbf{x}}(\mathbf{x}^\top \mathbf{A}\mathbf{x}) = (\mathbf{A} + \mathbf{A}^\top)\mathbf{x}, \quad \text{softmax+CE: } \frac{\partial L}{\partial z_i} = p_i - y_i.$$

$$\mathbb{E}[g(X)] = \int g(x)p(x)\,\mathrm{d}x, \qquad \mathrm{KL}(p\|q) = \int p \ln \frac{p}{q}\,\mathrm{d}x,$$

$$\mathrm{ELBO} = \mathbb{E}_q[\ln p(x \mid z)] - \mathrm{KL}\left(q(z \mid x) \,\|\, p(z)\right).$$

A.7 Minimal NumPy / PyTorch / SciPy reference

Task	Call
numeric derivative	`(f(x+h)-f(x-h))/(2*h)`
symbolic derivative / integral	`sympy.diff`, `sympy.integrate`
definite integral (numeric)	`scipy.integrate.quad`
autodiff gradient	`loss.backward()`; `torch.autograd.grad`
Jacobian / Hessian	`torch.autograd.functional.jacobian / hessian`
gradient descent step	`theta -= lr * grad`
optimizers	`torch.optim.SGD`, `torch.optim.Adam`

End of book. Derivatives tell AI which way to move; integrals tell it what to optimize.