Calculus & Optimization for Machine Learning

A Complete, Step-by-Step Treatment for the Quant / ML Researcher

Every result derived; every step justified; worked examples throughout

Abstract

Machine learning is, computationally, the minimization of a loss function. This document builds the calculus and optimization theory that makes such minimization possible and trustworthy, from first principles. The discipline of the companion linear-algebra note continues here: *nothing is asserted without explanation.* We derive the gradient and Hessian and explain what each measures; we prove why the gradient is the direction of steepest ascent; we establish exactly why convexity converts a hopeless global search into a tractable local one; we derive the convergence rate of gradient descent and show how the Hessian's conditioning controls it; and we develop Lagrangian duality and the KKT conditions to the point where the sparsity of SVM support vectors falls out as a corollary. Worked numerical examples and explicit ML/finance connections accompany every concept.

Contents

1 Orientation: Optimization Is the Engine of ML

Almost every learning algorithm has the same skeleton: define a loss $L(\boldsymbol{\theta})$ measuring how badly parameters $\boldsymbol{\theta}$ fit the data, then search for $\boldsymbol{\theta}^\star = \arg\min_{\boldsymbol{\theta}} L(\boldsymbol{\theta})$. Linear regression minimizes squared error; logistic regression minimizes cross-entropy; SVMs minimize hinge loss plus a penalty; neural networks minimize whatever differentiable loss you compose. The differences between methods are largely differences in the loss and the constraints; the *machinery* of minimization is shared.

That machinery is multivariate calculus (to know which way is "downhill") wrapped in optimization theory (to know when we have arrived, and how fast we will get there). Two questions organize everything:

- **Local question:** given where we stand, which direction decreases the loss, and by how much? Answered by the gradient and Hessian.
- **Global question:** is the point we converge to actually the best, or merely a local trap? Answered by convexity — and the dividing line between problems where the answer is "best" (classical ML) and "maybe a trap" (deep learning) is exactly convexity.

Intuition

Picture the loss as a landscape over parameter space: height is loss, position is parameters. Optimization is a blindfolded descent — you can feel the local slope (gradient) and curvature (Hessian) under your feet but cannot see the whole terrain. Convexity is the blessed case where the landscape is a single bowl: walk downhill and you *must* reach the bottom. Non-convexity is a mountain range of valleys; downhill walking reaches *a* valley, not necessarily the deepest. Most of this document is about navigating the bowl efficiently and recognizing when you have one.

2 Multivariate Differentiation

2.1 Partial derivatives and the gradient

For $f : \mathbb{R}^n \to \mathbb{R}$, the partial derivative $\partial f/\partial x_i$ is the ordinary derivative in the x_i direction, holding the other coordinates fixed — the slope of the one-dimensional slice. Collecting all n partials gives the **gradient**:

$$\nabla f(\mathbf{x}) = \Big(\frac{\partial f}{\partial x_1}, \ldots, \frac{\partial f}{\partial x_n}\Big)^\top.$$

The gradient is a vector, and its direction and magnitude both carry meaning, which the next result makes precise.

Theorem 1 (The gradient is the direction of steepest ascent)**.** *Among all unit directions* $\mathbf{u}$*, the directional derivative* $D_{\mathbf{u}} f(\mathbf{x}) = \lim_{t\to 0} \frac{f(\mathbf{x}+t\mathbf{u})-f(\mathbf{x})}{t}$ *is maximized by* $\mathbf{u} = \nabla f / \|\nabla f\|$*, and its maximum value is* $\|\nabla f\|$*.*

Proof. By the chain rule (or first-order Taylor), the directional derivative is $D_{\mathbf{u}} f(\mathbf{x}) = \nabla f(\mathbf{x})^\top \mathbf{u} = \langle \nabla f, \mathbf{u} \rangle$. By Cauchy–Schwarz, $\langle \nabla f, \mathbf{u} \rangle \le \|\nabla f\| \, \|\mathbf{u}\| = \|\nabla f\|$ (since $\|\mathbf{u}\| = 1$), with equality iff $\mathbf{u}$ points along ∇f. So the steepest ascent is along ∇f, with slope $\|\nabla f\|$; the steepest *descent* is along $-\nabla f$. □

This theorem is the entire justification for gradient descent. The reason every iterative optimizer steps in the direction $-\nabla f$ is that, locally, no direction decreases the function faster. The magnitude $\|\nabla f\|$ tells you how steep the descent is, and $\nabla f = \mathbf{0}$ signals you are at a flat point — a candidate optimum.

Intuition

The gradient is perpendicular to the level sets (contours) of f. Reason: moving along a contour keeps f constant, so the directional derivative along the contour is zero, i.e. ∇f is orthogonal to the contour's tangent. So gradient descent always steps perpendicular to the current contour — straight across it toward lower ground. When contours are circles, that points right at the minimum; when they are stretched ellipses (ill-conditioned), it points off to the side, and descent zig-zags. Hold this image; it explains both how descent works and why conditioning matters.

Worked Example

Let $f(x, y) = x^2 + 4y^2$ (an elliptical bowl, steeper in y). Then $\nabla f = (2x, 8y)$. At the point $(1, 1)$, $\nabla f = (2, 8)$, so steepest descent heads in direction $-(2, 8)$, dominated by the y-component because the bowl is four times steeper in y. The minimizer is at the origin, but $-(2, 8)$ does *not* point at the origin (which would be direction $-(1, 1)$) — it points too steeply "down the y-wall." This mismatch is the seed of the slow zig-zagging we will quantify with the condition number.

2.2 The Jacobian: derivatives of vector-valued maps

For a vector-valued $\boldsymbol{f} : \mathbb{R}^n \to \mathbb{R}^m$, the derivative is the **Jacobian** matrix $\boldsymbol{J} \in \mathbb{R}^{m \times n}$ with $J_{ij} = \partial f_i / \partial x_j$. Row i is the gradient of the i-th output component. The Jacobian is the best *linear* approximation to $\boldsymbol{f}$ near a point: $\boldsymbol{f}(\mathbf{x} + \boldsymbol{\delta}) \approx \boldsymbol{f}(\mathbf{x}) + \boldsymbol{J}\boldsymbol{\delta}$. When $m = 1$ the Jacobian is the gradient (transposed, as a row). The Jacobian is the object the chain rule multiplies, and — as we will see — backpropagation is the efficient evaluation of a long chain of Jacobians.

2.3 The Hessian: curvature

The second derivatives of a scalar f assemble into the **Hessian** $\mathbf{H} = \nabla^2 f \in \mathbb{R}^{n \times n}$ with $H_{ij} = \partial^2 f / \partial x_i \partial x_j$. It is the Jacobian of the gradient map. When f is twice continuously differentiable, mixed partials commute ($\partial^2 f / \partial x_i \partial x_j = \partial^2 f / \partial x_j \partial x_i$, Clairaut's theorem), so **the Hessian is symmetric** — which means everything we proved about symmetric matrices (real eigenvalues, orthogonal eigenvectors, definiteness) applies to it. This is the bridge from linear algebra: the curvature of a loss surface is a symmetric matrix, and its eigenvalues are the curvatures along principal directions.

Intuition

The Hessian's eigenvalues are the curvatures of the loss along its principal axes. Large eigenvalue = steeply curved (a narrow valley wall); small eigenvalue = gently curved (a flat floor). The ratio of largest to smallest eigenvalue — the condition number of the Hessian — measures how elongated the bowl is, and we will prove it directly controls how slowly gradient descent converges. A positive definite Hessian (all curvatures positive) means a true local minimum; an indefinite one (some positive, some negative curvature) means a saddle.

3 Taylor Expansion: the Local Model of Everything

Optimization works by replacing the true loss with a simple local model — linear (first-order) or quadratic (second-order) — taking a step based on the model, and repeating. Taylor's theorem provides these models with controlled error.

3.1 First- and second-order expansions

For $f : \mathbb{R}^n \to \mathbb{R}$ smooth near $\mathbf{x}$, with step $\boldsymbol{\delta}$:

$$\underbrace{f(\mathbf{x}+\boldsymbol{\delta}) = f(\mathbf{x}) + \nabla f(\mathbf{x})^\top \boldsymbol{\delta} + o(\|\boldsymbol{\delta}\|)}_{\text{first order}},$$

$$\underbrace{f(\mathbf{x}+\boldsymbol{\delta}) = f(\mathbf{x}) + \nabla f(\mathbf{x})^\top \boldsymbol{\delta} + \tfrac{1}{2}\boldsymbol{\delta}^\top \mathbf{H}(\mathbf{x})\boldsymbol{\delta} + o(\|\boldsymbol{\delta}\|^2)}_{\text{second order}}.$$

The first-order model is the tangent plane; gradient descent trusts it for a small step. The second-order model is the best-fitting quadratic bowl; Newton's method jumps to *its* minimum. The remainder terms ($o(\cdot)$) shrink faster than the kept terms, which is what makes the local models trustworthy for small steps and what every convergence proof controls.

Worked Example

For $f(x) = e^x$ at $x = 0$: first-order model $1 + x$; second-order model $1 + x + x^2/2$. At $x = 0.1$: true $e^{0.1} = 1.10517$; first-order 1.1 (error 0.0052); second-order 1.105 (error 0.00017, about 30× smaller). The quadratic model is dramatically more accurate near the expansion point — which is exactly why Newton's method, built on the quadratic model, converges so much faster than gradient descent near an optimum.

3.2 Critical points and the second-order test

A **critical point** is where $\nabla f = \mathbf{0}$ — the first-order model is flat, so it is a candidate optimum. Which kind it is comes from the Hessian, via the second-order expansion (the linear term vanishes at a critical point, leaving $f(\mathbf{x}^\star + \boldsymbol{\delta}) \approx f(\mathbf{x}^\star) + \tfrac{1}{2}\boldsymbol{\delta}^\top \mathbf{H}\boldsymbol{\delta}$):

- $\mathbf{H} \succ 0$ (all eigenvalues > 0): every direction curves up, so f increases in all directions — a strict **local minimum**.
- $\mathbf{H} \prec 0$: every direction curves down — a local **maximum**.
- $\mathbf{H}$ indefinite (mixed signs): up in some directions, down in others — a **saddle point**.
- $\mathbf{H}$ singular (a zero eigenvalue): the test is inconclusive; higher-order terms decide.

The quadratic form $\boldsymbol{\delta}^\top \mathbf{H}\boldsymbol{\delta}$ being the deciding quantity is why positive definiteness — from the linear-algebra note — is the language of optimality.

Intuition

In high dimensions, saddle points vastly outnumber local minima: for a random Hessian, having *all* n eigenvalues positive (a min) is exponentially unlikely compared to a mix of signs (a saddle). This is a central insight of modern non-convex optimization: the obstacle for training deep networks is rarely bad local minima — it is the proliferation of saddles, where the gradient is near zero and progress stalls. The random noise in stochastic gradient descent helps escape them, because a saddle is unstable: there is always a downhill direction to be nudged into.

4 The Chain Rule and Backpropagation

4.1 The multivariate chain rule

If $\boldsymbol{h} : \mathbb{R}^n \to \mathbb{R}^k$ and $\boldsymbol{g} : \mathbb{R}^k \to \mathbb{R}^m$, the composition $\boldsymbol{f} = \boldsymbol{g} \circ \boldsymbol{h}$ has Jacobian given by the **product** of Jacobians:

$$\boldsymbol{J_f}(\mathbf{x}) = \boldsymbol{J_g}(\boldsymbol{h}(\mathbf{x}))\, \boldsymbol{J_h}(\mathbf{x}).$$

This is just the one-dimensional chain rule $f' = g'(h(x))h'(x)$ promoted to matrices, and it composes: a chain of L maps has Jacobian equal to the product of L Jacobians, evaluated at the appropriate intermediate points. The order matters (matrix multiplication is non-commutative) and mirrors the order of composition: outermost map's Jacobian on the left.

Worked Example

Let $z = g(u) = u^2$ and $u = h(x, y) = x + 2y$. Then $\partial z/\partial x = g'(u) \cdot \partial u/\partial x = 2u \cdot 1 = 2(x + 2y)$ and $\partial z/\partial y = 2u \cdot 2 = 4(x + 2y)$. As a Jacobian product: $\boldsymbol{J}_g = [2u]$ (a 1×1), $\boldsymbol{J}_h = [1 \quad 2]$ (a 1×2), product $[2u \quad 4u]$ — matching. The chain rule bookkeeps how a perturbation in inputs propagates through each stage.

4.2 Why backpropagation is the chain rule, organized cleverly

A neural network computes a scalar loss L by composing many layers: $\mathbf{x} \to \boldsymbol{h}_1 \to \boldsymbol{h}_2 \to \cdots \to \boldsymbol{h}_L \to L$. We need $\nabla_{\boldsymbol{\theta}} L$ for all parameters $\boldsymbol{\theta}$. The chain rule gives the gradient as a product of Jacobians — but *the order in which we multiply that product determines the cost*, and this is the entire insight behind backpropagation.

Consider computing $\nabla_{\mathbf{x}} L = \boldsymbol{J}_1^\top \boldsymbol{J}_2^\top \cdots \boldsymbol{J}_L^\top (\nabla_{\boldsymbol{h}_L} L)$ (transposes because we are propagating a gradient, the adjoint of the forward Jacobian). There are two ways to evaluate this chain of matrix products:

- **Forward mode** (multiply right-to-left in the forward direction): propagate a full Jacobian matrix forward. Cost scales with the number of *inputs*.
- **Reverse mode / backpropagation** (multiply left-to-right starting from the scalar output): propagate a *vector* (the gradient so far) backward. Because the output is a single scalar, at every stage you are doing a cheap matrix-times-vector product, never matrix-times-matrix. Cost scales with the number of *outputs* — which is one.

Since ML has one scalar loss but millions of parameters, reverse mode is the overwhelming winner: it computes the entire gradient at the cost of a small constant multiple of one forward pass, independent of the number of parameters.

Intuition

The magic of backprop is not the chain rule (that is just calculus) — it is the *associativity* of matrix multiplication exploited in the right order. Multiplying a chain of Jacobians from the scalar-output end means every intermediate result is a vector (a gradient), so you never pay to form a giant Jacobian matrix. "Backward" is not mystical; it is the statement "a scalar function of many variables is cheapest to differentiate by starting from the scalar." Each backward step also reuses the forward pass's stored activations — trading memory for the enormous compute saving.

5 Convexity: the Great Divide

Whether an optimization problem is easy or hard is, to first approximation, whether it is convex. This section defines convexity three equivalent ways, proves the central theorem (local equals global), and explains why classical ML lives on the easy side.

5.1 Convex sets and convex functions

Definition 1 (Convex set)**.** A set $\mathcal{C} \subseteq \mathbb{R}^n$ is convex if for any two points in it, the entire segment between them stays in it: $\mathbf{x}, \mathbf{y} \in \mathcal{C} \Rightarrow \theta\mathbf{x} + (1-\theta)\mathbf{y} \in \mathcal{C}$ for all $\theta \in [0,1]$.

Definition 2 (Convex function)**.** $f : \mathcal{C} \to \mathbb{R}$ (with $\mathcal{C}$ convex) is convex if its graph lies below every chord:

$$f(\theta\mathbf{x} + (1-\theta)\mathbf{y}) \leq \theta f(\mathbf{x}) + (1-\theta) f(\mathbf{y}), \quad \forall\, \theta \in [0,1].$$

It is *strictly* convex if the inequality is strict for $\mathbf{x} \neq \mathbf{y}$, $\theta \in (0,1)$.

The chord condition says: the function never bulges above the straight line connecting two of its points. Equivalently, the *epigraph* (the region above the graph) is a convex set — linking the two definitions.

5.2 Three equivalent characterizations (for smooth f)

Theorem 2. *For f twice differentiable on a convex domain, the following are equivalent:*

(1) ***Chord:*** $f(\theta\mathbf{x} + (1-\theta)\mathbf{y}) \leq \theta f(\mathbf{x}) + (1-\theta) f(\mathbf{y})$.

(2) ***First-order (tangent below):*** $f(\mathbf{y}) \geq f(\mathbf{x}) + \nabla f(\mathbf{x})^\top (\mathbf{y} - \mathbf{x})$ *for all* $\mathbf{x}, \mathbf{y}$.

(3) ***Second-order (curvature):*** $\mathbf{H}(\mathbf{x}) = \nabla^2 f(\mathbf{x}) \succeq 0$ *everywhere.*

Proof of (3)⇒(2)⇒(1). (3)⇒(2): Taylor with exact (Lagrange) remainder gives $f(\mathbf{y}) = f(\mathbf{x}) + \nabla f(\mathbf{x})^\top (\mathbf{y} - \mathbf{x}) + \frac{1}{2}(\mathbf{y} - \mathbf{x})^\top \mathbf{H}(\boldsymbol{\xi})(\mathbf{y} - \mathbf{x})$ for some $\boldsymbol{\xi}$ on the segment. The last term is ≥ 0 because $\mathbf{H} \succeq 0$, yielding the tangent-below inequality (2).

(2)⇒(1): Let $\mathbf{z} = \theta\mathbf{x} + (1-\theta)\mathbf{y}$. Apply (2) twice, from $\mathbf{z}$ to $\mathbf{x}$ and from $\mathbf{z}$ to $\mathbf{y}$:

$$f(\mathbf{x}) \geq f(\mathbf{z}) + \nabla f(\mathbf{z})^\top (\mathbf{x} - \mathbf{z}), \qquad f(\mathbf{y}) \geq f(\mathbf{z}) + \nabla f(\mathbf{z})^\top (\mathbf{y} - \mathbf{z}).$$

Take the convex combination $\theta \cdot [\text{first}] + (1-\theta) \cdot [\text{second}]$. The gradient terms combine to $\nabla f(\mathbf{z})^\top (\theta\mathbf{x} + (1-\theta)\mathbf{y} - \mathbf{z}) = \nabla f(\mathbf{z})^\top (\mathbf{z} - \mathbf{z}) = 0$, leaving $\theta f(\mathbf{x}) + (1-\theta) f(\mathbf{y}) \geq f(\mathbf{z})$, which is (1). □

Characterization (3) is the one you *check* in practice: compute the Hessian and verify it is PSD. This is why the linear-algebra machinery for definiteness is an optimization prerequisite — convexity is a statement about the Hessian's eigenvalues.

Worked Example

Least squares is convex. $L(\boldsymbol{\beta}) = \|\mathbf{y} - \mathbf{X}\boldsymbol{\beta}\|_2^2$ has Hessian $\nabla^2 L = 2\mathbf{X}^\top \mathbf{X}$, which is PSD (a Gram matrix). So least squares is convex — any stationary point is a global minimum, which is why the normal equations give *the* answer, not *an* answer. It is *strictly* convex iff $\mathbf{X}^\top \mathbf{X} \succ 0$, i.e. $\mathbf{X}$ has full column rank (no multicollinearity); otherwise the minimum is a flat-bottomed valley (the null space), giving non-unique solutions — the same degeneracy we saw in linear algebra, now seen as a loss of strict convexity.

5.3 The theorem that makes convex optimization tractable

Theorem 3 (Local minima are global for convex functions)**.** *If f is convex and $\mathbf{x}^\star$ is a local minimum, then $\mathbf{x}^\star$ is a global minimum. If f is strictly convex, the global minimum is unique.*

Proof. Suppose $\mathbf{x}^\star$ is a local min but some $\mathbf{y}$ has $f(\mathbf{y}) < f(\mathbf{x}^\star)$. Consider points on the segment $\mathbf{z}_\theta = \theta\mathbf{y} + (1-\theta)\mathbf{x}^\star$ for small $\theta > 0$. By convexity,

$$f(\mathbf{z}_\theta) \le \theta f(\mathbf{y}) + (1-\theta) f(\mathbf{x}^\star) = f(\mathbf{x}^\star) + \theta\big(f(\mathbf{y}) - f(\mathbf{x}^\star)\big) < f(\mathbf{x}^\star).$$

But for small enough θ, $\mathbf{z}_\theta$ lies in any neighborhood of $\mathbf{x}^\star$, contradicting that $\mathbf{x}^\star$ is a *local* minimum (we found nearby points with strictly smaller value). So no such $\mathbf{y}$ exists: $\mathbf{x}^\star$ is global. Uniqueness under strict convexity: if two distinct global minima $\mathbf{x}_1, \mathbf{x}_2$ existed with equal value v, the midpoint would have value $< v$ by strict convexity, contradicting that v is the minimum. □

Intuition

This is *the* reason convex problems are "solved." In a convex landscape there are no traps: the local information the gradient provides is globally reliable, so downhill walking provably reaches the best point. The first-order condition $\nabla f = \mathbf{0}$ goes from *necessary* (true at any optimum) to *sufficient* (guarantees a global optimum). Every classical ML method we care about — OLS, ridge, lasso, logistic regression, SVMs — is convex by design, which is why they have reliable, reproducible solutions. Neural networks abandon convexity in exchange for expressive power, and inherit all the difficulty of saddles and local minima as the price.

5.4 Strong convexity and the convexity-preserving operations

f is **μ-strongly convex** if $f(\mathbf{x}) - \frac{\mu}{2}\|\mathbf{x}\|^2$ is still convex, equivalently $\mathbf{H} \succeq \mu\mathbf{I}$ (smallest eigenvalue $\ge \mu > 0$). Strong convexity means the function curves up *at least* as fast as a quadratic with curvature μ — it has a definite "bowl-ness" that, as we will prove, gives gradient descent a fast (geometric) convergence rate.

Convexity is preserved under operations that let you certify complex objectives by construction: nonnegative weighted sums ($\sum_i w_i f_i$ with $w_i \ge 0$), composition with an affine map ($f(\mathbf{A}\mathbf{x} + \mathbf{b})$), and pointwise maximum/supremum ($\max_i f_i$, which is why the hinge loss $\max(0, 1 - yz)$ is convex). Recognizing these closure rules lets you see at a glance that, e.g., ridge regression (convex loss + convex penalty) or an SVM objective (sum of convex hinge losses + convex regularizer) is convex without recomputing a Hessian.

Worked Example

Why the SVM objective is convex. The hinge loss $\ell(z) = \max(0, 1 - z)$ is a pointwise max of two affine (hence convex) functions, so it is convex. Composing with the affine map $z = y(\mathbf{w}^\top\mathbf{x} + b)$ preserves convexity. Summing over data points (nonnegative weights) preserves it. Adding $\frac{\lambda}{2}\|\mathbf{w}\|^2$ (strongly convex) preserves it and makes the whole objective strongly convex in $\mathbf{w}$. Conclusion: a unique global optimum exists — no random restarts, no local-minimum anxiety. This certificate came entirely from the closure rules, with no Hessian computation.

6 Gradient Descent and Its Convergence

6.1 The algorithm and the descent guarantee

Gradient descent iterates $\mathbf{x}_{t+1} = \mathbf{x}_t - \eta\nabla f(\mathbf{x}_t)$, stepping downhill with step size (learning rate) η. The first question is whether a step actually decreases f. It does, provided η is small enough, and "small enough" is dictated by curvature.

Definition 3 (L-smoothness)**.** f is L-smooth if its gradient is Lipschitz: $\|\nabla f(\mathbf{x}) - \nabla f(\mathbf{y})\| \le L\|\mathbf{x} - \mathbf{y}\|$. Equivalently (for twice-differentiable f), $\mathbf{H} \preceq L\mathbf{I}$: no curvature exceeds L.

L-smoothness bounds how fast the gradient can change, which bounds the error of the first-order (linear) model. Quantitatively it gives the **descent lemma**:

$$f(\mathbf{y}) \le f(\mathbf{x}) + \nabla f(\mathbf{x})^\top(\mathbf{y}-\mathbf{x}) + \tfrac{L}{2}\|\mathbf{y}-\mathbf{x}\|^2 .$$

This is Taylor's expansion with the second-order term bounded above by L (since $\mathbf{H} \preceq L\mathbf{I}$). Now substitute the gradient step $\mathbf{y} = \mathbf{x} - \eta\nabla f(\mathbf{x})$:

$$f(\mathbf{x}_{t+1}) \le f(\mathbf{x}_t) - \eta\|\nabla f(\mathbf{x}_t)\|^2 + \tfrac{L\eta^2}{2}\|\nabla f(\mathbf{x}_t)\|^2 = f(\mathbf{x}_t) - \eta\big(1 - \tfrac{L\eta}{2}\big)\|\nabla f(\mathbf{x}_t)\|^2 .$$

For $\eta < 2/L$ the coefficient is positive, so the loss strictly decreases whenever the gradient is nonzero. The optimal fixed step is $\eta = 1/L$, giving the cleanest guaranteed decrease of $\frac{1}{2L}\|\nabla f\|^2$ per step. **This is why the learning rate cannot be arbitrary:** too large ($\eta > 2/L$) and the quadratic overshoot term dominates, the loss *increases*, and the method diverges.

Intuition

The step size battles curvature. L is the steepest curvature anywhere; $1/L$ is the step that exactly accounts for the worst-case overshoot. Set η above $2/L$ and you leap past the bottom of the steepest valley and climb the far wall — divergence. This is the precise, non-mystical reason "the learning rate is too high" blows up training, and why practitioners reduce it when loss diverges.

6.2 Convergence rates: the role of conditioning

How *fast* does gradient descent converge? The answer depends on how "bowl-like" the function is.

Theorem 4 (Linear convergence under strong convexity)**.** *If f is μ-strongly convex and L-smooth, gradient descent with $\eta = 1/L$ satisfies*

$$f(\mathbf{x}_t) - f^\star \le \Big(1 - \tfrac{\mu}{L}\Big)^t \big(f(\mathbf{x}_0) - f^\star\big).$$

The error contracts by a constant factor $(1 - \mu/L)$ each step — **geometric ("linear" on a log scale) convergence**. The contraction factor is governed by $\kappa = L/\mu$, the **condition number** of the problem (ratio of largest to smallest curvature). When κ is small (well-conditioned, near-spherical bowl), $1 - 1/\kappa$ is small and convergence is fast. When κ is huge (ill-conditioned, elongated valley), $1 - 1/\kappa \approx 1$ and convergence crawls: you need $O(\kappa \log(1/\epsilon))$ iterations for accuracy ϵ.

Intuition

This is the exact same condition number from the linear-algebra note, now governing optimization speed instead of numerical error — because for a quadratic $f(\mathbf{x}) = \frac{1}{2}\mathbf{x}^\top\mathbf{H}\mathbf{x}$, the curvature *is* the Hessian and $\kappa = \lambda_{\max}(\mathbf{H})/\lambda_{\min}(\mathbf{H})$. The elongated-ellipse picture from earlier makes it visceral: in a stretched bowl, the negative gradient points across the valley rather than along it, so descent bounces between the walls, making slow progress along the valley floor. **Preconditioning** — rescaling coordinates to make the bowl rounder, e.g. by standardizing features so each has comparable curvature — directly lowers κ and is one of the highest-leverage practical tricks. Feature scaling is not cosmetic; it changes the convergence rate.

Worked Example

Minimize $f(x, y) = \frac{1}{2}(x^2 + 100y^2)$, so $\mathbf{H} = \mathrm{diag}(1, 100)$, $\kappa = 100$. Contraction factor $\approx 1 - 1/100 = 0.99$ per step — to cut the error by $10\times$ takes about $\ln(10)/\ln(1/0.99) \approx 229$ iterations. Now rescale $y' = 10y$: the function becomes $\frac{1}{2}(x^2 + y'^2)$, $\kappa = 1$, and gradient descent converges in *one* step. Same problem, different coordinates, $200\times$ speedup — the entire value

of preconditioning in one example.

For merely convex (not strongly convex) L-smooth f, the rate degrades to *sublinear* $O(1/t)$: $f(\mathbf{x}_t) - f^\star \le \frac{L\|\mathbf{x}_0 - \mathbf{x}^\star\|^2}{2t}$. Slower, but still a guarantee.

7 Stochastic Gradient Descent

7.1 Why stochastic

The ML loss is typically a sum over n data points: $L(\boldsymbol{\theta}) = \frac{1}{n}\sum_{i=1}^{n} \ell_i(\boldsymbol{\theta})$. The full gradient $\nabla L = \frac{1}{n}\sum_i \nabla \ell_i$ costs $O(n)$ per step — prohibitive for large n. **Stochastic gradient descent (SGD)** replaces it with the gradient of a single random example (or a mini-batch): $\nabla \ell_{i_t}$ for a randomly chosen i_t. The key property is *unbiasedness*: $\mathbb{E}_{i_t}[\nabla \ell_{i_t}] = \frac{1}{n}\sum_i \nabla \ell_i = \nabla L$. On average, the stochastic gradient points the right way; each individual step is a noisy estimate.

7.2 The step-size condition

That noise has a consequence: with a *fixed* step size, SGD cannot converge to the exact optimum — near the minimum the true gradient vanishes but the stochastic gradient does not, so steps keep jittering. The classical fix is a **decreasing** step-size schedule satisfying the Robbins–Monro conditions:

$$\sum_t \eta_t = \infty \quad \text{(steps sum to infinity: can travel any distance)}, \qquad \sum_t \eta_t^2 < \infty \quad \text{(squared steps sum finite: noise i}$$

The first condition ensures you can reach the optimum from anywhere; the second ensures the accumulated noise is finite so the iterates settle. A schedule like $\eta_t = \eta_0/t$ satisfies both. In practice constant-then-decayed schedules and adaptive methods are used, but the principle — shrink the step to quiet the noise as you approach — is universal.

Intuition

SGD's noise is a double-edged sword that turns out mostly helpful. Cost: it prevents exact convergence with fixed steps and adds variance. Benefits: (1) it is $n\times$ cheaper per step, so it makes many more updates per unit compute; (2) the noise helps *escape saddle points and sharp minima*, biasing the solution toward flatter minima that often generalize better; (3) mini-batching exploits vectorized hardware. The mini-batch size trades these off: larger batches reduce gradient variance (smoother steps) but cost more per step and lose some of the beneficial noise.

8 Momentum and Adaptive Methods

8.1 Momentum: damping the zig-zag

Plain gradient descent in an ill-conditioned valley oscillates across the narrow direction while creeping along the long one. **Momentum** accumulates a velocity that averages out the oscillation and builds speed along consistent directions:

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

with $\gamma \in [0, 1)$ (typically 0.9). The oscillating component reverses sign each step and cancels in the running sum; the consistent component reinforces and grows. **Nesterov's accelerated gradient**

refines this by evaluating the gradient at a look-ahead point, and provably achieves the optimal $O(1/t^2)$ rate for smooth convex functions (versus $O(1/t)$ for plain GD) — a genuine, not heuristic, acceleration.

8.2 Adaptive learning rates

Different parameters may need different step sizes (some features are frequent, others rare). Adaptive methods scale each coordinate's step by a running measure of its gradient magnitude:

- **AdaGrad** divides by the square root of the accumulated sum of squared gradients — giving large steps to rarely-updated parameters, small steps to frequently-updated ones. Weakness: the accumulator grows without bound, so steps eventually vanish.
- **RMSProp** fixes this with an *exponential moving average* of squared gradients instead of a cumulative sum, keeping the scale responsive.
- **Adam** combines RMSProp's per-coordinate scaling with momentum (a moving average of the gradient itself), plus bias-correction for the early steps when the moving averages are still warming up. It is the de facto default for deep learning because it is robust to a wide range of hyperparameters.

Intuition

Adaptive methods are an approximate, cheap form of preconditioning: dividing by the gradient's typical magnitude per coordinate is a diagonal approximation to rescaling by the Hessian, lowering the effective condition number without the $O(n^3)$ cost of true second-order methods. They trade a little theoretical optimality (Adam can fail to converge on some convex problems) for enormous practical robustness on the non-convex losses of deep learning.

9 Second-Order Methods

9.1 Newton's method

Newton's method minimizes the *quadratic* Taylor model exactly at each step. Setting the gradient of the second-order model to zero gives the step

$$\mathbf{x}_{t+1} = \mathbf{x}_t - \mathbf{H}(\mathbf{x}_t)^{-1}\nabla f(\mathbf{x}_t).$$

Where gradient descent multiplies the gradient by a scalar η, Newton multiplies by the inverse Hessian — it *automatically* accounts for curvature, stretching the step long in flat directions and short in steep ones. Consequences:

- **Quadratic convergence** near the optimum: the number of correct digits roughly doubles each step — vastly faster than gradient descent's geometric rate.
- **Affine invariance:** the iterates are unchanged under linear reparameterization, so there is *no* condition-number penalty and no learning rate to tune. Newton solves the elongated-bowl example in one step regardless of scaling.

The costs: forming and inverting the Hessian is $O(n^3)$ per step and $O(n^2)$ memory — infeasible for millions of parameters — and away from a minimum (where $\mathbf{H}$ may not be PD) the raw Newton step can head uphill or toward a saddle, requiring modifications (damping, trust regions, modifying $\mathbf{H}$ to be PD).

9.2 Quasi-Newton: BFGS and L-BFGS

Quasi-Newton methods get most of Newton's benefit without the Hessian. They build an approximation to $\mathbf{H}^{-1}$ from successive gradient differences — each step's change in gradient reveals curvature

information along the step direction (the secant condition). **BFGS** maintains a dense inverse-Hessian approximation; **L-BFGS** stores only the last m gradient/step pairs and reconstructs the action of $\mathbf{H}^{-1}$ implicitly, using $O(mn)$ memory. L-BFGS is the standard solver for medium-scale smooth convex problems — logistic regression, GLMs, conditional random fields — where it dramatically outpaces gradient descent. **Gauss–Newton** and **Levenberg–Marquardt** exploit the special structure of sum-of-squares (least-squares) objectives, approximating the Hessian by $\boldsymbol{J}^\top \boldsymbol{J}$ (dropping a term that is small near a good fit).

Intuition

The hierarchy is a compute–information tradeoff. First-order (GD/SGD): cheap steps, ignores curvature, many iterations, penalized by conditioning. Second-order (Newton): expensive steps, exact curvature, few iterations, immune to conditioning. Quasi-Newton sits between: approximate curvature from gradients you are computing anyway. Adaptive methods (Adam) are a further cheapening — a crude diagonal curvature estimate. Which to use depends on n (parameter count) and whether you can afford $O(n^2)$ memory: huge non-convex nets $\to$ Adam/SGD; medium convex problems $\to$ L-BFGS; small problems $\to$ Newton.

10 Constrained Optimization

Many ML problems are constrained: SVMs constrain margins, probability models constrain parameters to simplices, regularization can be cast as a constraint on coefficient norm. The theory of Lagrange multipliers and KKT conditions tells us how optima behave in the presence of constraints — and, as a bonus, explains the sparsity of SVM support vectors.

10.1 Equality constraints and Lagrange multipliers

Consider $\min f(\mathbf{x})$ subject to $g(\mathbf{x}) = 0$. At a constrained optimum $\mathbf{x}^\star$, you cannot decrease f while staying on the constraint surface $\{g = 0\}$. The directions that stay on the surface (to first order) are those orthogonal to ∇g (since moving along ∇g changes g). For f to be unimprovable along all such directions, its gradient must have *no component* in any of them — i.e. ∇f must be parallel to ∇g:

$$\nabla f(\mathbf{x}^\star) = -\lambda \nabla g(\mathbf{x}^\star) \quad \text{for some multiplier } \lambda.$$

This is captured compactly by defining the **Lagrangian** $\mathcal{L}(\mathbf{x}, \lambda) = f(\mathbf{x}) + \lambda g(\mathbf{x})$ and setting $\nabla_{\mathbf{x}} \mathcal{L} = \mathbf{0}$ (recovers the parallelism) and $\partial \mathcal{L}/\partial \lambda = 0$ (recovers the constraint $g = 0$).

Intuition

The geometric content: at the optimum, the level curves of f are *tangent* to the constraint surface. If they crossed it transversally, you could slide along the constraint to a lower level curve. Tangency means the gradients (each perpendicular to its own level set) are parallel. The multiplier λ is the proportionality constant — and it has a deep economic meaning: it is the **shadow price**, the rate at which the optimal value would improve if you relaxed the constraint by one unit. In a portfolio optimization with a budget constraint, λ is the marginal value of an extra dollar of budget.

Worked Example

Maximize $f(x, y) = xy$ subject to $x + y = 10$. Lagrangian $\mathcal{L} = xy + \lambda(x + y - 10)$. Stationarity: $\partial_x \mathcal{L} = y + \lambda = 0$, $\partial_y \mathcal{L} = x + \lambda = 0$, so $x = y = -\lambda$. The constraint gives $x + y = 10 \Rightarrow x = y = 5$, $\lambda = -5$. Maximum value 25. The shadow price $|\lambda| = 5$ predicts that relaxing the budget to $x + y = 11$ raises the optimum by about 5 (check: new optimum $5.5 \times 5.5 = 30.25$, an increase

of $5.25 \approx 5$). The multiplier is not bookkeeping — it is sensitivity information.

10.2 Inequality constraints and the KKT conditions

Now allow inequality constraints: $\min f(\mathbf{x})$ subject to $g_i(\mathbf{x}) \le 0$ $(i = 1, \dots, m)$ and $h_j(\mathbf{x}) = 0$. Form the Lagrangian $\mathcal{L} = f + \sum_i \mu_i g_i + \sum_j \lambda_j h_j$. The **Karush–Kuhn–Tucker (KKT) conditions** characterize an optimum:

Stationarity:	$\nabla f + \sum_i \mu_i \nabla g_i + \sum_j \lambda_j \nabla h_j = \mathbf{0}$,
Primal feasibility:	$g_i(\mathbf{x}) \le 0, \quad h_j(\mathbf{x}) = 0,$
Dual feasibility:	$\mu_i \ge 0,$
Complementary slackness:	$\mu_i \, g_i(\mathbf{x}) = 0 \quad \forall i.$

The two new ideas relative to equality constraints are dual feasibility and complementary slackness, and both have crisp meanings.

Dual feasibility ($\mu_i \ge 0$): an inequality constraint can only "push" from one side. At an active constraint ($g_i = 0$), the constraint gradient must point in a direction that opposes further decrease of f; the sign restriction $\mu_i \ge 0$ enforces this one-sidedness (an equality constraint, pushable from both sides, has an unrestricted-sign multiplier).

Complementary slackness ($\mu_i g_i = 0$): for each constraint, either it is *active* ($g_i = 0$, the optimum sits on its boundary) and its multiplier μ_i may be positive, or it is *inactive* ($g_i < 0$, slack) and then $\mu_i = 0$ (an inactive constraint exerts no force). A constraint cannot be both slack and exerting force.

Theorem 5 (Sufficiency under convexity)**.** *If f and the g_i are convex, the h_j affine, and a constraint qualification holds (e.g. Slater's condition: some strictly feasible point exists), then the KKT conditions are not just necessary but* sufficient *for global optimality.*

This is why convex constrained problems are solved exactly: find a KKT point and you are done.

10.3 The payoff: SVM support vectors are sparse

Recall the soft-margin SVM: $\min \frac{1}{2}\|\mathbf{w}\|^2 + C\sum_i \xi_i$ subject to $y_i(\mathbf{w}^\top \mathbf{x}_i + b) \ge 1 - \xi_i$ and $\xi_i \ge 0$. Introduce multipliers $\alpha_i \ge 0$ for the margin constraints. Complementary slackness reads $\alpha_i\big[y_i(\mathbf{w}^\top \mathbf{x}_i + b) - 1 + \xi_i\big] = 0$. Parse the cases:

- If a point is *strictly* inside the correct side of the margin, its constraint is slack, so $\alpha_i = 0$: **it contributes nothing to $\mathbf{w} = \sum_i \alpha_i y_i \mathbf{x}_i$.**
- Only points *on* the margin (or violating it) have $\alpha_i > 0$ and enter the solution. These are the **support vectors**.

The sparsity of the SVM — that the decision boundary depends only on a handful of boundary points — is a direct corollary of complementary slackness. No separate argument is needed; the KKT structure forces it. This is the kind of payoff that justifies developing the theory carefully: a deep property of a major algorithm drops out of a general optimality condition.

Intuition

Complementary slackness is the mathematical form of "only the binding constraints matter." In the SVM, only the data points that actively pin down the margin influence the model; the comfortable, well-classified majority are irrelevant to the boundary. The same principle explains why linear programming solutions sit at vertices (where constraints are active) and why, in constrained portfolio optimization, only the binding position limits affect the marginal allocation.

10.4 Lagrangian duality

From the Lagrangian, define the **dual function** $d(\boldsymbol{\mu}, \boldsymbol{\lambda}) = \inf_{\mathbf{x}} \mathcal{L}(\mathbf{x}, \boldsymbol{\mu}, \boldsymbol{\lambda})$. Two facts make duality powerful:

- **The dual is always concave** (it is an infimum of functions affine in $(\boldsymbol{\mu}, \boldsymbol{\lambda})$, and an infimum of affine functions is concave) — regardless of whether the primal is convex. So the dual problem $\max_{\boldsymbol{\mu} \geq 0, \boldsymbol{\lambda}} d$ is always a concave maximization, i.e. tractable.
- **Weak duality:** $d(\boldsymbol{\mu}, \boldsymbol{\lambda}) \leq p^\star$ for all feasible duals, where $p^\star$ is the primal optimum. The dual always lower-bounds the primal. Proof: for feasible $\mathbf{x}$ and $\boldsymbol{\mu} \geq 0$, $\mathcal{L}(\mathbf{x}, \boldsymbol{\mu}, \boldsymbol{\lambda}) = f(\mathbf{x}) + \sum \mu_i g_i + \sum \lambda_j h_j \leq f(\mathbf{x})$ (the $\mu_i g_i \leq 0$ since $\mu_i \geq 0, g_i \leq 0$, and $h_j = 0$); taking the inf over $\mathbf{x}$ on the left and noting it is $\leq f(\mathbf{x}^\star)$ gives the bound.

Under convexity plus Slater's condition, **strong duality** holds: $d^\star = p^\star$, the bound is tight. This is what licenses solving the SVM in its dual form (where the kernel trick lives) instead of the primal — the two give the same answer, and the dual exposes the inner products that kernels generalize.

Worked Example

For the hard-margin SVM, eliminating $\mathbf{w}, b$ from the Lagrangian (via stationarity $\mathbf{w} = \sum_i \alpha_i y_i \mathbf{x}_i$, $\sum_i \alpha_i y_i = 0$) yields the dual $\max_{\boldsymbol{\alpha} \geq 0} \sum_i \alpha_i - \frac{1}{2} \sum_{i,j} \alpha_i \alpha_j y_i y_j \, \mathbf{x}_i^\top \mathbf{x}_j$ subject to $\sum_i \alpha_i y_i = 0$. The data enter *only* through inner products $\mathbf{x}_i^\top \mathbf{x}_j$ — so replacing them with a kernel $K(\mathbf{x}_i, \mathbf{x}_j)$ gives nonlinear SVMs for free. Strong duality guarantees this dual solution reconstructs the primal optimum. The entire kernel machinery is a gift of Lagrangian duality.

10.5 Regularization as constrained optimization

The penalized and constrained forms of regularization are Lagrangian duals of each other. Minimizing $\text{Loss}(\boldsymbol{\beta}) + \lambda \|\boldsymbol{\beta}\|$ (penalized) is equivalent, for a corresponding budget $t(\lambda)$, to minimizing $\text{Loss}(\boldsymbol{\beta})$ subject to $\|\boldsymbol{\beta}\| \leq t$ (constrained). The multiplier on the constraint *is* the penalty weight λ. This duality is why we can interpret ridge/lasso geometrically (loss contours meeting a norm ball, as in the linear-algebra note) and analytically (a penalized objective) interchangeably — they are the same problem viewed through the primal and the constraint.

11 Differentiating Through Expectations

A recurring ML need is the gradient of an expectation $\nabla_{\boldsymbol{\theta}} \mathbb{E}_{p_{\boldsymbol{\theta}}}[f(\mathbf{x})]$ — e.g. in variational inference and policy-gradient reinforcement learning. Two tools:

11.1 The score-function (REINFORCE) estimator

When the distribution depends on $\boldsymbol{\theta}$, swap gradient and integral and use the log-derivative trick $\nabla_{\boldsymbol{\theta}} p_{\boldsymbol{\theta}} = p_{\boldsymbol{\theta}} \nabla_{\boldsymbol{\theta}} \log p_{\boldsymbol{\theta}}$:

$$\nabla_{\boldsymbol{\theta}} \mathbb{E}_{p_{\boldsymbol{\theta}}}[f(\mathbf{x})] = \int f(\mathbf{x}) \nabla_{\boldsymbol{\theta}} p_{\boldsymbol{\theta}}(\mathbf{x}) \, d\mathbf{x} = \mathbb{E}_{p_{\boldsymbol{\theta}}}\left[f(\mathbf{x}) \, \nabla_{\boldsymbol{\theta}} \log p_{\boldsymbol{\theta}}(\mathbf{x})\right].$$

The result is itself an expectation, so it can be estimated by sampling — no need to differentiate f, which may be non-differentiable or a black box (a reward signal). The price is high variance, mitigated by baselines.

11.2 Jensen's inequality and the ELBO

Theorem 6 (Jensen). *For convex φ, $\varphi(\mathbb{E}[X]) \leq \mathbb{E}[\varphi(X)]$ (reversed for concave φ).*

Proof. By the first-order convexity condition (tangent below), $\varphi(X) \geq \varphi(\mathbb{E}X) + \varphi'(\mathbb{E}X)(X - \mathbb{E}X)$. Take expectations of both sides; the last term has expectation zero, leaving $\mathbb{E}[\varphi(X)] \geq \varphi(\mathbb{E}X)$. □

Jensen is the engine of variational inference. The log-evidence $\log p(\mathbf{x}) = \log \mathbb{E}_q[p(\mathbf{x}, \mathbf{z})/q(\mathbf{z})]$ is bounded below, via Jensen (log is concave), by $\mathbb{E}_q[\log p(\mathbf{x}, \mathbf{z})/q(\mathbf{z})]$ — the **evidence lower bound (ELBO)**. Maximizing this tractable lower bound is how we fit latent-variable models (and is exactly what the EM algorithm does, alternating between tightening the bound and maximizing it). The gap in Jensen's inequality equals the KL divergence between the approximate and true posterior — so maximizing the ELBO minimizes that divergence.

11.3 The delta method

A Taylor-expansion tool for the variance of a transformed estimator: if $\sqrt{n}(\hat{\theta} - \theta) \to \mathcal{N}(0, \sigma^2)$, then for smooth g, $\sqrt{n}(g(\hat{\theta}) - g(\theta)) \to \mathcal{N}(0, g'(\theta)^2\sigma^2)$. The derivation is a first-order Taylor expansion $g(\hat{\theta}) \approx g(\theta) + g'(\theta)(\hat{\theta} - \theta)$, whose variance is $g'(\theta)^2 \operatorname{Var}(\hat{\theta})$. This is how standard errors propagate through nonlinear transformations — e.g. from a log-odds estimate to a probability, or from a volatility estimate to an option price (a calculus-driven sensitivity, akin to the option "greeks" that are themselves derivatives of price with respect to inputs).

12 Gradient Descent, Traced by Hand

The convergence theory is abstract until you watch the iterates move. We run gradient descent on a concrete quadratic and see exactly how the step size and conditioning govern the path.

12.1 A well-conditioned quadratic

Minimize $f(\mathbf{x}) = \frac{1}{2}(x_1^2 + x_2^2)$, whose minimum is at the origin with gradient $\nabla f = (x_1, x_2)$. The Hessian is $\mathbf{I}$ (condition number 1 — perfectly conditioned, circular level sets). Gradient descent is $\mathbf{x}_{k+1} = \mathbf{x}_k - \eta \nabla f(\mathbf{x}_k) = (1 - \eta)\mathbf{x}_k$. Starting at $\mathbf{x}_0 = (1, 1)$ with $\eta = 0.5$:

$$\mathbf{x}_1 = (0.5, 0.5),\ \mathbf{x}_2 = (0.25, 0.25),\ \mathbf{x}_3 = (0.125, 0.125), \ldots$$

The iterate halves each step, converging geometrically straight to the origin — no zig-zag, because the circular level sets mean the negative gradient points directly at the minimum. With $\eta = 1$ it would reach the minimum in *one* step (the Newton step for a quadratic); with $\eta = 2$ it would oscillate $(1, 1) \to (-1, -1) \to (1, 1)$ forever (the stability boundary); with $\eta > 2$ it would diverge. This already shows the step size must satisfy $\eta < 2/L$ where L is the largest Hessian eigenvalue (here $L = 1$).

12.2 An ill-conditioned quadratic and the zig-zag

Now $f(\mathbf{x}) = \frac{1}{2}(x_1^2 + 10x_2^2)$ — wait, let us make the conditioning vivid: $f(\mathbf{x}) = \frac{1}{2}(10x_1^2 + x_2^2)$, Hessian $\operatorname{diag}(10, 1)$, condition number $\kappa = 10$. The gradient is $(10x_1, x_2)$. The stable step size is bounded by the *largest* eigenvalue: $\eta < 2/10 = 0.2$. Take $\eta = 0.18$ and start at $\mathbf{x}_0 = (1, 1)$:

$$\mathbf{x}_1 = (1 - 1.8,\ 1 - 0.18) = (-0.8, 0.82), \quad \mathbf{x}_2 = (0.64, 0.672), \quad \mathbf{x}_3 = (-0.512, 0.551), \ldots$$

The x_1 coordinate (steep direction) overshoots and oscillates in sign, while the x_2 coordinate (shallow direction) creeps down slowly. This is the characteristic **zig-zag**: the step size is constrained by the steep direction (to avoid divergence there), but that small step makes the shallow direction crawl. The number of iterations to converge scales with κ — here $10\times$ slower than the well-conditioned case.

Intuition

The two traces show conditioning as the master variable for gradient descent's speed. With circular level sets ($\kappa = 1$) the negative gradient aims straight at the minimum and convergence is immediate. With elongated level sets (large κ) the gradient points mostly across the valley rather than down it, producing the zig-zag, and the step size — pinned by the steep direction — forces the shallow direction to crawl. The iteration count grows like κ (or $\sqrt{\kappa}$ with momentum). This is why preconditioning, feature scaling, batch normalization, and Adam exist: they all reshape the effective level sets toward circular, reducing κ and accelerating convergence. The same condition number from the linear-algebra note governs both numerical stability and optimization speed.

12.3 The convergence rate, made precise

For an L-smooth, μ-strongly-convex function (Hessian eigenvalues in $[\mu, L]$, so $\kappa = L/\mu$), gradient descent with $\eta = 1/L$ satisfies

$$f(\mathbf{x}_k) - f^\star \le \left(1 - \tfrac{1}{\kappa}\right)^k \big(f(\mathbf{x}_0) - f^\star\big).$$

The error contracts by a factor $(1 - 1/\kappa)$ each step — so to reduce the error by a factor ϵ takes about $\kappa \log(1/\epsilon)$ iterations. Worked: with $\kappa = 10$, the factor is 0.9 per step, and reaching $\epsilon = 10^{-3}$ needs roughly $10 \cdot \log(1000) \approx 10 \cdot 6.9 \approx 69$ iterations; with $\kappa = 100$ it needs ≈ 690. The linear (geometric) rate is good news — error falls exponentially in iteration count — but the base depends on κ, which is why ill-conditioning is the enemy. Momentum improves the dependence to $\sqrt{\kappa}$, the source of its acceleration.

13 Taylor Expansion and Newton's Method, Worked

Newton's method uses the second-order Taylor model; computing a step by hand shows why it converges so fast near the optimum.

13.1 The quadratic model

The second-order Taylor expansion of f around $\mathbf{x}_k$ is

$$f(\mathbf{x}) \approx f(\mathbf{x}_k) + \nabla f(\mathbf{x}_k)^\top (\mathbf{x} - \mathbf{x}_k) + \tfrac{1}{2}(\mathbf{x} - \mathbf{x}_k)^\top \mathbf{H} (\mathbf{x} - \mathbf{x}_k).$$

Minimizing this quadratic model (set its gradient to zero) gives the **Newton step** $\mathbf{x}_{k+1} = \mathbf{x}_k - \mathbf{H}^{-1}\nabla f(\mathbf{x}_k)$. Where gradient descent uses the gradient and a scalar step, Newton uses the inverse Hessian, which rescales each direction by its curvature — automatically taking large steps in shallow directions and small steps in steep ones, exactly correcting the zig-zag.

13.2 A worked Newton step

Minimize $f(x) = x^4 - 3x^2 + 2$ (a double-well, but we start near a minimum). $f'(x) = 4x^3 - 6x$, $f''(x) = 12x^2 - 6$. Start at $x_0 = 2$. Newton: $x_1 = x_0 - f'(x_0)/f''(x_0) = 2 - \frac{4 \cdot 8 - 12}{12 \cdot 4 - 6} = 2 - \frac{20}{42} = 2 - 0.476 = 1.524$. Again: $f'(1.524) = 4(3.54) - 9.14 = 14.16 - 9.14 = 5.02$, $f''(1.524) = 12(2.32) - 6 = 21.9$, $x_2 = 1.524 - 5.02/21.9 = 1.524 - 0.229 = 1.295$. Again: converging toward the true minimum at $x^\star = \sqrt{3/2} \approx 1.225$. The steps shrink rapidly — Newton's method has *quadratic* local convergence: the number of correct digits roughly doubles each step once close, far faster than gradient descent's linear rate. This is why Newton and quasi-Newton methods (BFGS, L-BFGS) are preferred when the Hessian is affordable.

Intuition

Newton's method is gradient descent with the perfect, curvature-aware step: by rescaling with $\mathbf{H}^{-1}$, it turns any quadratic into the well-conditioned case and solves it in one step, and near any minimum (where f is approximately quadratic) it converges quadratically. The catch is cost: forming and inverting the Hessian is $O(p^3)$, infeasible for the millions of parameters in deep learning. Quasi-Newton methods (BFGS) approximate $\mathbf{H}^{-1}$ from gradient differences; L-BFGS stores only a few vectors for huge problems; and the diagonal-rescaling of Adam is a cheap, crude curvature correction. The whole zoo of optimizers is a spectrum of how much curvature information they use, trading per-step cost against steps-to-converge — with gradient descent and Newton at the two extremes.

13.3 Why the second-order test classifies critical points

At a critical point ($\nabla f = \mathbf{0}$), the Taylor model reduces to $f(\mathbf{x}) \approx f(\mathbf{x}_k) + \frac{1}{2}\mathbf{d}^\top \mathbf{H}\mathbf{d}$ for a displacement $\mathbf{d}$. The sign of $\mathbf{d}^\top \mathbf{H}\mathbf{d}$ — the quadratic form of the Hessian (linear-algebra note) — determines the local shape: if $\mathbf{H}$ is positive definite (all eigenvalues > 0), every direction curves up, a minimum; negative definite, a maximum; indefinite (mixed signs), a saddle. So classifying a critical point is testing the definiteness of the Hessian, directly reusing the positive-definiteness machinery. In high dimensions, saddles (indefinite Hessians) vastly outnumber minima, which is why escaping saddles — not avoiding local minima — is the central challenge of non-convex deep-learning optimization.

14 Backpropagation, Worked on a Tiny Network

Backpropagation is just the chain rule applied systematically; tracing it on a two-layer network with numbers removes all mystery.

14.1 The network

A scalar example: input $x = 1$, one hidden unit with weight w_1 and ReLU activation, output weight w_2, target $y = 2$, squared loss. Forward pass with $w_1 = 0.5, w_2 = 1.5$:

$$z = w_1 x = 0.5, \quad a = \mathrm{ReLU}(z) = 0.5, \quad \hat{y} = w_2 a = 0.75, \quad L = \tfrac{1}{2}(\hat{y} - y)^2 = \tfrac{1}{2}(0.75 - 2)^2 = 0.781.$$

14.2 The backward pass

We want $\partial L/\partial w_1$ and $\partial L/\partial w_2$. Apply the chain rule, propagating the "error signal" backward:

$$\frac{\partial L}{\partial \hat{y}} = \hat{y} - y = 0.75 - 2 = -1.25.$$

Output weight: $\frac{\partial L}{\partial w_2} = \frac{\partial L}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial w_2} = (-1.25)(a) = (-1.25)(0.5) = -0.625$. **Hidden activation**: $\frac{\partial L}{\partial a} = \frac{\partial L}{\partial \hat{y}} \cdot w_2 = (-1.25)(1.5) = -1.875$. **Pre-activation** (ReLU derivative is 1 since $z > 0$): $\frac{\partial L}{\partial z} = \frac{\partial L}{\partial a} \cdot 1 = -1.875$. **Hidden weight**: $\frac{\partial L}{\partial w_1} = \frac{\partial L}{\partial z} \cdot \frac{\partial z}{\partial w_1} = (-1.875)(x) = (-1.875)(1) = -1.875$.

A gradient-descent step with $\eta = 0.1$: $w_2 \leftarrow 1.5 - 0.1(-0.625) = 1.5625$; $w_1 \leftarrow 0.5 - 0.1(-1.875) = 0.6875$. Recomputing the forward pass with the updated weights gives $\hat{y} = 0.6875 \cdot 1.5625 \cdot \ldots$ closer to 2 — the loss decreased, as a correct gradient step must.

Intuition

Backpropagation is nothing more than the chain rule, organized to reuse intermediate results. The key efficiency is that the "error signal" $\partial L/\partial$(layer output) is computed once per layer and

reused for all that layer's weights — so the cost of all gradients is the same order as one forward pass, not multiplied by the parameter count. The worked numbers show the mechanism: each backward step multiplies the incoming error by a local derivative (the weight, or the activation's derivative). This is why deep networks are trainable at all: the chain rule, bookkept cleverly, makes the gradient of millions of parameters as cheap as a forward evaluation. Every autodiff framework (PyTorch, JAX, TensorFlow) is an industrial-strength implementation of exactly this traced computation.

14.3 Why vanishing/exploding gradients happen

The backward pass multiplies derivatives layer by layer. In a deep network, the error signal at layer k is a product of many such factors. If the per-layer factors are consistently < 1 (e.g. sigmoid derivatives, which max out at 0.25), the product shrinks geometrically — the **vanishing gradient**, and early layers barely learn. If the factors are consistently > 1, the product explodes. This is the chain rule's product structure biting: a depth-D network multiplies D factors, and anything but factors near 1 causes exponential decay or growth. The fixes — ReLU (derivative exactly 1 on the active side), residual connections (adding an identity path so the factor includes a $+1$), and careful initialization and normalization — all aim to keep the per-layer factors near 1 so the product neither vanishes nor explodes. The worked chain rule makes the cause unmistakable.

15 Constrained Optimization and KKT, Worked

Lagrange multipliers and the KKT conditions are the foundation of SVMs and constrained ML; a worked example makes the abstract conditions concrete.

15.1 Equality constraint via Lagrange multipliers

Minimize $f(x, y) = x^2 + y^2$ subject to $x + y = 1$ (find the closest point on the line to the origin). Form the Lagrangian $\mathcal{L} = x^2 + y^2 - \lambda(x + y - 1)$. Stationarity:

$$\partial_x \mathcal{L} = 2x - \lambda = 0, \quad \partial_y \mathcal{L} = 2y - \lambda = 0 \;\Rightarrow\; x = y = \lambda/2.$$

The constraint $x + y = 1$ gives $\lambda = 1$, so $x = y = \frac{1}{2}$. The minimum is at $(\frac{1}{2}, \frac{1}{2})$ with value $\frac{1}{2}$. Geometrically, at the optimum the gradient of f (pointing radially, $(1, 1)$ direction) is parallel to the gradient of the constraint (the normal to the line, also $(1, 1)$) — the Lagrange condition $\nabla f = \lambda \nabla g$ says the objective cannot decrease without violating the constraint. The multiplier $\lambda = 1$ is the *shadow price*: relaxing the constraint to $x + y = 1 + \epsilon$ changes the optimum by $\lambda\epsilon$ to first order.

15.2 Inequality constraints and the KKT conditions

With inequality constraints $g_i(\mathbf{x}) \leq 0$, the **KKT conditions** for a minimum are: stationarity ($\nabla f + \sum_i \mu_i \nabla g_i = \mathbf{0}$), primal feasibility ($g_i \leq 0$), dual feasibility ($\mu_i \geq 0$), and **complementary slackness** ($\mu_i g_i = 0$). The last is the crucial one: for each constraint, either it is active ($g_i = 0$, and μ_i may be positive) or inactive ($g_i < 0$, forcing $\mu_i = 0$). Worked intuition: minimize x^2 subject to $x \geq 1$. The constraint is $g = 1 - x \leq 0$. Unconstrained, the minimum is $x = 0$, which violates $x \geq 1$, so the constraint is active: $x^\star = 1$, with multiplier from stationarity $2x - \mu = 0 \Rightarrow \mu = 2 > 0$ ✓ (dual feasible), and $g = 0$ ✓ (slackness holds). The constraint binds, the multiplier is positive. Had the constraint been $x \geq -1$ (inactive at the unconstrained optimum $x = 0$), we would have $\mu = 0$ and $x^\star = 0$.

Intuition

Complementary slackness is the engine of SVM sparsity. In the SVM dual, each training point has a KKT multiplier α_i, and slackness forces $\alpha_i = 0$ for every point that is not on the margin — so the solution depends only on the few "active" points (the support vectors). The worked logic is exactly this: an inactive constraint has a zero multiplier. The multiplier's other face — the shadow price — is equally important: it measures the sensitivity of the optimum to the constraint, which in the SVM is how much the margin would improve if a support vector were moved, and in regularized learning is the exchange rate between fit and penalty. KKT is the bridge from the optimization note to the SVM note, and the worked example is its smallest complete instance.

15.3 Lagrangian duality and the regularization connection

Every constrained problem has a **dual**: minimizing f subject to $g \le 0$ has the same optimum (under convexity and a constraint qualification) as maximizing the dual function $d(\mu) = \min_{\mathbf{x}} \mathcal{L}(\mathbf{x}, \mu)$ over $\mu \ge 0$. The SVM is solved in its dual precisely because the dual is a clean quadratic program in the multipliers α_i and exposes the kernel trick (the data appear only as inner products). The duality also explains **regularization as constrained optimization**: ridge regression "minimize $\|\mathbf{y} - \mathbf{X}\boldsymbol{\beta}\|^2 + \lambda \|\boldsymbol{\beta}\|^2$" is the Lagrangian form of "minimize $\|\mathbf{y} - \mathbf{X}\boldsymbol{\beta}\|^2$ subject to $\|\boldsymbol{\beta}\|^2 \le t$," with λ the multiplier on the budget constraint. So the penalty parameter λ *is* a shadow price — the rate at which loosening the coefficient budget reduces the loss. This is why every regularizer has a constrained twin, and why the penalty and constraint formulations of ridge, lasso, and the SVM are interchangeable.

Intuition

Duality and the penalty-constraint equivalence unify a great deal of ML. The constrained view ("keep $\|\boldsymbol{\beta}\|$ within a budget") and the penalized view ("add $\lambda \|\boldsymbol{\beta}\|^2$ to the loss") are two faces of one Lagrangian, with λ the multiplier. This is why you can think of ridge either as "shrink coefficients toward zero" or as "stay inside an ℓ_2 ball," and of the lasso as "stay inside an ℓ_1 ball" (whose corners produce sparsity). The dual perspective also delivers the SVM's kernelization and bounds the primal from below (useful for certificates of optimality). Holding both views — primal and dual, penalty and constraint — is what lets a practitioner move fluidly between the geometry of a problem and its algorithm.

16 Convexity, Proven and Applied

Convexity is the property that separates optimization problems we can solve globally from those we cannot. We prove the key facts and apply them.

16.1 Three views of convexity, and why they agree

A differentiable f is convex iff any of these hold: (1) the chord lies above the graph: $f(\lambda\mathbf{x}+(1-\lambda)\mathbf{y}) \le \lambda f(\mathbf{x}) + (1-\lambda) f(\mathbf{y})$; (2) the function lies above its tangents: $f(\mathbf{y}) \ge f(\mathbf{x}) + \nabla f(\mathbf{x})^\top (\mathbf{y} - \mathbf{x})$; (3) the Hessian is positive semidefinite everywhere. The equivalence (2)$\Leftrightarrow$(3) is instructive: the tangent-line inequality says the first-order Taylor approximation is a global *under*-estimate, and the second-order Taylor remainder is $\frac{1}{2}(\mathbf{y} - \mathbf{x})^\top \mathbf{H} (\mathbf{y} - \mathbf{x})$, which is ≥ 0 for all displacements exactly when $\mathbf{H} \succeq 0$. So "lies above its tangents" and "PSD Hessian" are the same statement viewed through the Taylor expansion.

16.2 The theorem that makes convex optimization tractable

For a convex function, every local minimum is a global minimum. Proof: suppose $\mathbf{x}^\star$ is a local but not global minimum, so some $\mathbf{y}$ has $f(\mathbf{y}) < f(\mathbf{x}^\star)$. By convexity, points on the segment $\mathbf{x}^\star + \lambda(\mathbf{y} - \mathbf{x}^\star)$ satisfy $f \le (1-\lambda)f(\mathbf{x}^\star) + \lambda f(\mathbf{y}) < f(\mathbf{x}^\star)$ for all $\lambda \in (0, 1]$ — so arbitrarily close to $\mathbf{x}^\star$ there are points with strictly smaller value, contradicting that $\mathbf{x}^\star$ is a local minimum. Hence no such $\mathbf{y}$ exists, and $\mathbf{x}^\star$ is global. This one theorem is why convex problems (linear/ridge/lasso regression, logistic regression, SVMs) are "solved" — any optimizer that reaches a local minimum has found the answer, with no worry about getting trapped. Non-convex problems (neural networks) lack this guarantee, which is the central practical divide in ML optimization.

16.3 Proving a loss is convex

Logistic loss is convex. The negative log-likelihood for one example is $\ell(\boldsymbol{\beta}) = -y\,\mathbf{x}^\top\boldsymbol{\beta} + \log(1 + e^{\mathbf{x}^\top\boldsymbol{\beta}})$. Its second derivative with respect to the linear predictor $\eta = \mathbf{x}^\top\boldsymbol{\beta}$ is $\sigma(\eta)(1-\sigma(\eta)) > 0$, and composing with the linear map $\eta = \mathbf{x}^\top\boldsymbol{\beta}$ preserves convexity (a convex function of an affine map is convex). The Hessian in $\boldsymbol{\beta}$ is $\sigma(1-\sigma)\mathbf{x}\mathbf{x}^\top \succeq 0$, PSD. Summing over examples preserves convexity (a sum of convex functions is convex). Therefore the logistic objective is convex — guaranteeing the IRLS/Newton iterations of the logistic-regression note converge to the global optimum. The convexity-preserving operations (nonnegative sums, affine composition, pointwise max) are the toolkit for certifying convexity without recomputing Hessians each time.

Intuition

Convexity is the single most consequential property in optimization-based ML. It guarantees a unique global solution reachable by local methods, makes the problem "solved" in a strong sense, and underlies the reliability of linear models, GLMs, and SVMs. The toolkit — check the Hessian is PSD, or build the function from convex pieces using sums, affine composition, and max — lets you certify convexity quickly. The great divide is that neural networks are non-convex: their loss landscapes have saddles and many minima, so the guarantees vanish and optimization becomes an empirical art (good initialization, momentum, normalization, learning-rate schedules). Knowing whether your problem is convex tells you whether you are doing reliable optimization or hopeful search.

17 Stochastic Gradient Descent, Quantified

Full-batch gradient descent computes the exact gradient over all n examples each step — too expensive at scale. SGD uses a random subset, trading gradient accuracy for speed.

17.1 Why a minibatch gradient is unbiased

The full gradient is $\nabla f(\boldsymbol{\beta}) = \frac{1}{n}\sum_i \nabla \ell_i(\boldsymbol{\beta})$. A minibatch of size B drawn uniformly gives $\hat{g} = \frac{1}{B}\sum_{i\in\text{batch}} \nabla \ell_i(\boldsymbol{\beta})$, and $\mathbb{E}[\hat{g}] = \nabla f(\boldsymbol{\beta})$ — unbiased, because each term's expectation is the average gradient. So SGD descends *in expectation*; the noise averages out over many steps. The variance of $\hat{g}$ scales as $1/B$: a batch of B reduces the gradient noise variance by a factor B relative to a single example. This is the bias–variance logic (probability note) applied to gradient estimation — larger batches give lower-variance gradient estimates at higher per-step cost.

17.2 The step-size condition for convergence

SGD's gradient noise does not vanish at the optimum (a single example's gradient is nonzero even when the average is zero), so a constant step size leaves the iterate bouncing in a noise ball around the minimum. To converge exactly, the step size must decay, satisfying the **Robbins–Monro conditions**: $\sum_k \eta_k = \infty$ (steps sum to infinity, so any distance can be covered) and $\sum_k \eta_k^2 < \infty$ (squared steps are summable, so the noise is eventually damped). The canonical schedule $\eta_k = \eta_0/k$ satisfies both. In practice, deep-learning schedules (step decay, cosine annealing, warmup) are engineered approximations: a larger step early for fast progress, decaying later to settle into the minimum. Worked intuition: with constant η, the final error plateaus at a level proportional to $\eta \cdot$ (gradient noise); halving η halves the plateau but doubles the steps to reach it — the speed/accuracy trade that motivates decay.

Intuition

SGD is the workhorse of modern ML because its per-step cost is independent of dataset size, and its noise is not purely a nuisance — the gradient noise helps escape saddle points and sharp minima, acting as an implicit regularizer that may improve generalization. The two facts to hold onto: minibatch gradients are unbiased with variance $\propto 1/B$ (so batch size trades gradient quality for compute), and convergence to the exact optimum requires decaying steps (Robbins–Monro), while constant steps leave a noise ball. The whole practice of learning-rate scheduling is managing this trade — big steps to move fast, small steps to settle — and it is why the learning rate is the single most important hyperparameter in deep learning.

18 Momentum and Adaptive Methods, Mechanically

18.1 Momentum damps the zig-zag

Plain gradient descent zig-zags in ill-conditioned valleys (the worked trace earlier). **Momentum** accumulates a velocity $\mathbf{v}_{k+1} = \beta \mathbf{v}_k + \nabla f(\mathbf{x}_k)$ and steps $\mathbf{x}_{k+1} = \mathbf{x}_k - \eta \mathbf{v}_{k+1}$. The oscillating components (which flip sign each step) cancel in the running average, while the consistent down-valley component accumulates — so momentum speeds the shallow direction and damps the steep oscillation. The effect on the convergence rate is dramatic: from κ iterations to $\sqrt{\kappa}$ (Nesterov's accelerated rate), the optimal first-order rate. The momentum coefficient β (typically 0.9) sets the averaging window: $1/(1-\beta) \approx 10$ steps of memory.

18.2 Adaptive learning rates

AdaGrad/RMSProp/Adam give each parameter its own step size, scaled down for parameters with large historical gradients. Adam maintains a running mean (momentum) and a running mean of squared gradients $\mathbf{v}$ (per-parameter scale), updating $\mathbf{x} \leftarrow \mathbf{x} - \eta\,/(\sqrt{\mathbf{v}} + \epsilon)$. Dividing by $\sqrt{\mathbf{v}}$ is a cheap, diagonal approximation to the Newton rescaling $\mathbf{H}^{-1}$ — it gives steep directions (large $\mathbf{v}$) small steps and shallow directions (small $\mathbf{v}$) large steps, automatically improving conditioning without forming the Hessian. This is why Adam "just works" across many problems with little tuning: it adapts the per-coordinate step to the local curvature scale, mimicking second-order behavior at first-order cost.

Intuition

Momentum and Adam are both curvature corrections in disguise. Momentum averages gradients to cancel oscillation and accelerate, achieving the optimal $\sqrt{\kappa}$ first-order rate. Adam additionally rescales each coordinate by its gradient magnitude, a diagonal stand-in for the inverse Hessian

that fixes per-feature conditioning. Together they explain why modern optimizers are robust where plain SGD struggles: they reshape the effective landscape toward the well-conditioned, circular-level-set ideal that the worked gradient-descent traces showed converges fastest. The progression — gradient descent, momentum, Adam, full Newton — is increasing use of curvature information at increasing cost, and choosing among them is choosing where to sit on that trade for your problem's size and conditioning.

19 Additional Worked Exercises

Worked Example

Exercise. Show that for a quadratic $f(\mathbf{x}) = \frac{1}{2}\mathbf{x}^\top \mathbf{A}\mathbf{x} - \mathbf{b}^\top \mathbf{x}$ with $\mathbf{A} \succ 0$, Newton's method converges in exactly one step from any start.
Solution. The gradient is $\nabla f = \mathbf{A}\mathbf{x} - \mathbf{b}$ and the Hessian is $\mathbf{A}$. The Newton step from any $\mathbf{x}_0$ is $\mathbf{x}_1 = \mathbf{x}_0 - \mathbf{A}^{-1}(\mathbf{A}\mathbf{x}_0 - \mathbf{b}) = \mathbf{x}_0 - \mathbf{x}_0 + \mathbf{A}^{-1}\mathbf{b} = \mathbf{A}^{-1}\mathbf{b}$. The true minimum (setting $\nabla f = \mathbf{0}$) is $\mathbf{x}^\star = \mathbf{A}^{-1}\mathbf{b}$, so $\mathbf{x}_1 = \mathbf{x}^\star$ exactly — one step, regardless of starting point. This is because the second-order Taylor model is *exact* for a quadratic, so minimizing the model minimizes f. It is also why Newton converges quadratically near any minimum: close to the optimum, smooth functions look quadratic, and each step nearly solves the local quadratic exactly. The price is the $O(p^3)$ Hessian inversion, motivating quasi-Newton approximations for large p.

Worked Example

Exercise. A function is L-smooth (gradient is L-Lipschitz). Show gradient descent with $\eta = 1/L$ decreases the objective each step.
Solution. L-smoothness gives the descent lemma $f(\mathbf{y}) \le f(\mathbf{x}) + \nabla f(\mathbf{x})^\top(\mathbf{y} - \mathbf{x}) + \frac{L}{2}\|\mathbf{y} - \mathbf{x}\|^2$. Set $\mathbf{y} = \mathbf{x} - \eta\nabla f(\mathbf{x})$ (the gradient step) with $\eta = 1/L$:

$$f(\mathbf{y}) \le f(\mathbf{x}) - \eta\|\nabla f(\mathbf{x})\|^2 + \tfrac{L}{2}\eta^2\|\nabla f(\mathbf{x})\|^2 = f(\mathbf{x}) - \tfrac{1}{2L}\|\nabla f(\mathbf{x})\|^2 .$$

So the objective decreases by at least $\frac{1}{2L}\|\nabla f\|^2 \ge 0$ each step, with equality only when the gradient is zero (a stationary point). This guarantees monotone progress and is the foundation of the convergence-rate proofs. The condition $\eta \le 1/L$ is exactly the requirement that the step not overshoot the quadratic upper bound — larger steps can increase f (the divergence seen in the worked trace with $\eta > 2/L$). It connects the smoothness constant L (the largest Hessian eigenvalue) directly to the safe step size.

Worked Example

Exercise. Explain, using the KKT conditions, why the lasso produces exactly-zero coefficients but ridge does not.
Solution. The lasso objective $\|\mathbf{y} - \mathbf{X}\boldsymbol{\beta}\|^2 + \lambda\|\boldsymbol{\beta}\|_1$ has a non-differentiable penalty at $\beta_j = 0$. The optimality (KKT/subgradient) condition for coordinate j is that 0 lies in the subdifferential: the data-fit gradient must be balanced by a subgradient of $\lambda|\beta_j|$, which is $\lambda\,\mathrm{sign}(\beta_j)$ for $\beta_j \neq 0$ but the entire interval $[-\lambda, \lambda]$ at $\beta_j = 0$. So whenever the data-fit gradient's magnitude is $\le \lambda$, the coordinate can satisfy optimality at exactly $\beta_j = 0$ — the soft-thresholding that zeros weak coefficients. Ridge's penalty $\lambda\beta_j^2$ has derivative $2\lambda\beta_j$, which is zero only at $\beta_j = 0$ and provides no "dead zone" — the optimum balances forces at a small but nonzero value. Geometrically, the ℓ_1 ball has corners on the axes (where coordinates are zero) that the solution tends to hit, while the ℓ_2 ball is round with no such corners. This is the optimization-theoretic root of lasso sparsity.

20 Quasi-Newton Methods: Curvature Without the Hessian

Newton's method is fast but needs the $O(p^3)$ Hessian inverse. Quasi-Newton methods build an approximation to $\mathbf{H}^{-1}$ from gradient information alone.

20.1 The secant idea

Over one step, the change in gradient relates to the change in position through the Hessian: $\nabla f(\mathbf{x}_{k+1}) - \nabla f(\mathbf{x}_k) \approx \mathbf{H}(\mathbf{x}_{k+1} - \mathbf{x}_k)$ (the multivariate mean-value relation). Writing $\mathbf{s}_k = \mathbf{x}_{k+1} - \mathbf{x}_k$ and $\mathbf{y}_k = \nabla f(\mathbf{x}_{k+1}) - \nabla f(\mathbf{x}_k)$, this is the **secant equation** $\mathbf{y}_k \approx \mathbf{H}\mathbf{s}_k$. **BFGS** maintains an approximation $\mathbf{B}_k \approx \mathbf{H}^{-1}$ that satisfies this equation, updating it by a rank-two correction each step using only $\mathbf{s}_k$ and $\mathbf{y}_k$ — gradients and positions it already has. No second derivatives are ever computed; the curvature is *inferred* from how the gradient changed.

20.2 Why L-BFGS scales

Full BFGS stores the $p \times p$ matrix $\mathbf{B}_k$ — infeasible for large p. **L-BFGS** (limited-memory BFGS) stores only the last m pairs $(\mathbf{s}_k, \mathbf{y}_k)$ (typically $m = 5$–20) and reconstructs the action of $\mathbf{B}_k$ on a vector by a two-loop recursion, never forming the matrix. The cost per step is $O(mp)$ — linear in the parameter count — making L-BFGS the method of choice for smooth, medium-scale problems (logistic regression, CRFs, many scientific models) where it converges in far fewer iterations than gradient descent while staying memory-light. It sits between first-order methods (cheap steps, many of them) and full Newton (expensive steps, few of them).

Intuition

Quasi-Newton methods embody a beautiful idea: you can learn the curvature from the trail of gradients without ever computing a second derivative. The secant equation says "the Hessian is what maps your position-change to your gradient-change," and BFGS fits an approximation respecting that. L-BFGS makes it scalable by remembering only a short history. For a quant fitting smooth convex models (GLMs, maximum-likelihood estimators), L-BFGS often dominates both plain gradient descent (too slow) and Newton (too expensive) — and it is the default optimizer behind many `scikit-learn` and statistical-modeling routines. It is the practical sweet spot on the first-order-to-second-order spectrum.

21 Proximal and Coordinate Methods for Non-Smooth Problems

The lasso's ℓ_1 penalty is non-differentiable, so plain gradient descent does not apply. Two families handle non-smooth regularizers.

21.1 Proximal gradient descent

Split the objective into a smooth loss f plus a non-smooth penalty g (e.g. $\lambda \|\boldsymbol{\beta}\|_1$). **Proximal gradient descent** takes a gradient step on f, then applies the **proximal operator** of g — for the ℓ_1 penalty, this is exactly soft-thresholding $S_\lambda(z) = \text{sign}(z)\max(|z| - \lambda, 0)$. So each iteration is: gradient step on the loss, then shrink-and-threshold. This is ISTA (iterative soft-thresholding), and its accelerated version FISTA adds momentum for the $\sqrt{\kappa}$ speedup. The proximal operator generalizes the gradient step to non-smooth functions, and for the lasso it is the source of sparsity — the threshold zeros small coefficients exactly, as the regression note's coordinate-descent treatment showed.

21.2 Coordinate descent

Alternatively, minimize one coordinate at a time, cycling through them. For the lasso, each coordinate update is a scalar soft-threshold (a closed form), so coordinate descent is extremely fast and exploits sparsity (zero coordinates stay cheap to check). This is the algorithm behind `glmnet` and `scikit-learn`'s lasso path. It works because the ℓ_1 penalty is *separable* across coordinates (a sum of per-coordinate terms), so optimizing one at a time is well-behaved — a property that fails for non-separable penalties, where coordinate descent can stall.

Intuition

Non-smooth optimization is essential to modern ML because the most useful regularizers — ℓ_1 for sparsity, nuclear norm for low rank, total variation for smoothness — are non-differentiable by design (their kinks are what produce the desired structure). Proximal methods and coordinate descent handle them by replacing the gradient step with a proximal/soft-threshold step that respects the kink. The recurring character is that the non-differentiable point (e.g. $\beta_j = 0$ for ℓ_1) is exactly where the solution *wants* to sit, so the optimizer must be built to land there rather than smooth it away. This is the optimization counterpart of the regression note's observation that the ℓ_1 ball's corners produce sparsity.

22 Differentiating Through Expectations

Many ML objectives are expectations — expected reward in reinforcement learning, the ELBO in variational inference, expected loss under data augmentation. Differentiating them needs special care.

22.1 The score-function (REINFORCE) estimator

To differentiate $\mathbb{E}_{p_\theta(z)}[f(z)]$ with respect to the distribution's parameters θ, the **score-function trick** uses $\nabla_\theta \mathbb{E}[f] = \mathbb{E}[f(z)\nabla_\theta \log p_\theta(z)]$. The identity follows from $\nabla_\theta p_\theta = p_\theta \nabla_\theta \log p_\theta$ (the "log-derivative trick"). It works for *any* distribution, even non-differentiable f, but has high variance (hence baselines and variance-reduction tricks in policy-gradient RL).

22.2 The reparameterization trick

When z can be written as a differentiable transform of parameter-free noise — e.g. $z = \mu_\theta + \sigma_\theta \epsilon$ with $\epsilon \sim \mathcal{N}(0,1)$ — then $\nabla_\theta \mathbb{E}[f(z)] = \mathbb{E}_\epsilon[\nabla_\theta f(\mu_\theta + \sigma_\theta \epsilon)]$, pushing the gradient inside through the deterministic transform. This has much lower variance than the score function and is what makes variational autoencoders (the unsupervised note's ELBO) trainable by ordinary backpropagation. The two estimators are the standard tools for stochastic objectives, trading generality (score function works always) against variance (reparameterization is lower-variance when applicable).

22.3 Jensen's inequality and the ELBO

For a concave function (like log), **Jensen's inequality** gives $\log \mathbb{E}[X] \geq \mathbb{E}[\log X]$. This single inequality generates the **evidence lower bound**: the intractable log-marginal-likelihood $\log p(\mathbf{x}) = \log \mathbb{E}_q[p(\mathbf{x}, z)/q(z)] \geq \mathbb{E}_q[\log(p(\mathbf{x}, z)/q(z))]$, the ELBO that EM and variational inference maximize (the unsupervised note). The gap in Jensen's inequality is exactly the KL divergence between q and the true posterior — so tightening the bound (the E-step) means matching q to the posterior. Jensen's inequality is thus the mathematical seed of the entire variational-inference framework.

Intuition

Differentiating through randomness is what connects optimization to probabilistic ML. The score-function estimator (general, higher-variance) powers policy-gradient reinforcement learning; the reparameterization trick (lower-variance, needs a differentiable sampling path) powers variational autoencoders; and Jensen's inequality generates the ELBO that both EM and variational inference climb. The unifying theme is that an expectation over a parameterized distribution can be differentiated, by one route or another, so that gradient-based learning extends from deterministic losses to stochastic objectives. This is the bridge from the deterministic optimization of this note to the latent-variable and reinforcement-learning models that define much of modern ML.

23 Further Worked Exercises

Worked Example

Exercise. Show that the sum of a convex function and a strongly convex function is strongly convex, and explain why this matters for regularized learning.
Solution. A function is μ-strongly convex if $f(\mathbf{x}) - \frac{\mu}{2}\|\mathbf{x}\|^2$ is convex (equivalently, its Hessian satisfies $\mathbf{H} \succeq \mu\mathbf{I}$). Let f be convex ($\mathbf{H}_f \succeq 0$) and g be μ-strongly convex ($\mathbf{H}_g \succeq \mu\mathbf{I}$). Then the Hessian of $f+g$ is $\mathbf{H}_f + \mathbf{H}_g \succeq 0 + \mu\mathbf{I} = \mu\mathbf{I}$, so $f+g$ is μ-strongly convex. Application: an arbitrary convex loss (e.g. a possibly-flat logistic loss under collinearity) plus the strongly convex ridge penalty $\lambda\|\boldsymbol{\beta}\|^2$ (which is 2λ-strongly convex) yields a strongly convex objective. Strong convexity guarantees a *unique* minimum and a fast (linear, rate $1-\mu/L$) convergence for gradient descent. So ridge regularization not only controls variance statistically but also makes the optimization well-posed and fast — the same $\lambda\mathbf{I}$ that lifts the Hessian's eigenvalues away from zero, connecting to the linear-algebra note's conditioning.

Worked Example

Exercise. A practitioner sets the SGD learning rate too high and the loss diverges; too low and it barely moves. Explain both using the descent lemma.
Solution. The descent lemma bounds one step's change as $f(\mathbf{y}) \le f(\mathbf{x}) - \eta(1 - \frac{L\eta}{2})\|\nabla f\|^2$ (for the full-gradient step). The factor $(1 - \frac{L\eta}{2})$ is positive (guaranteeing decrease) only when $\eta < 2/L$; for $\eta > 2/L$ it is negative, so the bound permits — and in practice produces — an *increase*, hence divergence (the steep direction overshoots, as in the worked zig-zag trace). For very small η, the decrease per step $\approx \eta\|\nabla f\|^2$ is tiny, so progress is glacial and many steps are needed. The sweet spot is just below $2/L$ (often $\eta \approx 1/L$), large enough for fast progress, small enough for stability. With SGD, gradient noise tightens the requirement further and motivates a decaying schedule. This is why the learning rate is the first hyperparameter to tune and why learning-rate finders (sweeping η and watching the loss) are standard practice.

24 A Worked Newton Step for Logistic Regression

We connect the optimization machinery to a real model by taking one Newton/IRLS step on a tiny logistic regression, computing every number.

24.1 Setup

Two data points, one feature plus intercept. Point A: $x = 1$, $y = 1$. Point B: $x = -1$, $y = 0$. Start at $\boldsymbol{\beta} = (\beta_0, \beta_1) = (0, 0)$, so the design rows are $(1, 1)$ and $(1, -1)$.

24.2 Forward quantities

At $\boldsymbol{\beta} = (0, 0)$, the linear predictors are $\eta_A = 0$ and $\eta_B = 0$, so both predicted probabilities are $p = \sigma(0) = 0.5$. The gradient of the negative log-likelihood is $\mathbf{g} = \mathbf{X}^\top(\mathbf{p} - \mathbf{y})$:

$$\mathbf{p} - \mathbf{y} = (0.5 - 1,\, 0.5 - 0) = (-0.5,\, 0.5), \quad \mathbf{g} = \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}^\top \begin{pmatrix} -0.5 \\ 0.5 \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix} \begin{pmatrix} -0.5 \\ 0.5 \end{pmatrix} = \begin{pmatrix} 0 \\ -1 \end{pmatrix}.$$

(The first row of $\mathbf{X}^\top$ is $(1, 1)$, the second $(1, -1)$.) So $\mathbf{g} = (0, -1)$: the intercept gradient is zero (the two points are balanced), and the slope gradient is -1 (the data want a positive slope, since larger x goes with $y = 1$).

24.3 The Hessian and the step

The IRLS weights are $w_i = p_i(1 - p_i) = 0.5 \cdot 0.5 = 0.25$ for both points, so $\mathbf{W} = 0.25\mathbf{I}$. The Hessian is $\mathbf{H} = \mathbf{X}^\top\mathbf{W}\mathbf{X} = 0.25\,\mathbf{X}^\top\mathbf{X}$:

$$\mathbf{X}^\top\mathbf{X} = \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix} = \begin{pmatrix} 2 & 0 \\ 0 & 2 \end{pmatrix}, \quad \mathbf{H} = 0.25 \begin{pmatrix} 2 & 0 \\ 0 & 2 \end{pmatrix} = \begin{pmatrix} 0.5 & 0 \\ 0 & 0.5 \end{pmatrix}.$$

The Newton step is $\boldsymbol{\beta} \leftarrow \boldsymbol{\beta} - \mathbf{H}^{-1}\mathbf{g} = (0, 0) - \left(\begin{smallmatrix} 2 & 0 \\ 0 & 2 \end{smallmatrix}\right)\left(\begin{smallmatrix} 0 \\ -1 \end{smallmatrix}\right) = (0, 0) - (0, -2) = (0, 2)$. So after one step $\boldsymbol{\beta} = (0, 2)$: zero intercept (correct, by symmetry), slope 2. Recomputing, the new probabilities are $\sigma(2) = 0.88$ for point A and $\sigma(-2) = 0.12$ for point B — both moved decisively toward their labels in a single step, the quadratic convergence of Newton's method on the convex logistic loss.

Intuition

This worked step ties the optimization note to the logistic-regression note with explicit numbers. The IRLS weights $p(1 - p)$ are the Bernoulli variances and the Hessian curvature; the Newton step solves the local quadratic exactly; and because the logistic loss is convex (proven earlier), the iteration heads to the unique global optimum. The same computation, vectorized over thousands of points and features, is exactly what a logistic-regression solver runs — and seeing it once by hand makes the abstract "Fisher scoring" and "IRLS" concrete. It also shows why the method is fast: one well-chosen second-order step moved the fit most of the way, where gradient descent would have inched.

24.4 A worked momentum comparison

On the ill-conditioned quadratic $f = \frac{1}{2}(10x_1^2 + x_2^2)$ from earlier, recall plain gradient descent zig-zagged. With momentum ($\beta = 0.9$, small η), the velocity in the oscillating x_1 direction accumulates alternating-sign gradients that partly cancel, damping the oscillation, while the consistent x_2 gradient accumulates into a larger effective step — so x_2 descends faster and x_1 stops overshooting. Quantitatively, momentum changes the iteration count from $O(\kappa) = O(10)$ to $O(\sqrt{\kappa}) = O(\sqrt{10}) \approx 3.2$ times the well-conditioned count — a roughly $3\times$ speedup here, more for stiffer problems. This is the acceleration that makes momentum standard in deep learning.

25 Final Worked Exercises

Worked Example

Exercise. Why is the negative log-likelihood, not the likelihood, minimized — and why does this turn products into sums?

Solution. For i.i.d. data the likelihood is a *product* $\prod_i p(y_i \mid \mathbf{x}_i; \boldsymbol{\theta})$. Maximizing a product is awkward: it underflows numerically (many small numbers multiplied) and its derivative (by the product rule) is messy. Taking the logarithm — a monotone transform, so the maximizer is unchanged — converts the product to a *sum* $\sum_i \log p(y_i \mid \mathbf{x}_i; \boldsymbol{\theta})$, whose derivative is a sum of per-example gradients (the basis of SGD's unbiased minibatch gradient). Negating turns maximization into minimization, matching the optimization conventions of this note. So "minimize the negative log-likelihood" is "maximize the likelihood" made numerically stable and differentiation-friendly, and the sum structure is exactly what lets stochastic gradient methods sample a subset of terms. This is why cross-entropy (the negative log-likelihood for classification) is the universal training loss.

Worked Example

Exercise. Explain why gradient descent on a non-convex neural-network loss still works in practice, despite no global-optimum guarantee.
Solution. Several empirical and theoretical reasons. (1) In high dimensions, most critical points are *saddles*, not poor local minima (an indefinite Hessian is overwhelmingly likely when there are many eigenvalues), and SGD's noise helps escape saddles. (2) Over-parameterized networks have many global minima forming connected low-loss regions, so reaching *a* good minimum is easier than the worst-case picture suggests. (3) The implicit regularization of SGD biases toward flat, well-generalizing minima. (4) Architecture choices (ReLU, residual connections, normalization) keep the chain-rule gradient factors near 1, avoiding vanishing/exploding gradients. So while convexity's clean guarantee is gone, the landscape of large networks is empirically benign enough that first-order methods with good initialization, momentum, and normalization reliably find useful solutions. This is the central reason deep learning is an empirical discipline: the optimization works, but for subtle reasons rather than the clean convex guarantee that governs linear models and SVMs.

26 Consolidating Exercises: Optimization in Practice

Worked Example

Exercise. You observe that training loss decreases smoothly but validation loss decreases then rises. Frame this as an optimization-vs-generalization distinction and state the standard response.
Solution. Optimization concerns minimizing the *training* objective; generalization concerns performance on *unseen* data. The smooth training-loss decrease shows optimization is working — the optimizer is successfully reducing the training objective. The validation-loss U-shape shows the model is beginning to fit training-set noise (overfitting): past the U's minimum, further optimization of the training loss *hurts* test performance. The standard response is **early stopping** — halt at the validation minimum — which acts as a regularizer by limiting how far the optimizer drives the parameters toward the training optimum (it is closely related to ℓ_2 regularization for linear models, bounding the effective parameter norm). This separates the two concerns cleanly: the optimizer should minimize training loss, but *how long you let it* is a generalization decision, made on held-out data. It is the optimization-side view of the bias–variance tradeoff.

Worked Example

Exercise. Derive the gradient of the ridge objective and show how λ rescales the solution's eigencomponents.

Solution. The ridge objective is $\|\mathbf{y} - \mathbf{X}\boldsymbol{\beta}\|^2 + \lambda \|\boldsymbol{\beta}\|^2$, with gradient $-2\mathbf{X}^\top(\mathbf{y} - \mathbf{X}\boldsymbol{\beta}) + 2\lambda\boldsymbol{\beta}$. Setting to zero: $(\mathbf{X}^\top\mathbf{X} + \lambda\mathbf{I})\boldsymbol{\beta} = \mathbf{X}^\top\mathbf{y}$, so $\hat{\boldsymbol{\beta}} = (\mathbf{X}^\top\mathbf{X} + \lambda\mathbf{I})^{-1}\mathbf{X}^\top\mathbf{y}$. Using the SVD $\mathbf{X} = \mathbf{U}\boldsymbol{\Sigma}\mathbf{V}^\top$, the solution decomposes along the right-singular directions $\mathbf{v}_i$ with coefficients scaled by the *filter factor* $\sigma_i^2/(\sigma_i^2 + \lambda)$ (versus OLS's $1/\sigma_i$). For large σ_i (high-variance, well-determined directions) the factor is near 1 (barely shrunk); for small σ_i (low-variance, noise-prone directions) it is near 0 (heavily shrunk). So λ acts as a low-pass filter on the data's spectral directions, damping the noisy small-σ directions that destabilize OLS — the optimization producing exactly the spectral-shrinkage behavior the linear-algebra and regression notes describe. This unifies the gradient (optimization), the SVD (linear algebra), and the shrinkage (statistics) views of ridge.

Worked Example

Exercise. Why does the choice of optimizer rarely change the final model for a convex problem but can dramatically change it for a neural network?

Solution. For a convex problem, every local minimum is the global minimum, so *any* convergent optimizer (gradient descent, L-BFGS, Newton, coordinate descent) reaches the *same* unique solution — they differ only in speed, not destination. For a non-convex neural-network loss, there are many minima of differing generalization quality, and different optimizers (and even different random seeds) follow different trajectories into different basins. The optimizer's implicit bias — SGD toward flat minima, adaptive methods toward different regions — then affects *which* solution is found, and hence test performance. This is why optimizer choice and hyperparameters are a minor concern for linear models and GLMs (convex, one answer) but a significant tuning axis for deep learning (non-convex, many answers). It is the practical consequence of the convexity divide that organizes this entire note.

27 Worked Convergence: Counting Iterations

A final quantitative section makes the convergence rates concrete by computing iteration counts for representative problems.

27.1 Gradient descent on a conditioned problem

For an L-smooth, μ-strongly-convex objective with $\kappa = L/\mu$, gradient descent contracts the error by $(1 - 1/\kappa)$ per step. To reach relative error ϵ requires $k \approx \kappa \ln(1/\epsilon)$ steps. Worked values for $\epsilon = 10^{-4}$ (so $\ln(1/\epsilon) \approx 9.2$):

- $\kappa = 1$ (perfectly conditioned): ≈ 9 steps — essentially immediate.
- $\kappa = 10$: ≈ 92 steps.
- $\kappa = 100$: ≈ 921 steps.
- $\kappa = 10^4$ (ill-conditioned, e.g. unscaled features): $\approx 92{,}000$ steps — painfully slow.

The lesson is stark: conditioning multiplies the iteration count linearly, so a $100\times$ worse condition number means $100\times$ more iterations. This is why feature scaling (which can drop κ from thousands to single digits) is often the highest-leverage preprocessing step, and why batch normalization accelerates deep-network training so dramatically — both attack κ.

27.2 The momentum and Newton improvements

Momentum improves the rate to $(1 - 1/\sqrt{\kappa})$, so the iteration count becomes $\approx \sqrt{\kappa}\ln(1/\epsilon)$. For $\kappa = 100$ that is $\approx 10 \cdot 9.2 = 92$ steps instead of 921 — a $10\times$ speedup, growing with κ. **Newton's method**, by contrast, converges *quadratically* near the optimum: the number of correct digits roughly doubles each step, so reaching $\epsilon = 10^{-4}$ (about 13 bits) from a decent start takes only a

handful of steps, *independent of* κ (Newton rescales away the conditioning entirely). The trade is per-step cost: gradient descent is $O(np)$ per step, Newton $O(np^2 + p^3)$. For p in the millions, the p^3 kills Newton, so first-order methods with momentum and adaptive scaling dominate deep learning; for p in the hundreds or thousands with a smooth convex loss, L-BFGS or Newton win by needing far fewer steps.

Intuition

These iteration counts turn the abstract "linear" versus "quadratic" convergence into engineering reality. Gradient descent's cost scales with κ (so condition the problem, by scaling and normalization); momentum buys a $\sqrt{\kappa}$ improvement for free; Newton removes the κ dependence entirely but at cubic per-step cost. The optimizer you choose is dictated by the problem size p and the conditioning κ: big p forces first-order methods, where you fight κ with preconditioning, momentum, and adaptivity; small-to-medium p with a smooth convex loss invites quasi-Newton. Knowing the iteration-count formulas lets you predict, before running anything, whether a problem will train in seconds or days — and what to change if it is the latter.

28 Consolidated Exercises

1. **(Gradient)** For $f(\mathbf{x}) = \frac{1}{2}\mathbf{x}^\top \mathbf{A}\mathbf{x} - \mathbf{b}^\top \mathbf{x}$ with symmetric $\mathbf{A}$, find ∇f and the minimizer; relate to solving $\mathbf{A}\mathbf{x} = \mathbf{b}$.

2. **(Convexity)** Prove the log-sum-exp function $f(\mathbf{x}) = \log \sum_i e^{x_i}$ is convex by showing its Hessian is PSD. (This is the normalizer of the softmax.)

3. **(Step size)** For $f(x) = \frac{L}{2}x^2$, show gradient descent diverges for $\eta > 2/L$ and find the η giving fastest convergence.

4. **(Conditioning)** For a quadratic with Hessian eigenvalues $\{1, \kappa\}$, derive the number of gradient-descent steps to reach accuracy ϵ as a function of κ.

5. **(KKT)** Solve $\min x^2 + y^2$ subject to $x + y \geq 1$ using KKT; identify whether the constraint is active and interpret the multiplier.

6. **(Duality)** Derive the dual of $\min \frac{1}{2}\|\mathbf{x}\|^2$ subject to $\mathbf{A}\mathbf{x} = \mathbf{b}$ and verify strong duality.

7. **(Newton)** Apply one Newton step to $f(x) = x^4$ from $x_0 = 1$; contrast with a gradient step and comment on the behavior of Newton when the Hessian degenerates at the optimum.

8. **(Expectation gradient)** Derive the reparameterization gradient for $\mathbb{E}_{\mathcal{N}(\mu,\sigma^2)}[f(x)]$ by writing $x = \mu + \sigma\epsilon$, $\epsilon \sim \mathcal{N}(0,1)$, and contrast its variance with the score-function estimator.

End of the Calculus & Optimization prerequisite. The companion documents cover Linear Algebra and Probability & Statistics; together they ground the algorithm series.