Mathematical Foundations of Data Sciences

Gabriel Peyré
CNRS & DMA, École Normale Supérieure

CHAPTER 15

Optimization & Machine Learning: Advanced Topics

September 9, 2026

Reliable and efficient learning requires control of model complexity, economical use of large datasets, and accurate derivative calculations. We study regularization through ridge regression and sparsity, followed by stochastic gradient methods that reduce the cost of individual updates. The final section derives automatic differentiation on computational graphs and applies it to feedforward networks, recurrent models, and optimization procedures.

15.1 Regularization

When the sample size n is small relative to the model dimension p, regularization helps control the complexity of empirical risk minimization.

15.1.1 Penalized Least Squares

For simplicity, consider regularized least-squares regression:

$$\min_{x\in\mathbb{R}^p} f_\lambda(x) \overset{\text{def.}}{=} \frac{1}{2}\|Ax-y\|^2 + \lambda R(x) \tag{15.1}$$

where $R(x)$ is the regularizer and $\lambda \geqslant 0$ the regularization parameter. The regularizer expresses prior structure of the weight vector x, such as small magnitude or sparsity. The strength λ can be selected by cross-validation.

We assume that R is finite, nonnegative, lower semicontinuous, and coercive, i.e. $R(x) \to +\infty$ as $\|x\| \to +\infty$. The following proposition shows that as $\lambda \downarrow 0$, regularization selects among the exact solutions when $Ax = y$ is feasible. This is particularly useful when $\ker(A) \neq \{0\}$ and there are infinitely many exact solutions.

Proposition 15.1. *Assume $Ax = y$ is feasible and $\lambda_k > 0$ tends to zero. If x_{λ_k} minimizes f_{λ_k}, then this sequence is bounded, and every accumulation point $x^\star$ is a solution of the constrained optimization problem*

$$\min_{Ax=y} R(x). \tag{15.2}$$

Proof. Choose an arbitrary feasible point x_0, so $Ax_0 = y$. Optimality of x_{λ_k} gives

$$\frac{1}{2}\|Ax_{\lambda_k}-y\|^2 + \lambda_k R(x_{\lambda_k}) \leqslant \lambda_k R(x_0). \tag{15.3}$$

Nonnegativity of the terms implies $R(x_{\lambda_k}) \leqslant R(x_0)$. Coercivity of the regularizer makes the sequence $(x_{\lambda_k})_k$ bounded, since R has bounded sublevel sets. The residual also satisfies $\|Ax_{\lambda_k}-y\|^2 \leqslant 2\lambda_k R(x_0)$, so every accumulation point obeys $Ax^\star = y$. Along a subsequence converging to $x^\star$, lower semicontinuity gives $R(x^\star) \leqslant R(x_0)$. Since x_0 was arbitrary, $x^\star$ solves (15.2). □

15.1.2 Ridge Regression

Ridge regression uses the quadratic penalty $R(x) = \frac{1}{2}\|x\|^2_{\mathbb{R}^p}$. For $\lambda > 0$, the objective in (15.1) has a unique minimizer:

$$x_\lambda \overset{\text{def.}}{=} \operatorname*{argmin}_{x\in\mathbb{R}^p} f_\lambda(x) = \frac{1}{2}\|Ax-y\|^2_{\mathbb{R}^n} + \frac{\lambda}{2}\|x\|^2_{\mathbb{R}^p}.$$

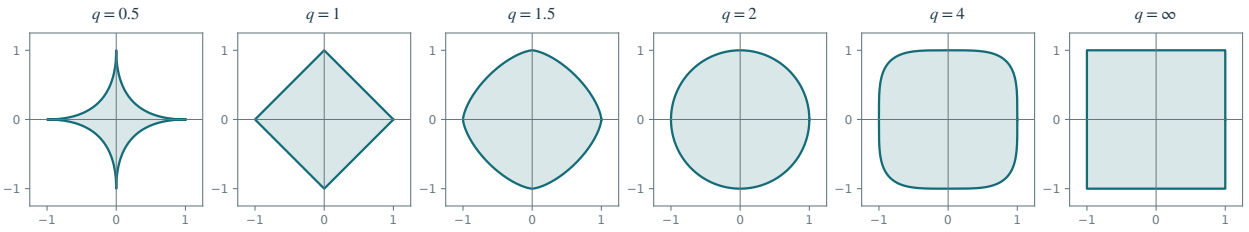

Figure 15.1. ℓ^q balls $\{x \,;\, \sum_k |x_k|^q \leqslant 1\}$ for varying q.

One has

$$\nabla f_\lambda(x) = A^\top(Ax - y) + \lambda x, \qquad \nabla f_\lambda(x_\lambda) = 0$$

Thus x_λ depends linearly on y and can be computed by solving a linear system. Two equivalent expressions are available.

Proposition 15.2. *One has*

$$x_\lambda = (A^\top A + \lambda \mathrm{Id}_p)^{-1} A^\top y, \tag{15.4}$$
$$= A^\top (AA^\top + \lambda \mathrm{Id}_n)^{-1} y. \tag{15.5}$$

Proof. Denoting $B \overset{\text{def.}}{=} (A^\top A + \lambda \mathrm{Id}_p)^{-1} A^\top$ and $C \overset{\text{def.}}{=} A^\top (AA^\top + \lambda \mathrm{Id}_n)^{-1}$, one has $(A^\top A + \lambda \mathrm{Id}_p)B = A^\top$ while

$$(A^\top A + \lambda \mathrm{Id}_p)C = (A^\top A + \lambda \mathrm{Id}_p)A^\top (AA^\top + \lambda \mathrm{Id}_n)^{-1} = A^\top (AA^\top + \lambda \mathrm{Id}_n)(AA^\top + \lambda \mathrm{Id}_n)^{-1} = A^\top.$$

Since $A^\top A + \lambda \mathrm{Id}_p$ is invertible, this gives the desired result. □

The systems can be solved directly, for example by Cholesky factorization, or iteratively, for example by conjugate gradients. The latter exploits the quadratic structure more effectively than ordinary gradient descent.

For $n > p$, use the smaller system (15.4); for $n < p$, use (15.5).

Pseudo-inverse. As $\lambda \downarrow 0$, x_λ converges to the minimum-norm least-squares solution $A^+ y$. If $Ax = y$ is feasible, this is, by (15.2)

$$\operatorname*{argmin}_{Ax=y} \|x\|.$$

If A has full column rank, so $\ker(A) = \{0\}$ and $n \geqslant p$, $A^\top A \in \mathbb{R}^{p \times p}$ is an invertible matrix, and $(A^\top A + \lambda \mathrm{Id}_p)^{-1} \to (A^\top A)^{-1}$, so that

$$x_0 = A^+ y \quad \text{where} \quad A^+ \overset{\text{def.}}{=} (A^\top A)^{-1} A^\top.$$

If A has full row rank, so $\ker(A^\top) = \{0\}$ and $n \leqslant p$, the alternative formula is

$$x_0 = A^+ y \quad \text{where} \quad A^+ \overset{\text{def.}}{=} A^\top (AA^\top)^{-1}.$$

When $n = p$ and A is invertible, both formulas reduce to $A^+ = A^{-1}$. For any rank, the singular value decomposition (SVD) gives the Moore–Penrose pseudoinverse by inverting the nonzero singular values and leaving the zero ones unchanged.

15.1.3 Lasso

The Lasso uses an ℓ^1 penalty:

$$R(x) = \|x\|_1 \overset{\text{def.}}{=} \sum_{k=1}^{p} |x_k|.$$

The penalty promotes sparsity in the solutions x_λ of

$$x_\lambda \in \operatorname*{argmin}_{x \in \mathbb{R}^p} f_\lambda(x) = \frac{1}{2}\|Ax - y\|_{\mathbb{R}^n}^2 + \lambda \|x\|_1$$

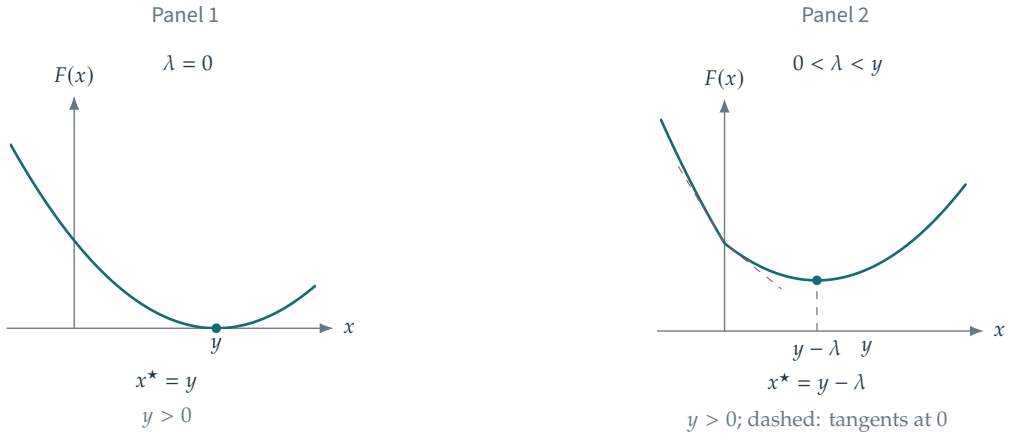

Figure 15.2. Effect of λ on the penalized scalar objective $F(x) \overset{\text{def.}}{=} \frac{1}{2}(x-y)^2 + \lambda|x|$.

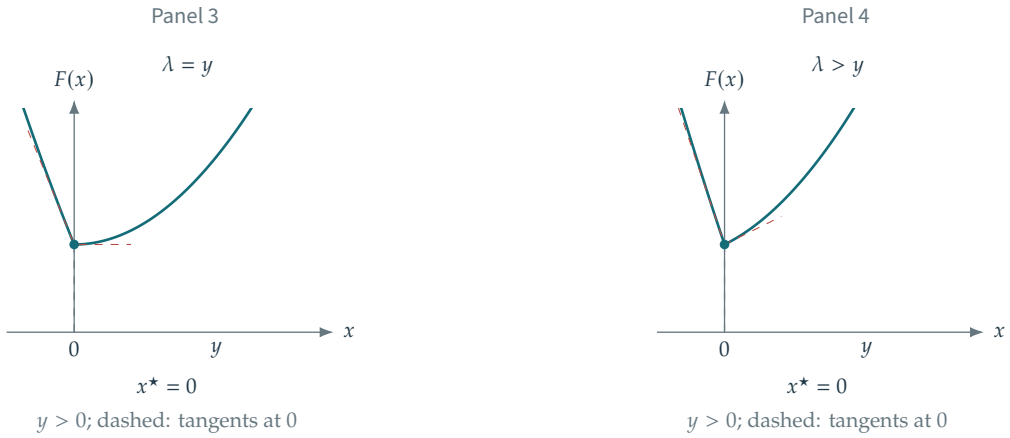

Figure 15.2 (continued). Effect of λ on the penalized scalar objective $F(x) \overset{\text{def.}}{=} \frac{1}{2}(x-y)^2 + \lambda|x|$.

The solutions x_λ may be nonunique, and typically have many zero coefficients. Figure 15.1 illustrates the geometry of ℓ^q balls. They concentrate around the coordinate axes as $q \to 0$, favoring sparse solutions, but become nonconvex for $q < 1$.

Sparsity can encode a prior on the unknown, as in imaging, or select a small subset of predictive features. Feature selection can make a model easier to interpret and cheaper to evaluate. Whether Lasso or ridge regression performs better depends on sparsity, correlations, noise, and the prediction task.

The objective f_λ is convex, but the ℓ^1 penalty is nonsmooth, so ordinary gradient descent does not apply at all points. Section 15.1.4 develops an appropriate modification of gradient descent. A closed-form solution x_λ is generally unavailable, but the case of an orthogonal design matrix A reduces to coordinatewise thresholding.

Proposition 15.3. *When $n = p$ and $A = \mathrm{Id}_n$, one has*

$$\operatorname*{argmin}_{x \in \mathbb{R}^p} \frac{1}{2}\|x-y\|^2 + \lambda\|x\|_1 = S_\lambda(y) \quad \textit{where} \quad S_\lambda(x) = (\operatorname{sign}(x_k)\max(|x_k| - \lambda, 0))_k$$

Proof. The separable form $f_\lambda(x) = \sum_k [\frac{1}{2}(x_k - y_k)^2 + \lambda|x_k|]$ reduces the problem to minimizing the scalar function $x \in \mathbb{R} \mapsto \frac{1}{2}(x-y)^2 + \lambda|x|$. Figure 15.2 illustrates this minimization graphically. First suppose $y \geqslant 0$. On $x < 0$, $F'(x) = x - y - \lambda < 0$, so no minimizer is negative. On $x > 0$, $F'(x) = x - y + \lambda$ vanishes at $x = y - \lambda$ when $y > \lambda$. Otherwise, the one-sided derivatives at zero have opposite weak signs, making zero the minimizer. The negative case follows by the symmetry $x \mapsto -x$. □

The nonlinear map S_λ is soft thresholding.

15.1.4 Iterative Soft Thresholding

We derive an iterative algorithm by minimizing a quadratic surrogate. We aim at minimizing

$$f(x) \overset{\text{def.}}{=} \frac{1}{2}\|y - Ax\|^2 + \lambda\|x\|_1$$

For a fixed reference point x', introduce the surrogate with parameter $\tau > 0$:

$$f_\tau(x, x') \overset{\text{def.}}{=} f(x) - \frac{1}{2}\|Ax - Ax'\|^2 + \frac{1}{2\tau}\|x - x'\|^2.$$

The surrogate touches the objective, $f_\tau(x, x) = f(x)$. Its additional quadratic term is

$$K(x, x') \overset{\text{def.}}{=} -\frac{1}{2}\|Ax - Ax'\|^2 + \frac{1}{2\tau}\|x - x'\|^2 = \frac{1}{2}\langle\left(\frac{1}{\tau}\mathrm{Id}_p - A^\top A\right)(x - x'),\, x - x'\rangle.$$

The term $K(x, x')$ is nonnegative when $\lambda_{\max}(A^\top A) \leqslant 1/\tau$, equivalently $\tau \leqslant 1/\|A\|_{\mathrm{op}}^2$, where $\|A\|_{\mathrm{op}} = \sigma_{\max}(A)$ is the operator norm. Thus $f_\tau(x, x')$ is a valid surrogate:

$$f(x) \leqslant f_\tau(x, x'), \quad f_\tau(x, x) = f(x), \quad \text{and} \quad f(\cdot) - f_\tau(\cdot, x') \text{ is smooth.}$$

The surrogate is strongly convex in its first argument, so its minimizer is unique. Define the iteration

$$x_{k+1} \overset{\text{def.}}{=} \operatorname*{argmin}_x f_\tau(x, x_k) \tag{15.6}$$

which ensures by construction that

$$f(x_{k+1}) \leqslant f(x_k).$$

Proposition 15.4. *The iterates x_k defined by (15.6) satisfy*

$$x_{k+1} = S_{\lambda\tau}\left(x_k - \tau A^\top(Ax_k - y)\right) \tag{15.7}$$

Here $S_\lambda(x) = (s_\lambda(x_m))_m$ applies the scalar soft-thresholding map $s_\lambda(r) = \mathrm{sign}(r)\max(|r| - \lambda, 0)$ coordinatewise.

Proof. One has

$$\begin{aligned} f_\tau(x, x') &= \frac{1}{2}\|Ax - y\|^2 - \frac{1}{2}\|Ax - Ax'\|^2 + \frac{1}{2\tau}\|x - x'\|^2 + \lambda\|x\|_1 \\ &= C + \frac{1}{2}\|Ax\|^2 - \frac{1}{2}\|Ax\|^2 + \frac{1}{2\tau}\|x\|^2 - \langle Ax,\, y\rangle + \langle Ax,\, Ax'\rangle - \frac{1}{\tau}\langle x,\, x'\rangle + \lambda\|x\|_1 \\ &= C + \frac{1}{2\tau}\|x\|^2 + \langle x,\, -A^\top y + A^\top Ax' - \frac{1}{\tau}x'\rangle + \lambda\|x\|_1 \\ &= C' + \frac{1}{\tau}\left(\frac{1}{2}\|x - (x' - \tau A^\top(Ax' - y))\|^2 + \tau\lambda\|x\|_1.\right) \end{aligned}$$

Proposition 15.3 shows that the minimizer of $f_\tau(x, x')$ is thus indeed $S_{\lambda\tau}(x' - \tau A^\top(Ax' - y))$ as claimed. □

Equation (15.7) defines iterative soft thresholding. For $A \neq 0$, the surrogate argument applies when $0 < \tau \leqslant 1/\|A\|_{\mathrm{op}}^2$. Convergence to a Lasso solution holds for a fixed step in the larger range $0 < \tau < 2/\|A\|_{\mathrm{op}}^2$. If $A = 0$, every positive step is admissible.

15.2 Stochastic Optimization

We study stochastic gradient methods for large datasets of size n and for objectives defined as expectations.

15.2.1 Minimizing Sums and Expectations

Many learning objectives are finite averages:

$$\min_{x \in \mathbb{R}^p} f(x) \overset{\text{def.}}{=} \frac{1}{n}\sum_{i=1}^{n} f_i(x) \tag{15.8}$$

or expectations:

$$\min_{x\in\mathbb{R}^p} f(x) \overset{\text{def.}}{=} \mathbb{E}_{\mathbf{z}\sim\pi}(f(x,\mathbf{z})) = \int_{\mathcal{Z}} f(x,z)\mathrm{d}\pi(z). \tag{15.9}$$

A finite average is an expectation under the uniform measure $\pi = \frac{1}{n}\sum_{i=1}^n \delta_i$, with $f(x,i) = f_i(x)$. Conversely, drawing $(z_i)_i$ i.i.d. from π and setting $f_i(x) = f(x,z_i)$ gives an empirical approximation of (15.9). Under integrability, the empirical objective converges pointwise by the law of large numbers. Convergence of minima or minimizers requires additional assumptions.

For example, empirical risk minimization with linear predictors uses

$$f_i(x) = \ell(\langle a_i,\, x\rangle, y_i) \quad \text{and} \quad f(x,z) = \ell(\langle a,\, x\rangle, y) \tag{15.10}$$

for $z = (a,y) \in \mathcal{Z} = (\mathcal{A} = \mathbb{R}^p) \times \mathcal{Y}$ (typically $\mathcal{Y} = \mathbb{R}$ or $\mathcal{Y} = \{-1,+1\}$ for regression and classification), where ℓ is some loss function. We illustrate the methods with binary logistic classification, using

$$L(s,y) \overset{\text{def.}}{=} \log(1+\exp(-sy)). \tag{15.11}$$

The same computational approach extends to general parametric models, including deep neural networks.

The stochastic methods below apply both to finite sums (15.8) and to expectations (15.9), under the corresponding unbiasedness and moment assumptions. We present them for finite sums. When n is large, the aim is to make each iteration inexpensive without evaluating every summand.

When the summands $f_i(x)$ are similar, a single gradient can approximate the average well: $\nabla f_i \approx \nabla f$. Evaluating ∇f_i is typically n times cheaper than forming the full gradient. Stochastic optimization is nevertheless not always faster than batch gradient descent. For moderate n, a deterministic method may use the available computation more effectively. When n is too large for a full pass to be practical, stochastic methods divide the work into small updates and begin improving the model immediately. Mini-batches also permit parallel computation.

15.2.2 Batch Gradient Descent (BGD)

Batch gradient descent (BGD), analyzed in Section 14.4, iterates

$$x_{k+1} = x_k - \tau_k \nabla f(x_k)$$

For a convex objective with an L-Lipschitz gradient and a minimizer, the condition $0 < \tau_{\min} \leqslant \tau_k \leqslant \tau_{\max} < 2/L$ ensures convergence. Strong convexity strengthens this to a linear rate.

For a finite-sum objective, the gradient is

$$\nabla f(x) = \frac{1}{n}\sum_{i=1}^n \nabla f_i(x) \tag{15.12}$$

Its cost is typically $O(np)$ when each ∇f_i can be evaluated in time linear in p.

For an ERM summand of the form (15.10), Taylor expansion gives

$$\begin{aligned} f_i(x+\varepsilon) &= \ell(\langle a_i,\, x\rangle + \langle a_i,\, \varepsilon\rangle, y_i) = \ell(\langle a_i,\, x\rangle, y_i) + \ell'(\langle a_i,\, x\rangle, y_i)\langle a_i,\, \varepsilon\rangle + o(\|\varepsilon\|) \\ &= f_i(x) + \langle \ell'(\langle a_i,\, x\rangle, y_i)a_i,\, \varepsilon\rangle + o(\|\varepsilon\|), \end{aligned}$$

Here $\ell'(s,y)$ denotes differentiation with respect to the score s, with the response y fixed. Consequently,

$$\nabla f_i(x) = \ell'(\langle a_i,\, x\rangle, y_i)a_i. \tag{15.13}$$

For the logistic loss, one has

$$L'(s,y) = -y\frac{e^{-sy}}{1+e^{-sy}}.$$

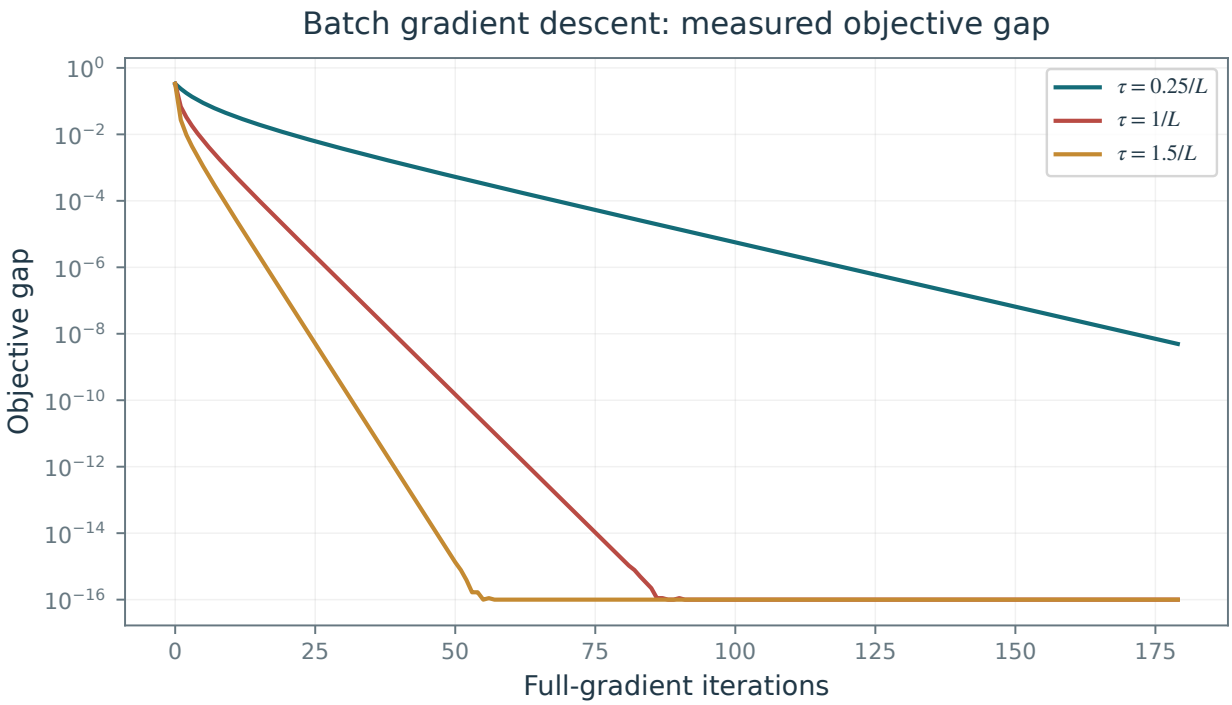

Figure 15.3. Objective-error evolution for batch gradient descent in logistic classification.

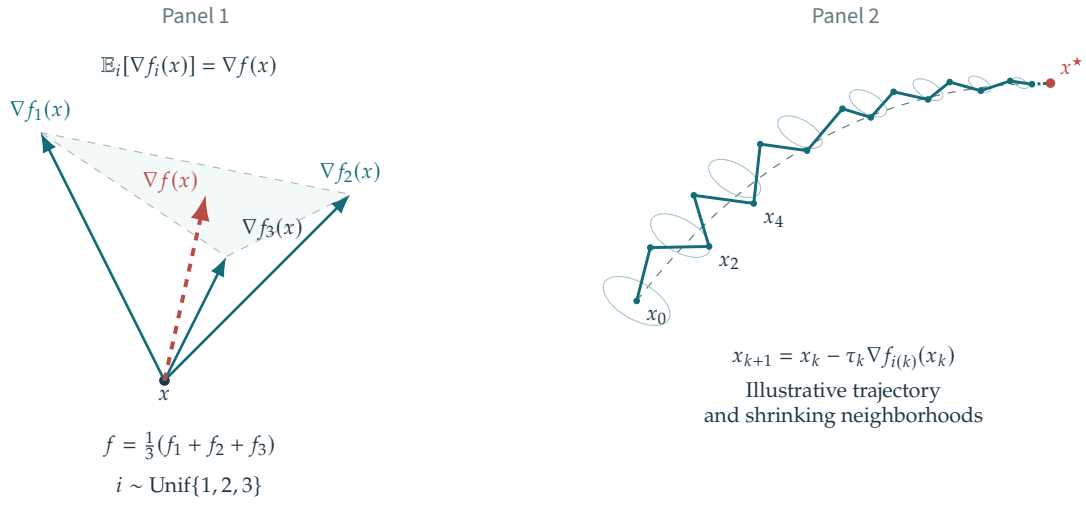

Figure 15.4. Unbiased stochastic gradient estimate (panel 1) and a schematic SGD trajectory (panel 2).

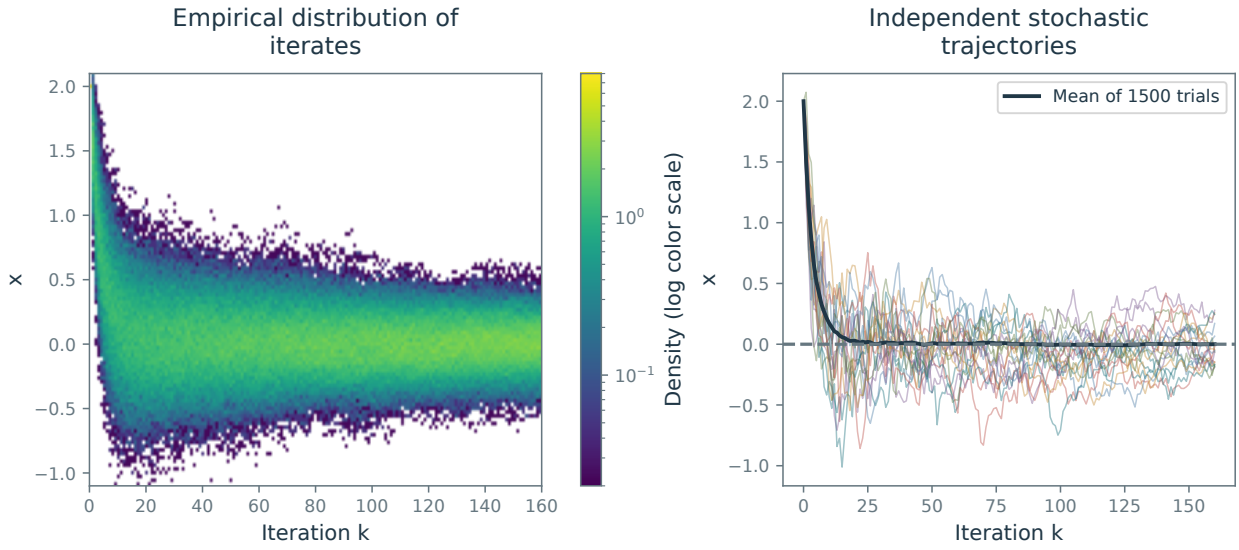

Figure 15.5. Repeated SGD runs for $k \mapsto x_k \in \mathbb{R}$. Left: the distribution of iterates at each iteration. Right: individual trajectories.

15.2.3 Stochastic Gradient Descent (SGD)

For very large n, computing the full gradient ∇f as in (15.12) is prohibitive. SGD replaces the full gradient by the gradient of one summand f_i, with i sampled uniformly. This gives an unbiased estimate:

$$\mathbb{E}_{\mathbf{i}} \nabla f_{\mathbf{i}}(x) = \nabla f(x) \tag{15.14}$$

where $\mathbf{i}$ is a random variable distributed uniformly in $\{1, \dots, n\}$.

Starting from x_0, stochastic gradient descent (SGD) iterates

$$x_{k+1} = x_k - \tau_k \nabla f_{i(k)}(x_k)$$

where, for each iteration index k, $i(k)$ is drawn independently of previous draws, uniformly from $\{1, \dots, n\}$. The iterates x_{k+1} are random. We therefore study convergence to a minimizer of f in a probabilistic sense, such as mean square or probability, together with the corresponding rates.

A batch-gradient step costs $O(np)$, whereas an SGD step costs $O(p)$. This makes SGD attractive when n is large and repeated passes through the data are costly. Redundancy between observations can sometimes yield accurate predictions after only $k \ll n$ updates, before all observations have been used.

The step-size schedule τ_k must balance two requirements: it usually needs to decrease to suppress stochastic noise, but not so quickly that the method stops making progress before reaching a minimizer.

A common schedule has asymptotic behavior $\tau_k \sim k^{-1}$ as $k \to +\infty$. One such schedule is

$$\tau_k \overset{\text{def.}}{=} \frac{\tau_0}{1 + k/k_0} \tag{15.15}$$

where $\tau_0 > 0$ sets the initial step and $k_0 > 0$ controls how long the steps remain approximately constant.

Figure 15.5 illustrates minimization of $f_1(x) + f_2(x)$ for $x \in \mathbb{R}$, with $f_1(x) = (x-1)^2$ and $f_2(x) = (x+1)^2$. The distribution of x_k concentrates around the minimizer $x^\star = 0$. The initial iterate x_0 is uniform on $[-1/2, 1/2]$.

The following theorem gives a mean-square convergence rate. The second-moment assumption is imposed along the iterates, not uniformly on all of $\mathbb{R}^p$: a globally strongly convex function cannot have a globally bounded gradient.

Theorem 15.5. *Let f be differentiable and μ-strongly convex, with minimizer $x^\star$. Let $\mathcal{F}_k$ contain the initial point and all randomness before the kth update, so x_k is $\mathcal{F}_k$-measurable. Assume the stochastic gradient G_k satisfies*

$$\mathbb{E}(G_k \mid \mathcal{F}_k) = \nabla f(x_k), \qquad \mathbb{E}(\|G_k\|^2 \mid \mathcal{F}_k) \leqslant C^2.$$

For $x_{k+1} = x_k - \tau_k G_k$, $\tau_k = 1/(\mu(k+2))$, and deterministic x_0,

$$\mathbb{E}\|x_k - x^\star\|^2 \leqslant \frac{R}{k+2}, \qquad R = \max\{2\|x_0 - x^\star\|^2, C^2/\mu^2\}. \tag{15.16}$$

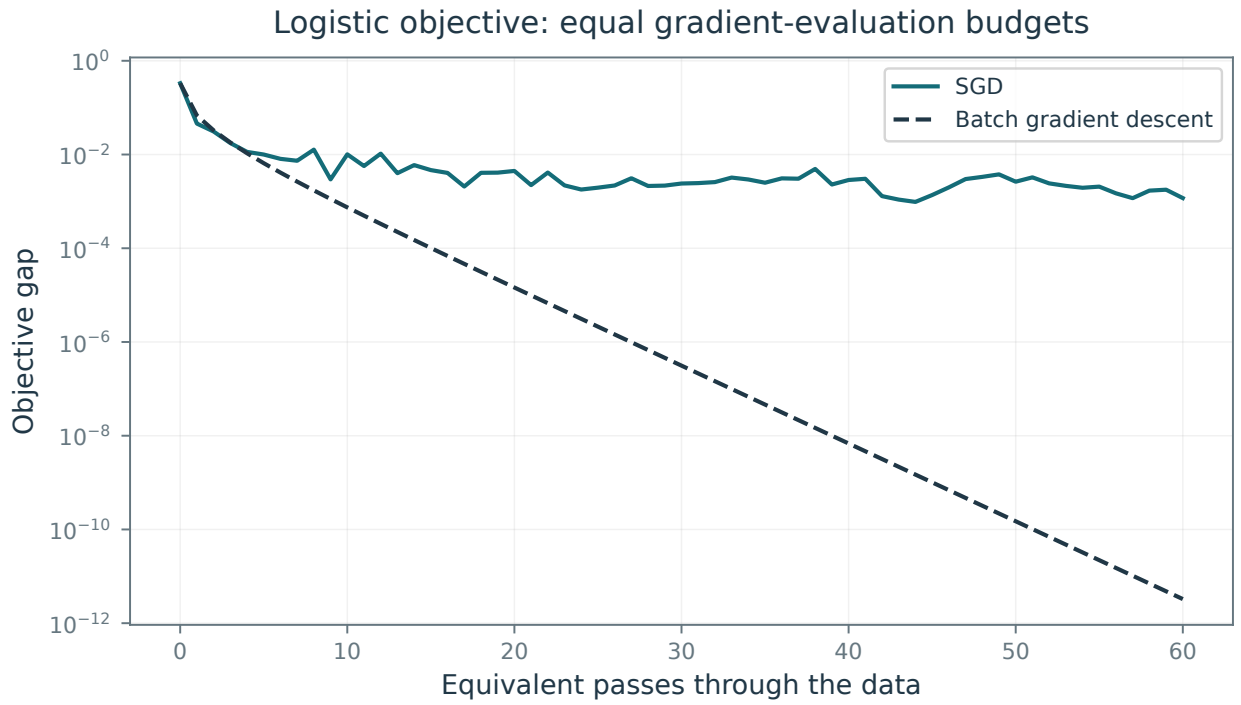

Figure 15.6. Objective-error evolution for SGD in logistic classification; the dashed curve shows batch gradient descent.

If f additionally has an L-Lipschitz gradient, then $\mathbb{E}(f(x_k) - f(x^\star)) \leqslant LR/(2(k+2))$.

Proof. Strong convexity and $\nabla f(x^\star) = 0$ imply

$$\langle \nabla f(x_k),\, x_k - x^\star \rangle \geqslant \mu \|x_k - x^\star\|^2. \tag{15.17}$$

Expand the squared update and take a conditional expectation, then average over the history. With $e_k = \mathbb{E}\|x_k - x^\star\|^2$,

$$e_{k+1} \leqslant (1 - 2\mu\tau_k) e_k + \tau_k^2 C^2. \tag{15.18}$$

The claim holds at $k = 0$ by the definition of R. Set $m = k + 2 \geqslant 2$. Since $1 - 2/m \geqslant 0$, the induction hypothesis gives

$$e_{k+1} \leqslant \left(1 - \frac{2}{m}\right) \frac{R}{m} + \frac{R}{m^2} = \frac{R(m-1)}{m^2} \leqslant \frac{R}{m+1}.$$

The objective bound follows from L-smoothness at the minimizer. □

The preceding SGD bound, and the averaging scheme below, give sublinear rates even when f is strongly convex. By contrast, batch gradient descent has a linear rate under the assumptions of Theorem 14.16.

Figure 15.6 compares the objective values $f(x_k)$. The black dashed curve shows batch gradient descent, with the iteration axis rescaled to reflect the n-fold cost of a batch update.

15.2.4 Stochastic Gradient Descent with Averaging (SGA)

Diminishing steps can slow SGD as τ_k approaches zero. Averaging reduces fluctuations in the estimate. Run SGD on auxiliary iterates $(\tilde{x}_k)_k$,

$$\tilde{x}_{k+1} = \tilde{x}_k - \tau_k \nabla f_{i(k)}(\tilde{x}_k)$$

and return their Cesàro average:

$$x_k \overset{\text{def.}}{=} \frac{1}{k} \sum_{\ell=1}^{k} \tilde{x}_\ell.$$

This gives stochastic gradient descent with averaging (SGA), also called averaged SGD.

The average can be maintained without storing the full trajectory, using the recursive update

$$x_{k+1} = \frac{1}{k+1}\tilde{x}_{k+1} + \frac{k}{k+1}x_k.$$

A typical step-size schedule is

$$\tau_k \overset{\text{def.}}{=} \frac{\tau_0}{1+\sqrt{k/k_0}}.$$

These steps decrease more slowly, at rate $k^{-1/2}$.

Averaging can reduce sensitivity to the tuning parameters (k_0, τ_0), although their choice still affects performance.

Bach proves that for logistic classification, averaging can adapt to local curvature under suitable regularity and step-size assumptions. This is a problem-dependent result, rather than a universal improvement of every constant over SGD.

15.2.5 Stochastic Averaged Gradient Descent (SAG)

For a finite dataset with n samples of total size $n \times p$, one can trade additional memory for reduced gradient noise by storing past gradient evaluations. This yields the stochastic averaged gradient (SAG) method.

Store the latest gradient of each summand in a table $(G^i)_{i=1}^n$, requiring $O(np)$ memory. Maintain its average g as an approximation to the full gradient.

Initialize $G^i = 0$ for all i and $g = 0$, so that $g = n^{-1}\sum_i G^i$ is the stored average. At each iteration, update the table and then x:

$$x_{k+1} = x_k - \tau g \quad \text{where} \quad \begin{cases} h \leftarrow \nabla f_{i(k)}(x_k), \\ g \leftarrow g + \frac{1}{n}(h - G^{i(k)}), \\ G^{i(k)} \leftarrow h. \end{cases}$$

Because most entries were evaluated at earlier iterates, the table average is generally biased conditional on the current history. Its convergence analysis therefore differs from the unbiased-SGD proof above. SAG uses a fixed step size τ. As for batch gradient descent, choose τ on the scale of $1/L$, where L bounds the Lipschitz constants of the individual gradients ∇f_i; the admissible numerical constant depends on the convergence theorem.

For smooth convex finite sums with a minimizer and a suitable step size, SAG admits an $O(1/k)$ expected objective bound for averaged iterates. Furthermore, in the presence of strong convexity (for instance after adding a positive quadratic penalty to logistic regression), it has a linear convergence rate, i.e.

$$\mathbb{E}(f(x_k)) - f(x^\star) = O\left(\rho^k\right),$$

for some $0 < \rho < 1$.

These improvements exploit the finite-sum structure: SAG stores information about each of the n summands. SGD and averaged SGD also extend beyond finite n to general expectations (15.9).

Figure 15.7 shows a comparison of SGD, SGA and SAG.

15.3 Automatic Differentiation

Gradient evaluation $\nabla f(x)$ is the main operation in batch and stochastic gradient methods. For linear models and shallow networks, explicit formulas usually reduce it to matrix-vector products. For deeper or more elaborate computations, expanded derivative formulas become unwieldy and can duplicate work. Automatic differentiation organizes the chain rule recursively to reuse intermediate results.

15.3.1 Finite Differences and Symbolic Calculus

For $f : \mathbb{R}^p \to \mathbb{R}$, we want to evaluate $\nabla f : \mathbb{R}^p \mapsto \mathbb{R}^p$. A finite-difference approximation with a small parameter $\varepsilon > 0$ is

$$\frac{1}{\varepsilon}(f(x+\varepsilon\delta_1) - f(x), \ldots, f(x+\varepsilon\delta_p) - f(x))^\top \approx \nabla f(x)$$

This requires $p+1$ evaluations of f. Here $\delta_k = (0, \ldots, 0, 1, 0, \ldots, 0)$ is the coordinate vector with its 1 at position k. For large p, repeated evaluations are expensive. Finite differences also balance truncation error against floating-point cancellation as ε decreases. Reverse-mode automatic differentiation computes the full

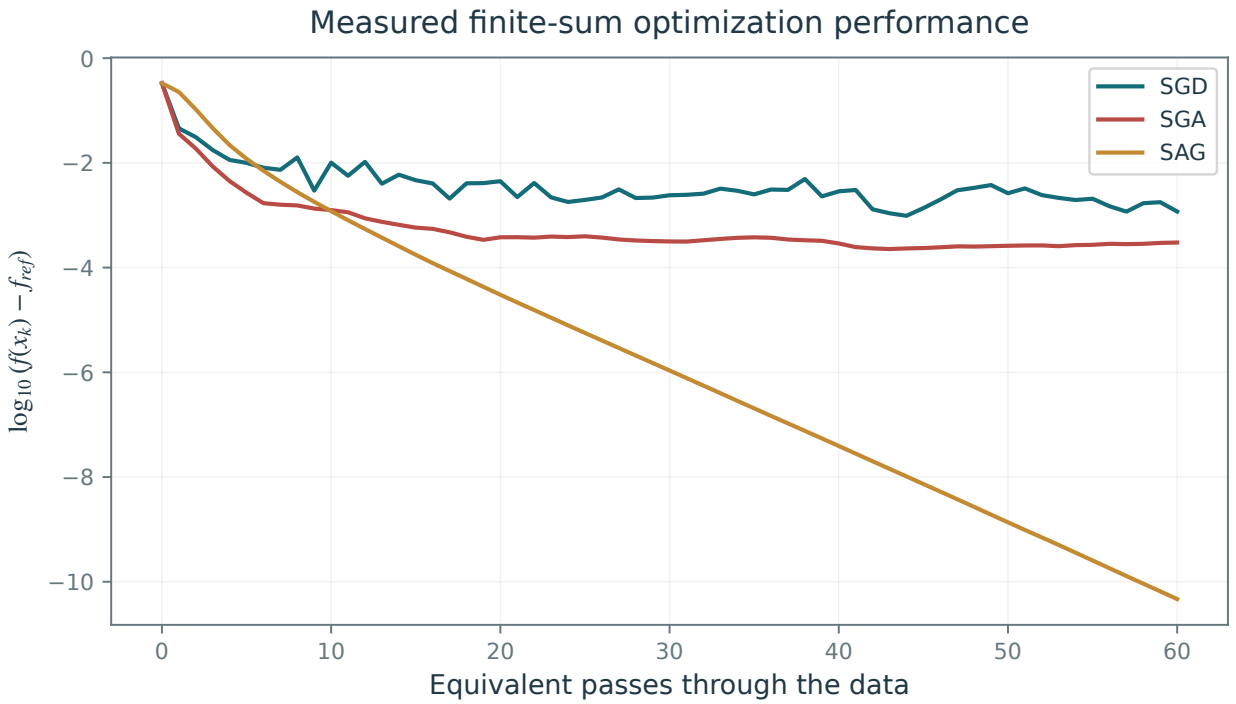

Figure 15.7. Objective gaps for SGD, averaged SGD (SGA), and SAG, displayed on logarithmic axes.

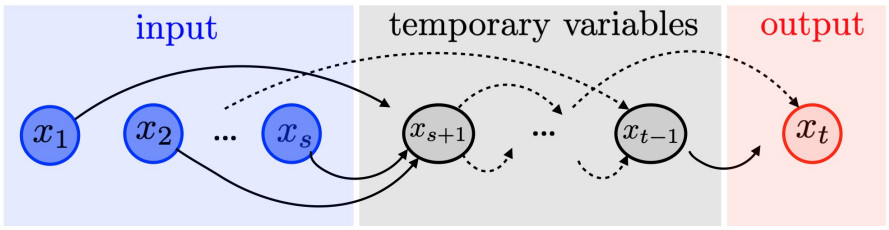

Figure 15.8. A computational graph.

gradient at a cost typically proportional to one evaluation of f. Like symbolic differentiation, automatic differentiation applies exact derivative rules, subject to floating-point error. It organizes those rules around the operations of the program that evaluates the function, preserving reuse of intermediate computations.

15.3.2 Computational Graphs

Let $x = (x_1, \dots, x_s)$ be the inputs to a program evaluating f. Number its intermediate variables $x_{s+1}, \dots, x_t$, with output $x_t = f(x)$, and let $x_k \in \mathbb{R}^{n_k}$. We seek the Jacobian blocks $\frac{\partial f(x)}{\partial x_k} \in \mathbb{R}^{n_t \times n_k}$ for $k = 1, \dots, s$. The scalar case $n_k = 1$ is easiest to follow; the formulas also apply to vector variables if the matrix products retain the displayed order. The total input dimension is $p = \sum_{k=1}^{s} n_k$.

Represent the computation as a sequence of elementary maps:

$$\forall\, k = s+1, \dots, t, \quad x_k = f_k(x_1, \dots, x_{k-1})$$

Each f_k depends only on earlier variables; see Figure 15.8. The directed acyclic graph (DAG) connects each argument used by f_k to its output x_k. The indexing is a topological order: every edge points from a smaller to a larger index. Evaluating $f(x)$ is a forward traversal of the graph. Automatic differentiation differentiates the supplied computational graph. It does not by itself choose an efficient implementation of the original formula; the derivative computation benefits from the efficiency and reuse already present in the forward program.

15.3.3 Forward Mode of Automatic Differentiation

Forward mode propagates derivatives of each intermediate variable with respect to a chosen input block, here x_1. For scalar inputs, repeating the traversal for all p coordinates gives the full derivative. The formulas

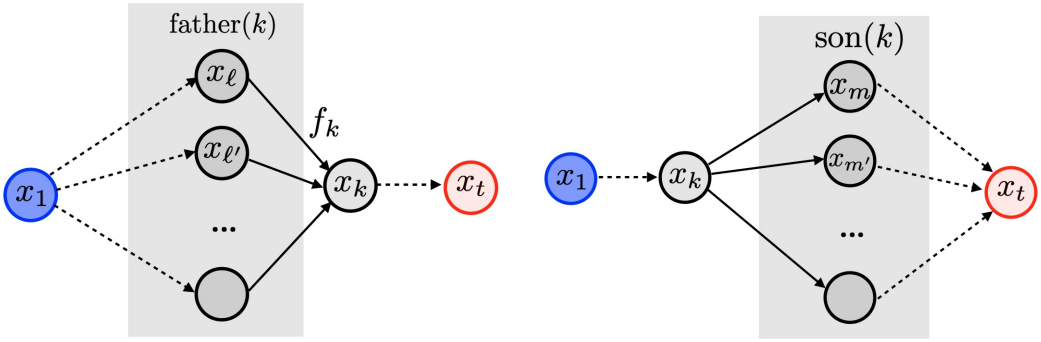

Figure 15.9. Derivative dependencies in forward mode (left) and reverse mode (right).

below also propagate a block Jacobian when $n_1 > 1$.

Initialize the input derivatives by

$$\frac{\partial x_1}{\partial x_1} = \mathrm{Id}_{n_1 \times n_1}, \quad \frac{\partial x_2}{\partial x_1} = 0_{n_2 \times n_1}, \ldots, \quad \frac{\partial x_s}{\partial x_1} = 0_{n_s \times n_1},$$

These are one and zero for scalar inputs. Propagate the derivatives using

$$\forall\, k = s+1, \ldots, t, \quad \frac{\partial x_k}{\partial x_1} = \sum_{\ell \in \mathrm{parent}(k)} \left[\frac{\partial x_k}{\partial x_\ell}\right] \times \frac{\partial x_\ell}{\partial x_1} = \sum_{\ell \in \mathrm{parent}(k)} \frac{\partial f_k}{\partial x_\ell}(x_1, \ldots, x_{k-1}) \times \frac{\partial x_\ell}{\partial x_1}.$$

The notation "parent(k)" denotes the nodes $\ell < k$ of the graph that are connected to k, see Figure 15.9, left. The stored quantities are the derivatives $\frac{\partial x_\ell}{\partial x_1}$. The symbol $\times$ denotes matrix multiplication in the vector-valued case. The brackets $[\ldots]$ indicate a local derivative: $\frac{\partial x_k}{\partial x_\ell}$ is the derivative of the elementary map f_k with respect to that argument. It can be evaluated when needed, assuming the local derivative is available in closed form.

Suppose the local derivative operations $\frac{\partial f_k}{\partial x_\ell}$ have comparable costs, as when the variable dimensions n_k are uniformly bounded, and each node has a bounded number of parents. Computing the full Jacobian coordinate by coordinate requires p traversals, each with cost comparable to a forward evaluation. This becomes expensive for large p.

Simple example. We consider the function

$$f(x, y) = y \log(x) + \sqrt{y \log(x)} \tag{15.19}$$

on the open domain $x > 0$, $y \log x > 0$, where all elementary derivatives exist. Its computational graph is displayed in Figure 15.10. The iterations of the forward mode to compute the derivative with respect to x read

$$\begin{aligned}
&\frac{\partial x}{\partial x} = 1, \quad \frac{\partial y}{\partial x} = 0 \\
&\frac{\partial a}{\partial x} = \left[\frac{\partial a}{\partial x}\right] \frac{\partial x}{\partial x} = \frac{1}{x}\frac{\partial x}{\partial x} && \{x \mapsto a = \log(x)\} \\
&\frac{\partial b}{\partial x} = \left[\frac{\partial b}{\partial a}\right] \frac{\partial a}{\partial x} + \left[\frac{\partial b}{\partial y}\right] \frac{\partial y}{\partial x} = y\frac{\partial a}{\partial x} + 0 && \{(y, a) \mapsto b = ya\} \\
&\frac{\partial c}{\partial x} = \left[\frac{\partial c}{\partial b}\right] \frac{\partial b}{\partial x} = \frac{1}{2\sqrt{b}}\frac{\partial b}{\partial x} && \{b \mapsto c = \sqrt{b}\} \\
&\frac{\partial f}{\partial x} = \left[\frac{\partial f}{\partial b}\right] \frac{\partial b}{\partial x} + \left[\frac{\partial f}{\partial c}\right] \frac{\partial c}{\partial x} = 1\frac{\partial b}{\partial x} + 1\frac{\partial c}{\partial x} && \{(b, c) \mapsto f = b + c\}
\end{aligned}$$

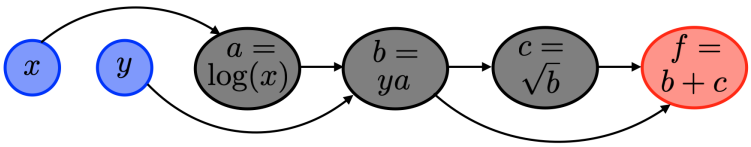

Figure 15.10. Example of a simple computational graph.

A second forward pass gives the derivative with respect to y:

$$\begin{aligned}
&\frac{\partial x}{\partial y} = 0, \quad \frac{\partial y}{\partial y} = 1 \\
&\frac{\partial a}{\partial y} = \left[\frac{\partial a}{\partial x}\right]\frac{\partial x}{\partial y} = 0 && \{x \mapsto a = \log(x)\} \\
&\frac{\partial b}{\partial y} = \left[\frac{\partial b}{\partial a}\right]\frac{\partial a}{\partial y} + \left[\frac{\partial b}{\partial y}\right]\frac{\partial y}{\partial y} = 0 + a\frac{\partial y}{\partial y} && \{(y, a) \mapsto b = ya\} \\
&\frac{\partial c}{\partial y} = \left[\frac{\partial c}{\partial b}\right]\frac{\partial b}{\partial y} = \frac{1}{2\sqrt{b}}\frac{\partial b}{\partial y} && \{b \mapsto c = \sqrt{b}\} \\
&\frac{\partial f}{\partial y} = \left[\frac{\partial f}{\partial b}\right]\frac{\partial b}{\partial y} + \left[\frac{\partial f}{\partial c}\right]\frac{\partial c}{\partial y} = 1\frac{\partial b}{\partial y} + 1\frac{\partial c}{\partial y} && \{(b, c) \mapsto f = b + c\}
\end{aligned}$$

Dual numbers. Dual numbers provide a convenient implementation of the forward pass. They form a real algebra whose elements have the form $x + \varepsilon x'$ where ε is a symbol obeying the rule that $\varepsilon^2 = 0$. Here $(x, x') \in \mathbb{R}^2$ and x' is intended to store a derivative with respect to some input variable. Multiplication and, when $x \neq 0$, inversion are given by

$$(x + \varepsilon x')(y + \varepsilon y') = xy + \varepsilon(xy' + yx') \quad \text{and} \quad \frac{1}{x + \varepsilon x'} = \frac{1}{x} - \varepsilon\frac{x'}{x^2}.$$

For polynomial or rational f, the arithmetic rules imply

$$f(x + \varepsilon) = f(x) + \varepsilon f'(x).$$

For another differentiable elementary function f, define its action on dual numbers by

$$f(x + \varepsilon x') \overset{\text{def.}}{=} f(x) + \varepsilon f'(x)x'.$$

This definition gives

$$(f \circ g)(x + \varepsilon) = f(g(x)) + \varepsilon f'(g(x))g'(x)$$

which is the chain rule. More generally, for $f(x_1, \ldots, x_s)$ built from these elementary operations,

$$f(x_1 + \varepsilon, x_2, \ldots, x_s) = f(x_1, \ldots, x_s) + \varepsilon\frac{\partial f}{\partial x_1}(x_1, \ldots, x_s)$$

the dual-number evaluation computes $\frac{\partial f}{\partial x_1}(x_1, \ldots, x_s)$ by forward-mode differentiation. The other coordinates are treated identically.

15.3.4 Reverse Mode of Automatic Differentiation

Reverse mode computes the derivatives $\frac{\partial x_t}{\partial x_k}$ of the output with respect to the intermediate variables. For a scalar output and many input coordinates, this avoids the repeated forward traversals required to compute each $\frac{\partial x_k}{\partial x_i}$ separately.

Initialize the derivative of the output node by

$$\frac{\partial x_t}{\partial x_t} = \mathrm{Id}_{n_t \times n_t},$$

and apply the following recursion from the last node to the first:

$$\forall\, k = t-1, t-2, \ldots, 1, \quad \frac{\partial x_t}{\partial x_k} = \sum_{m \in \mathrm{son}(k)} \frac{\partial x_t}{\partial x_m} \times \left[\frac{\partial x_m}{\partial x_k}\right] = \sum_{m \in \mathrm{son}(k)} \frac{\partial x_t}{\partial x_m} \times \frac{\partial f_m(x_1, \ldots, x_{m-1})}{\partial x_k}.$$

The set $\mathrm{son}(k)$ contains the children $m > k$ that depend directly on x_k, as shown in Figure 15.9, right. Contributions from every child must be added.

Back-propagation. For a scalar output $x_t \in \mathbb{R}$, we have $\frac{\partial x_t}{\partial x_k} = [\nabla_{x_k} f(x)]^\top \in \mathbb{R}^{1 \times n_k}$. The reverse recursion can then be written for gradient vectors:

$$\forall\, k = t-1, t-2, \ldots, 1, \quad \nabla_{x_k} f(x) = \sum_{m \in \mathrm{son}(k)} \left(\frac{\partial f_m(x_1, \ldots, x_{m-1})}{\partial x_k}\right)^\top (\nabla_{x_m} f(x)).$$

Here $\left(\frac{\partial f_m(x_1,\ldots,x_{m-1})}{\partial x_k}\right)^\top \in \mathbb{R}^{n_k \times n_m}$ is the adjoint Jacobian of f_m. This adjoint recursion is called backpropagation and is the standard form used in machine learning.

For a scalar output ($n_t = 1$), reverse mode is usually the preferred way to compute the full gradient. It requires access to the intermediate variables $(x_k)_{k=s+1}^t$, which can be costly to store for a large graph. Checkpointing trades additional computation for reduced storage.

Simple example. For the two-input example $f(x, y)$ in (15.19), reverse mode gives

$$\begin{aligned}
&\frac{\partial f}{\partial f} = 1 \\
&\frac{\partial f}{\partial c} = \frac{\partial f}{\partial f}\left[\frac{\partial f}{\partial c}\right] = \frac{\partial f}{\partial f} 1 && \{c \mapsto f = b + c\} \\
&\frac{\partial f}{\partial b} = \frac{\partial f}{\partial c}\left[\frac{\partial c}{\partial b}\right] + \frac{\partial f}{\partial f}\left[\frac{\partial f}{\partial b}\right] = \frac{\partial f}{\partial c}\frac{1}{2\sqrt{b}} + \frac{\partial f}{\partial f} 1 && \{b \mapsto c = \sqrt{b}, b \mapsto f = b + c\} \\
&\frac{\partial f}{\partial a} = \frac{\partial f}{\partial b}\left[\frac{\partial b}{\partial a}\right] = \frac{\partial f}{\partial b} y && \{a \mapsto b = ya\} \\
&\frac{\partial f}{\partial y} = \frac{\partial f}{\partial b}\left[\frac{\partial b}{\partial y}\right] = \frac{\partial f}{\partial b} a && \{y \mapsto b = ya\} \\
&\frac{\partial f}{\partial x} = \frac{\partial f}{\partial a}\left[\frac{\partial a}{\partial x}\right] = \frac{\partial f}{\partial a}\frac{1}{x} && \{x \mapsto a = \log(x)\}
\end{aligned}$$

A single reverse traversal computes both derivatives with respect to x and y, whereas coordinatewise forward mode requires two passes.

15.3.5 Feed-forward Compositions

For a feedforward chain, the computed function is the composition

$$f = f_t \circ f_{t-1} \circ \ldots \circ f_2 \circ f_1 \tag{15.20}$$

for functions $f_k : \mathbb{R}^{n_{k-1}} \to \mathbb{R}^{n_k}$.

The forward evaluation starts from $x_0 = x \in \mathbb{R}^{n_0}$ and computes

$$\forall\, k = 1, \ldots, t, \quad x_k = f_k(x_{k-1})$$

giving the output $f(x) = x_t$.

Let $A_k \overset{\text{def.}}{=} \partial f_k(x_{k-1}) \in \mathbb{R}^{n_k \times n_{k-1}}$ be the Jacobian of the local map. The chain rule gives

$$\partial f(x) = A_t \times A_{t-1} \times \ldots A_2 \times A_1.$$

Forward mode multiplies the Jacobians from right to left, while reverse mode multiplies them from left to right

$$\begin{aligned}
\partial f(x) &= A_t \times (A_{t-1} \times (\ldots \times (A_3 \times (A_2 \times A_1)))), \\
\partial f(x) &= ((((A_t \times A_{t-1}) \times A_{t-2}) \times \ldots) \times A_2) \times A_1.
\end{aligned}$$

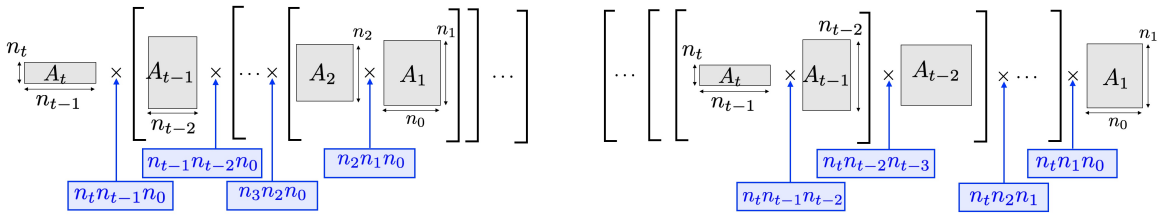

Figure 15.11. Matrix multiplication costs for forward mode (left) and reverse mode (right) along a chain.

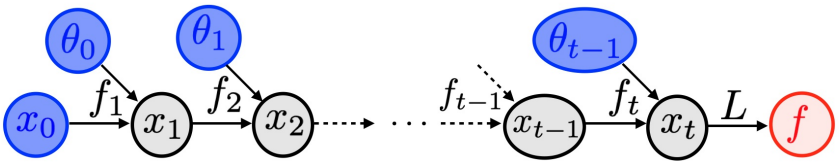

Figure 15.12. Computational graph for a feedforward architecture.

We note that the computation of the product $A \times B$ of $A \in \mathbb{R}^{n \times p}$ with $B \in \mathbb{R}^{p \times q}$ requires $O(npq)$ arithmetic operations with the standard dense algorithm. As shown on Figure 15.11, the arithmetic costs of the two parenthesizations are proportional to

$$n_0 \sum_{k=1}^{t-1} n_k n_{k+1} \quad \text{and} \quad n_t \sum_{k=0}^{t-2} n_k n_{k+1}$$

For roughly comparable intermediate widths, these costs favor reverse mode when $n_t \ll n_0$. A scalar learning objective typically has $n_t = 1$, making reverse mode a natural choice; the exact cost comparison depends on all the intermediate dimensions.

15.3.6 Feed-forward Architecture

A parameterized feedforward architecture, such as a neural network, has the form

$$\forall\, k = 1, \dots, t, \quad x_k = f_k(x_{k-1}, \theta_{k-1}) \tag{15.21}$$

where θ_{k-1} is the parameter vector for a layer and the input $x_0 \in \mathbb{R}^{n_0}$ is fixed. The objective is

$$f(\theta) \overset{\text{def.}}{=} L(x_t) \tag{15.22}$$

Here $L : \mathbb{R}^{n_t} \to \mathbb{R}$ is a loss, such as squared prediction error or logistic loss, and $\theta = (\theta_k)_{k=0}^{t-1}$ collects the parameters. Figure 15.12 shows the computational graph.

Reverse mode computes the gradient of f by propagating derivatives through the variables (x_k, θ_k). Initialize

$$\nabla_{x_t} f = \nabla L(x_t)$$

and recurse from $k = t$ down to 1:

$$z_{k-1} = [\partial_x f_k(x_{k-1}, \theta_{k-1})]^\top z_k \quad \text{and} \quad \nabla_{\theta_{k-1}} f = [\partial_\theta f_k(x_{k-1}, \theta_{k-1})]^\top (\nabla_{x_k} f) \tag{15.23}$$

where $z_k \overset{\text{def.}}{=} \nabla_{x_k} f(\theta)$ is the gradient with respect to x_k.

Multilayer perceptrons. For a fully connected network without biases, use

$$\forall\, x_{k-1} \in \mathbb{R}^{n_{k-1}}, \quad f_k(x_{k-1}, \theta_{k-1}) = \rho(\theta_{k-1} x_{k-1}) \tag{15.24}$$

where $\theta_{k-1} \in \mathbb{R}^{n_k \times n_{k-1}}$ contains the weights and ρ acts coordinatewise; see Figure 15.13. The derivative formulas below apply where ρ is differentiable. At a ReLU kink, a chosen derivative convention defines the computed update, but the classical derivative need not exist. For an adjoint vector $z_k \in \mathbb{R}^{n_k}$, typically $\nabla_{x_k} f$,

$$\begin{cases} [\partial_x f_k(x_{k-1}, \theta_{k-1})]^\top (z_k) = \theta_{k-1}^\top w_k z_k, \\ [\partial_\theta f_k(x_{k-1}, \theta_{k-1})]^\top (z_k) = (w_k z_k) x_{k-1}^\top \end{cases} \quad \text{where} \quad w_k \overset{\text{def.}}{=} \operatorname{diag}(\rho'(\theta_{k-1} x_{k-1})).$$

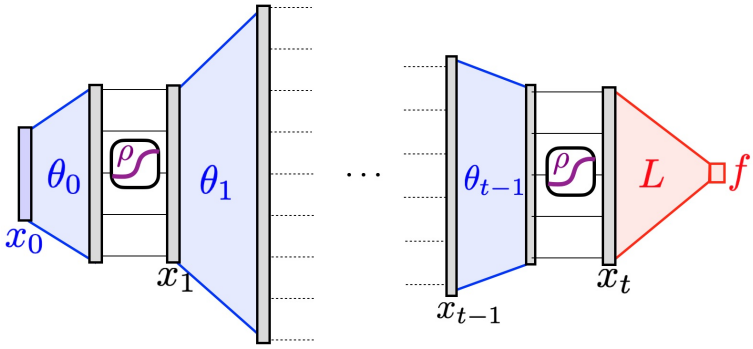

Figure 15.13. Multilayer-perceptron parameterization.

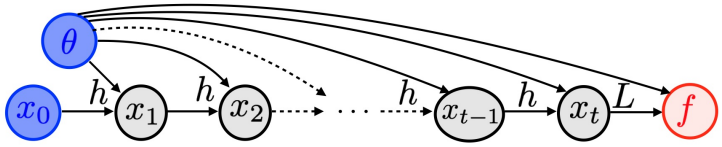

Figure 15.14. Computational graph for a recurrent architecture.

Link with adjoint state method. A residual form of (15.21) can be interpreted as a time discretization of an ordinary differential equation. Keep the state dimension fixed, $n_k = n$, and interpret x_k as an approximation of a continuous trajectory $x(t)$ at time $t = k\tau$. The formal continuum limit takes $\tau \downarrow 0$ with $k\tau$ approaching a fixed time. Impose the residual structure

$$f_k(x_{k-1}, \theta_{k-1}) = x_{k-1} + \tau u(x_{k-1}, \theta_{k-1}, k\tau) \tag{15.25}$$

with a parameterized vector field $u(x, \theta, t) \in \mathbb{R}^n$. Formally, as $\tau \to 0$, this approaches the nonlinear ODE

$$\dot{x}(t) = u(x(t), \theta(t), t) \tag{15.26}$$

with $x(t = 0) = x_0$.

Writing $z(t) = \nabla_{x(t)} f(\theta)$ for the adjoint, the discrete relations (15.23) become a linear backward ODE:

$$\dot{z}(t) = -[\partial_x u(x(t), \theta(t), t)]^\top z(t) \quad \text{and} \quad \nabla_{\theta(t)} f(\theta) = [\partial_\theta u(x(t), \theta(t), t)]^\top z(t).$$

The adjoint has terminal condition $z(T) = \nabla L(x(T))$. The parameter derivative is interpreted as the L^2 functional gradient with respect to the time-dependent control. Its discrete normalization is $\frac{1}{\tau}\nabla_{\theta_{k-1}} f \to \nabla_{\theta(t)} f(\theta)$.

15.3.7 Recurrent Architectures

A recurrent architecture reuses the parameter $\theta = \theta_k$ and map $f_k = h$ at every step of (15.21):

$$\forall\, k = 1, \dots, t, \quad x_k = h(x_{k-1}, \theta). \tag{15.27}$$

Consider a scalar objective

$$f(\theta) = L(x_t, \theta)$$

The loss now depends explicitly on θ, extending (15.22). Figure 15.14 shows the corresponding graph.

Backpropagation gives

$$\nabla_{x_{k-1}} f = [\partial_x h(x_{k-1}, \theta)]^\top \nabla_{x_k} f \quad \text{and} \quad \nabla_\theta f = \nabla_\theta L(x_t, \theta) + \sum_k [\partial_\theta h(x_{k-1}, \theta)]^\top \nabla_{x_k} f. \tag{15.28}$$

Similarly, with $h(x, \theta) = x + \tau u(x, \theta)$, the formal limit $(k, k\tau) \to (+\infty, t)$ gives a forward nonlinear ODE with a time-independent vector field:

$$\dot{x}(t) = u(x(t), \theta)$$

and the following linear backward adjoint equation, for $f(\theta) = L(x(T), \theta)$

$$\dot{z}(t) = -[\partial_x u(x(t), \theta)]^\top z(t) \quad \text{and} \quad \nabla_\theta f(\theta) = \nabla_\theta L(x(T), \theta) + \int_0^T [\partial_\theta u(x(t), \theta)]^\top z(t) \mathrm{d}t. \tag{15.29}$$

with terminal condition $z(T) = \nabla_x L(x(T), \theta)$.

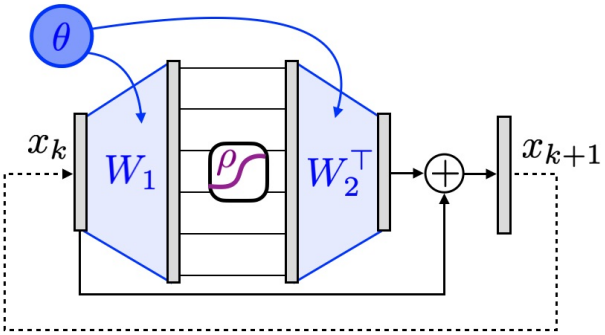

Figure 15.15. Recurrent residual perceptron parameterization.

Residual recurrent networks. A residual recurrent network uses the update

$$h(x, \theta) = x + W_2^\top \rho(W_1 x)$$

as illustrated in Figure 15.15. Here $\theta = (W_1, W_2) \in (\mathbb{R}^{q \times n})^2$ contains the weights and ρ is a pointwise activation. Increasing the hidden width q enlarges the class of representable residual maps. When $W_2 = -\tau W_1$ and $\rho = \psi'$, this update is a gradient step for $\mathcal{E}(x, \theta) = \sum_i \psi((W_1 x)_i)$. Repeating it gives an iterative implementation of the argmin layer (15.31), provided the iterations converge to a minimizer. The Jacobians $\partial_\theta h$ and $\partial_x h$ follow from the layer formulas in the multilayer-perceptron discussion.

Mitigating Memory Requirements. Backpropagation requires access to the iterates $(x_k)_{k=0}^t$, whose storage can dominate memory use. Checkpointing stores selected states and reruns parts of the forward computation to reconstruct missing values. Recursive schedules can use $O(\log t)$ stored states with an $O(\log t)$ computational overhead.

If the forward computation is invertible, stored states can sometimes be reconstructed instead. Assume maps g_k satisfy

$$x_k = g_k(x_{k+1}, \dots, x_t).$$

These inverse updates may amplify numerical errors, so algebraic invertibility alone does not ensure stable reconstruction. In practice, the inverse can also depend on a few extra variables, in particular on the input values $(x_0, \dots, x_s)$.

For example, split the continuous state as $x(t) = (r(t), s(t))$ and suppose the vector field in (15.26) has the separated form $u((r, s), \theta, t) = (F(s, \theta, t), G(r, \theta, t))$. An invertible staggered update is

$$r_{k+1} = r_k + \tau F(s_k, \theta_k, \tau k) \quad \text{and} \quad s_{k+1} = s_k + \tau G(r_{k+1}, \theta_{k+1/2}, \tau(k + 1/2)).$$

In exact arithmetic, these updates can be inverted as

$$s_k = s_{k+1} - \tau G(r_{k+1}, \theta_{k+1/2}, \tau(k + 1/2)). \quad \text{and} \quad r_k = r_{k+1} - \tau F(s_k, \theta_k, \tau k).$$

Fixed point maps If h is continuous and the iterates converge to $x^\star(\theta)$, their limit is a fixed point:

$$x^\star(\theta) = h(x^\star(\theta), \theta).$$

Backpropagation differentiates the finite computation $f(\theta) = L(x_t, \theta)$. For the limiting objective $f^\star(\theta) = L(x^\star(\theta), \theta)$, the implicit function theorem gives an alternative:

$$\nabla f^\star(\theta) = [\partial x^\star(\theta)]^\top (\nabla_x L(x^\star(\theta), \theta)) + \nabla_\theta L(x^\star(\theta), \theta). \tag{15.30}$$

Assume h is continuously differentiable and $\mathrm{Id} - \partial_x h$ is invertible at the fixed point. The implicit function theorem then gives

$$\partial x^\star(\theta) = \big(\mathrm{Id} - \partial_x h(x^\star(\theta), \theta)\big)^{-1} \partial_\theta h(x^\star(\theta), \theta).$$

In practice, one replaces in these formulas $x^\star(\theta)$ by x_t, which approximates the derivative of the converged objective $f^\star$, rather than the derivative of the finite unrolling f. This approach replaces storage of the iteration history by a linear solve at the approximate equilibrium.

Argmin layers An argmin layer maps a parameter θ to a minimizer $x(\theta)$ of a parameterized objective. Assume a unique minimizer, or specify a branch when several exist:

$$x(\theta) = \operatorname*{argmin}_x \mathcal{E}(x, \theta).$$

One way to compute the layer is gradient descent, initialized here at $x_0 = 0$:

$$x_{k+1} = x_k - \tau \nabla \mathcal{E}(x_k, \theta). \tag{15.31}$$

This has the form (15.25) when using the vector field $u(x, \theta) = -\nabla_x \mathcal{E}(x, \theta)$.

Assume $\mathcal{E}$ is twice continuously differentiable near the selected minimizer and its Hessian with respect to x is invertible. The implicit function theorem gives a locally differentiable minimizer branch; applying (15.30) to its stationarity equation yields

$$\nabla f^\star(\theta) = -\left(\frac{\partial^2 \mathcal{E}}{\partial x \partial \theta}(x^\star(\theta), \theta)\right)^\top \left(\frac{\partial^2 \mathcal{E}}{\partial x^2}(x^\star(\theta), \theta)\right)^{-1} (\nabla_x L(x^\star(\theta), \theta)) + \nabla_\theta L(x^\star(\theta), \theta)$$

If $f(\theta)$ is the minimized value itself, $f(\theta) = \mathcal{E}(x^\star(\theta), \theta)$, then $L = \mathcal{E}$. Formula (15.30) simplifies because $\nabla_x L(x^\star(\theta), \theta) = 0$, giving

$$\nabla f^\star(\theta) = \nabla_\theta L(x^\star(\theta), \theta). \tag{15.32}$$

This is the envelope theorem, also associated with Danskin's theorem.

Sinkhorn's algorithm Let $a \in \mathbb{R}^n$ and $b \in \mathbb{R}^m$ be strictly positive probability vectors, let $C \in \mathbb{R}^{n \times m}$ be a cost matrix, and let $\varepsilon > 0$. Entropic optimal transport minimizes

$$\sum_{ij} C_{ij} P_{ij} + \varepsilon \sum_{ij} P_{ij}(\log P_{ij} - 1)$$

over $P \geqslant 0$ with row sums a and column sums b. Set $K_{ij} = e^{-C_{ij}/\varepsilon} > 0$. Sinkhorn's alternating updates, initialized with positive scaling vectors, are

$$u_{k+1} = \frac{a}{K v_k}, \qquad v_{k+1} = \frac{b}{K^\top u_{k+1}}.$$

All divisions are componentwise. The second update uses the newly computed u_{k+1}; this sequential dependence is part of Sinkhorn's alternating scheme.

The regularized transport value is the maximum of the dual objective

$$\mathcal{E}(u, v; a, b) = \varepsilon \left(\langle a, \log u \rangle + \langle b, \log v \rangle - \langle u, Kv \rangle \right).$$

Sinkhorn performs alternating maximization of this objective. A finite number of iterations defines a differentiable recurrent map on positive inputs. Its Jacobian products can be computed by chaining the two scaling steps. For example, for $u = a/(Kv)$,

$$\partial_v u = -\operatorname{diag}\left(\frac{a}{(Kv)^2}\right) K, \qquad \partial_a u = \operatorname{diag}\left(\frac{1}{Kv}\right),$$

and analogous formulas apply to the v update, with the dependence through the newly computed u included.

At convergence, the envelope theorem gives marginal derivatives represented by the dual potentials

$$\nabla_a f^\star = \varepsilon \log u^\star, \qquad \nabla_b f^\star = \varepsilon \log v^\star.$$

The scalings have the invariance $(u, v) \mapsto (cu, v/c)$, so these potentials are defined up to opposite additive constants. On the probability simplices, derivatives act on zero-sum perturbations and are therefore independent of this gauge. An implicit differentiation formula must fix the gauge before inverting a Jacobian; the unmodified fixed-point Jacobian is singular in that direction.