Data Science for Mathematicians
Lesson 5: Gradient Descent

February 18, 2026

Abstract

This lecture establishes the link between multivariable calculus and the task of training a machine learning model. We define *learning* as an optimization problem: finding model parameters that minimize a loss function over a high-dimensional space. We derive the gradient descent algorithm from a first-order Taylor expansion and analyze the critical role of the learning rate hyperparameter. We then introduce convex optimization theory, proving that for convex loss functions—such as the mean squared error in ordinary least squares—any local minimum is a global minimum, guaranteeing convergence of gradient descent to the optimal solution. Finally, we address the computational challenges of large-scale datasets by developing stochastic gradient descent and mini-batch gradient descent. We analyze these stochastic methods through unbiased estimation and explore the trade-offs between computational efficiency, update variance, and hardware parallelization.

1 The Engine of Learning

In previous lectures, we constructed the foundational framework of a supervised machine learning problem. This framework consists of two components. First, a **model**—a parameterized function $f_{\boldsymbol{\theta}}(\mathbf{x})$ that maps inputs $\mathbf{x}$ from a feature space to a predicted output. The vector $\boldsymbol{\theta} \in \mathbb{R}^p$ contains the p parameters that define the model's behavior. Second, a **loss function** $L(\boldsymbol{\theta})$—a scalar-valued function that quantifies the aggregate discrepancy between the model's predictions and the true target values across a dataset. For the model of ordinary least squares (OLS), where our model is $f_{\boldsymbol{\beta}}(\mathbf{x}) = \mathbf{x}^T\boldsymbol{\beta}$, we focused on the mean squared error (MSE) loss function:

$$L(\boldsymbol{\beta}) = \frac{1}{n}\sum_{i=1}^{n}(\mathbf{x}_i^T\boldsymbol{\beta} - y_i)^2 = \frac{1}{n}\|\mathbf{X}\boldsymbol{\beta} - \mathbf{y}\|_2^2.$$

Here, $\mathbf{X} \in \mathbb{R}^{n\times p}$ is the design matrix, $\mathbf{y} \in \mathbb{R}^n$ is the vector of true outcomes, and $\boldsymbol{\beta} \in \mathbb{R}^p$ is the parameter vector we wish to determine. The loss function $L(\boldsymbol{\beta})$ defines a high-dimensional surface, called the **loss landscape**, where the value at any point $\boldsymbol{\beta}$ corresponds to the total error of the model with that parameterization.

This construction leads us to the central problem in machine learning. We seek the parameter vector $\boldsymbol{\theta}^*$ that minimizes the loss function:

$$\boldsymbol{\theta}^* = \arg\min_{\boldsymbol{\theta}\in\mathbb{R}^p} L(\boldsymbol{\theta}).$$

The act of *training* a model is the process of solving this optimization problem.

For the specific case of OLS, we found a rare closed-form solution. By computing the gradient of the loss function with respect to $\boldsymbol{\beta}$ and setting it to the zero vector,

$$\nabla L(\boldsymbol{\beta}) = \frac{2}{n}\mathbf{X}^T(\mathbf{X}\boldsymbol{\beta} - \mathbf{y}) = \mathbf{0},$$

and solving the resulting system of linear equations (the normal equations),

$$\mathbf{X}^T\mathbf{X}\boldsymbol{\beta} = \mathbf{X}^T\mathbf{y},$$

we derived the explicit solution

$$\boldsymbol{\beta}^* = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{y}.$$

This solution is elegant and exact. However, its existence is an anomaly in the broader landscape of machine learning. Such an analytical solution fails under two common conditions.

First, consider **analytical intractability**. For many powerful models—including logistic regression and virtually all deep neural networks—the equation $\nabla L(\boldsymbol{\theta}) = \mathbf{0}$ yields a system of non-linear equations with no closed-form solution.

Second, even when an analytical solution exists, we face **computational infeasibility**. The normal equations require computing the inverse of $\mathbf{X}^T\mathbf{X}$, a $p \times p$ matrix. Standard matrix inversion has computational complexity $O(p^3)$. For $p = 100{,}000$, this operation is prohibitively expensive. Furthermore, when the number of samples n is in the billions, the design matrix $\mathbf{X}$ itself may be too large to fit into memory.

These limitations reveal that the transition from analytical to iterative methods is not merely a practical workaround; it represents a fundamental paradigm shift. Iterative methods do not seek a perfect, global answer in a single step. Instead, they make local, incremental progress toward a solution, making them far more broadly applicable.

This lecture develops the fundamental iterative algorithm that serves as the engine of machine learning: **Gradient Descent**. We leverage multivariable calculus to formalize the loss landscape and derive a principled algorithm for navigating this surface to find its lowest point.

2 The Calculus of Loss Landscapes

To derive a principled optimization algorithm, we must first formalize the geometric space in which our search for optimal parameters takes place. The geometric properties of the loss landscape, described by calculus, dictate the behavior and success of our algorithm.

Definition 2.1 (Loss Landscape)**.** The **loss landscape** of a machine learning model with parameter vector $\boldsymbol{\theta} \in \mathbb{R}^p$ and loss function $L(\boldsymbol{\theta})$ is the scalar field $L\colon \mathbb{R}^p \to \mathbb{R}$. Each point in the domain represents a unique parameterization of the model, and the function value at that point represents the corresponding model error.

Example 2.2 (Regression through the Origin, $p = 1$)**.** Consider the model $y = \beta_1 x$ with a single parameter $\theta = \beta_1 \in \mathbb{R}$. The MSE loss is $L(\beta_1) = \frac{1}{n}\sum_{i=1}^{n}(\beta_1 x_i - y_i)^2$. The loss landscape $L\colon \mathbb{R} \to \mathbb{R}$ is a parabola, plotted with β_1 on the horizontal axis and L on the vertical axis.

Example 2.3 (Simple Linear Regression, $p = 2$)**.** For the model $y = \beta_0 + \beta_1 x$, the parameter vector is $\boldsymbol{\theta} = [\beta_0, \beta_1]^T \in \mathbb{R}^2$. The MSE loss is $L(\beta_0, \beta_1) = \frac{1}{n}\sum_{i=1}^{n}((\beta_0 + \beta_1 x_i) - y_i)^2$. The loss landscape $L\colon \mathbb{R}^2 \to \mathbb{R}$ is a 3-dimensional surface. The domain is the (β_0, β_1) plane, and the height at any point is the loss L. For OLS, this landscape is a convex paraboloid.

Example 2.4 (Multiple Linear Regression, $p > 2$)**.** Consider the model $y = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \cdots + \beta_m x_m$ with parameter vector $\boldsymbol{\theta} = [\beta_0, \beta_1, \ldots, \beta_m]^T \in \mathbb{R}^p$, where $p = m + 1$. The MSE loss is $L(\boldsymbol{\theta}) = \frac{1}{n}\sum_{i=1}^{n}\left(\left(\beta_0 + \sum_{j=1}^{m}\beta_j x_{ij}\right) - y_i\right)^2$. For $m = 50$ features, the loss landscape is $L\colon \mathbb{R}^{51} \to \mathbb{R}$—a 52-dimensional object impossible to visualize directly. We must rely on gradients and Hessians to navigate this high-dimensional convex paraboloid.

Our primary tools for analyzing and navigating this landscape are the gradient and the Hessian matrix.

2.1 The Gradient as a Vector Field

The first derivative of a multivariable function is its gradient. For the loss landscape, the gradient is a vector field that assigns a vector to every point in the parameter space.

Definition 2.5 (Gradient)**.** The **gradient** of the loss function $L(\boldsymbol{\theta})$ at a point $\boldsymbol{\theta} \in \mathbb{R}^p$, denoted $\nabla L(\boldsymbol{\theta})$ or $\nabla_{\boldsymbol{\theta}} L$, is the vector of first-order partial derivatives:

$$\nabla L(\boldsymbol{\theta}) = \begin{pmatrix} \frac{\partial L}{\partial \theta_1} \\ \frac{\partial L}{\partial \theta_2} \\ \vdots \\ \frac{\partial L}{\partial \theta_p} \end{pmatrix} = \left[\frac{\partial L}{\partial \theta_1}, \frac{\partial L}{\partial \theta_2}, \ldots, \frac{\partial L}{\partial \theta_p} \right]^T .$$

Example 2.6 (Gradient of a Quadratic in $\mathbb{R}^2$)**.** Let $L(\boldsymbol{\theta}) = \theta_1^2 + 4\theta_2^2$ for $\boldsymbol{\theta} = [\theta_1, \theta_2]^T \in \mathbb{R}^2$. The gradient is

$$\nabla L(\boldsymbol{\theta}) = \begin{pmatrix} 2\theta_1 \\ 8\theta_2 \end{pmatrix} .$$

At $\boldsymbol{\theta} = [1,1]^T$, we have $\nabla L = [2,8]^T$. The gradient points predominantly in the θ_2 direction, reflecting the steeper curvature along that axis. The magnitude $\|\nabla L\|_2 = \sqrt{4+64} = \sqrt{68} \approx 8.25$ indicates a steep slope. At the origin $\boldsymbol{\theta} = \mathbf{0}$, the gradient vanishes: $\nabla L = \mathbf{0}$, confirming that the origin is a critical point.

The gradient has a profound geometric meaning that is the cornerstone of our optimization strategy.

- **Direction of Steepest Ascent:** At any point $\boldsymbol{\theta}$ in the parameter space, the gradient vector $\nabla L(\boldsymbol{\theta})$ points in the direction in which L increases most rapidly.
- **Magnitude as Rate of Ascent:** The norm $\|\nabla L(\boldsymbol{\theta})\|_2$ quantifies the rate of increase. A large magnitude signifies a steep slope; a small magnitude signifies a gentle slope. At a point where $\|\nabla L(\boldsymbol{\theta})\|_2 = 0$, the landscape is locally flat, and we are at a **critical point** (or **stationary point**).
- **Direction of Steepest Descent:** The negative gradient, $-\nabla L(\boldsymbol{\theta})$, points in the direction of steepest local *descent.* To reduce the loss as efficiently as possible, we take a step in the direction of the negative gradient. This intuition forms the basis of the gradient descent algorithm.

2.2 The Hessian and Local Curvature

While the gradient describes the first-order behavior (slope) of the loss landscape, the Hessian matrix describes its second-order behavior (curvature).

Definition 2.7 (Hessian Matrix)**.** Let $L\colon \mathbb{R}^p \to \mathbb{R}$ be twice continuously differentiable. The **Hessian matrix** of L at a point $\boldsymbol{\theta} \in \mathbb{R}^p$, denoted $\mathbf{H}_L(\boldsymbol{\theta})$ or $\nabla^2 L(\boldsymbol{\theta})$, is the $p \times p$ matrix whose (i,j)-th entry is the second-order mixed partial derivative,

$$[\mathbf{H}_L(\boldsymbol{\theta})]_{ij} = \frac{\partial^2 L}{\partial \theta_i \, \partial \theta_j}, \qquad i,j = 1, \ldots, p.$$

Written explicitly,

$$\mathbf{H}_L(\boldsymbol{\theta}) = \begin{pmatrix} \frac{\partial^2 L}{\partial \theta_1^2} & \frac{\partial^2 L}{\partial \theta_1 \partial \theta_2} & \cdots & \frac{\partial^2 L}{\partial \theta_1 \partial \theta_p} \\ \frac{\partial^2 L}{\partial \theta_2 \partial \theta_1} & \frac{\partial^2 L}{\partial \theta_2^2} & \cdots & \frac{\partial^2 L}{\partial \theta_2 \partial \theta_p} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial^2 L}{\partial \theta_p \partial \theta_1} & \frac{\partial^2 L}{\partial \theta_p \partial \theta_2} & \cdots & \frac{\partial^2 L}{\partial \theta_p^2} \end{pmatrix} .$$

Example 2.8. Consider the function $L\colon \mathbb{R}^2 \to \mathbb{R}$ defined by $L(\theta_1, \theta_2) = \theta_1^2 + 3\theta_1\theta_2 + 2\theta_2^2$. We compute all four second-order partial derivatives,

$$\frac{\partial^2 L}{\partial \theta_1^2} = 2, \qquad \frac{\partial^2 L}{\partial \theta_2^2} = 4, \qquad \frac{\partial^2 L}{\partial \theta_1 \partial \theta_2} = 3, \qquad \frac{\partial^2 L}{\partial \theta_2 \partial \theta_1} = 3.$$

These are constant (independent of $\boldsymbol{\theta}$), so the Hessian is the same at every point,

$$\mathbf{H}_L(\boldsymbol{\theta}) = \begin{pmatrix} 2 & 3 \\ 3 & 4 \end{pmatrix}.$$

Note that the two mixed partials agree, illustrating the symmetry $\mathbf{H}_L = \mathbf{H}_L^T$.

Theorem 2.9 (Symmetry of the Hessian)**.** *Let $L\colon U \to \mathbb{R}$ be twice continuously differentiable on an open set $U \subseteq \mathbb{R}^p$. Then the Hessian is symmetric at every $\boldsymbol{\theta} \in U$:*

$$[\mathbf{H}_L(\boldsymbol{\theta})]_{ij} = [\mathbf{H}_L(\boldsymbol{\theta})]_{ji}, \qquad i, j = 1, \ldots, p.$$

Proof. Fix indices $i \neq j$. Since L is twice continuously differentiable on U, the mixed partial derivatives $\frac{\partial^2 L}{\partial \theta_i \partial \theta_j}$ and $\frac{\partial^2 L}{\partial \theta_j \partial \theta_i}$ both exist and are continuous throughout U. Clairaut's theorem asserts that continuity of the mixed partials is sufficient to guarantee their equality at every point, so $\frac{\partial^2 L}{\partial \theta_i \partial \theta_j}(\boldsymbol{\theta}) = \frac{\partial^2 L}{\partial \theta_j \partial \theta_i}(\boldsymbol{\theta})$ for all $\boldsymbol{\theta} \in U$. The diagonal entries satisfy $[\mathbf{H}_L]_{ii} = [\mathbf{H}_L]_{ii}$ trivially, so $\mathbf{H}_L(\boldsymbol{\theta}) = \mathbf{H}_L(\boldsymbol{\theta})^T$ everywhere on U. □

Example 2.10 (Hessian of a Quadratic in $\mathbb{R}^2$)**.** Continuing with $L(\boldsymbol{\theta}) = \theta_1^2 + 4\theta_2^2$, the Hessian is

$$\mathbf{H}_L(\boldsymbol{\theta}) = \begin{pmatrix} 2 & 0 \\ 0 & 8 \end{pmatrix}.$$

This matrix is constant (independent of $\boldsymbol{\theta}$), symmetric, and positive definite (eigenvalues 2 and 8 are both positive). The eigenvalues reveal that the curvature along the θ_2 axis is four times steeper than along θ_1, consistent with the elliptical level curves of this function.

Geometrically, the Hessian describes the local *curvature* of the loss landscape. It encodes how the gradient vector field changes in the neighborhood of a point $\boldsymbol{\theta}$.

2.3 Classifying Critical Points

The primary use of the Hessian in optimization is to classify critical points—those points where the gradient vanishes. In single-variable calculus, the sign of $f''(x)$ determines whether a critical point is a local minimum or maximum. The Hessian generalizes this test to higher dimensions.

The classification depends on the **definiteness** of the Hessian at the critical point $\boldsymbol{\theta}_c$, determined by the signs of its eigenvalues.

Theorem 2.11 (Second Derivative Test)**.** *Let $L\colon \mathbb{R}^p \to \mathbb{R}$ be twice continuously differentiable, and let $\boldsymbol{\theta}_c$ be a critical point with $\nabla L(\boldsymbol{\theta}_c) = \mathbf{0}$. Let $\mathbf{H}_L(\boldsymbol{\theta}_c)$ be the Hessian evaluated at this point.*

1. *If $\mathbf{H}_L(\boldsymbol{\theta}_c)$ is **positive definite** (all eigenvalues strictly positive), then $\boldsymbol{\theta}_c$ is a strict local minimum of L.*

2. *If $\mathbf{H}_L(\boldsymbol{\theta}_c)$ is **negative definite** (all eigenvalues strictly negative), then $\boldsymbol{\theta}_c$ is a strict local maximum of L.*

3. *If $\mathbf{H}_L(\boldsymbol{\theta}_c)$ is **indefinite** (both positive and negative eigenvalues), then $\boldsymbol{\theta}_c$ is a saddle point of L.*

*4. If $\mathbf{H}_L(\boldsymbol{\theta}_c)$ is **singular** (at least one zero eigenvalue), the test is inconclusive.*

Example 2.12 (Classifying Critical Points)**.** Consider the function $f(x_1, x_2) = x_1^2 - x_2^2$. The gradient is $\nabla f = [2x_1, -2x_2]^T$, which vanishes at the origin. The Hessian is

$$\mathbf{H}_f = \begin{pmatrix} 2 & 0 \\ 0 & -2 \end{pmatrix}.$$

The eigenvalues are 2 and -2. Since one is positive and one is negative, the Hessian is indefinite, and the origin is a saddle point. The surface curves upward along x_1 and downward along x_2.

A local minimum corresponds to the bottom of a bowl, a local maximum to the peak of a hill, and a saddle point to a location resembling a mountain pass—a minimum along one direction but a maximum along another. In deep learning, singular Hessians can produce large flat regions (plateaus) that significantly slow optimization.

The gradient, the Hessian, and optimization are connected through the multivariable Taylor expansion. Expanding $L(\boldsymbol{\theta})$ around a point $\boldsymbol{\theta}_k$, the gradient defines the best *linear* approximation (the first-order Taylor expansion):

$$L(\boldsymbol{\theta}) \approx L(\boldsymbol{\theta}_k) + \nabla L(\boldsymbol{\theta}_k)^T (\boldsymbol{\theta} - \boldsymbol{\theta}_k).$$

The Hessian defines the best *quadratic* approximation (the second-order Taylor expansion):

$$L(\boldsymbol{\theta}) \approx L(\boldsymbol{\theta}_k) + \nabla L(\boldsymbol{\theta}_k)^T (\boldsymbol{\theta} - \boldsymbol{\theta}_k) + \frac{1}{2}(\boldsymbol{\theta} - \boldsymbol{\theta}_k)^T \mathbf{H}_L(\boldsymbol{\theta}_k)(\boldsymbol{\theta} - \boldsymbol{\theta}_k).$$

This framework reveals a hierarchy of optimization algorithms. Gradient Descent operates on the linear approximation, making it a **first-order method**. More advanced techniques such as Newton's method use the quadratic approximation, making them **second-order methods**.

3 The Batch Gradient Descent Algorithm

Armed with the geometric intuition that the negative gradient points in the direction of steepest local descent, we now formally derive the gradient descent algorithm. The derivation follows directly from the first-order Taylor expansion of the loss function.

3.1 Derivation from First-Order Taylor Expansion

Our goal is to construct a sequence of parameter vectors $\{\boldsymbol{\theta}_0, \boldsymbol{\theta}_1, \boldsymbol{\theta}_2, \dots\}$ such that $L(\boldsymbol{\theta}_{k+1}) < L(\boldsymbol{\theta}_k)$, converging to a minimum $\boldsymbol{\theta}^*$.

Let us consider our position at step k, given by $\boldsymbol{\theta}_k$. We seek a new point

$$\boldsymbol{\theta}_{k+1} = \boldsymbol{\theta}_k + \Delta\boldsymbol{\theta},$$

where $\Delta\boldsymbol{\theta}$ is a small step vector, that reduces the loss. For a sufficiently small step, we approximate the loss at the new point via a first-order Taylor expansion:

$$L(\boldsymbol{\theta}_{k+1}) = L(\boldsymbol{\theta}_k + \Delta\boldsymbol{\theta}) \approx L(\boldsymbol{\theta}_k) + \nabla L(\boldsymbol{\theta}_k)^T \Delta\boldsymbol{\theta}.$$

To ensure $L(\boldsymbol{\theta}_{k+1}) < L(\boldsymbol{\theta}_k)$, we must choose $\Delta\boldsymbol{\theta}$ so that $\nabla L(\boldsymbol{\theta}_k)^T \Delta\boldsymbol{\theta} < 0$. To make the most progress, we choose $\Delta\boldsymbol{\theta}$ to make this term as negative as possible.

By the Cauchy–Schwarz inequality, $|\mathbf{u}^T\mathbf{v}| \le \|\mathbf{u}\|_2 \|\mathbf{v}\|_2$, which implies $\mathbf{u}^T\mathbf{v} \ge -\|\mathbf{u}\|_2\|\mathbf{v}\|_2$. The minimum is achieved when $\mathbf{v}$ is antiparallel to $\mathbf{u}$. Setting $\mathbf{u} = \nabla L(\boldsymbol{\theta}_k)$ and $\mathbf{v} = \Delta\boldsymbol{\theta}$, we conclude that $\Delta\boldsymbol{\theta}$ must be antiparallel to $\nabla L(\boldsymbol{\theta}_k)$.

We define our step direction as the negative gradient and introduce a positive scalar hyperparameter $\eta > 0$, called the **learning rate**, to control the step magnitude:

$$\Delta\boldsymbol{\theta} = -\eta \nabla L(\boldsymbol{\theta}_k).$$

Substituting into the Taylor approximation gives

$$\begin{aligned} L(\boldsymbol{\theta}_{k+1}) &\approx L(\boldsymbol{\theta}_k) + \nabla L(\boldsymbol{\theta}_k)^T(-\eta \nabla L(\boldsymbol{\theta}_k)) \\ &= L(\boldsymbol{\theta}_k) - \eta \|\nabla L(\boldsymbol{\theta}_k)\|_2^2. \end{aligned}$$

Since $\eta > 0$ and $\|\nabla L(\boldsymbol{\theta}_k)\|_2^2 \geq 0$, the term $-\eta\|\nabla L(\boldsymbol{\theta}_k)\|_2^2$ is non-positive. As long as $\nabla L(\boldsymbol{\theta}_k) \neq \mathbf{0}$, this update guarantees that, for sufficiently small η, the loss decreases.

This leads to the formal definition of the batch gradient descent algorithm, so named because the gradient $\nabla L(\boldsymbol{\theta}_k)$ is computed using the entire batch of training data.

```
Algorithm 1: Batch Gradient Descent
  Input: Initial parameters $\boldsymbol{\theta}_0 \in \mathbb{R}^p$, learning rate $\eta > 0$, tolerance $\epsilon > 0$
  Result: Optimized parameters $\boldsymbol{\theta}^*$
1 $k \leftarrow 0$;
2 while $\|\nabla L(\boldsymbol{\theta}_k)\|_2 > \epsilon$ do
3     Compute the gradient over all $n$ samples: $\nabla L(\boldsymbol{\theta}_k) \leftarrow \frac{1}{n}\sum_{i=1}^{n} \nabla L_i(\boldsymbol{\theta}_k)$;
4     Update parameters: $\boldsymbol{\theta}_{k+1} \leftarrow \boldsymbol{\theta}_k - \eta \nabla L(\boldsymbol{\theta}_k)$;
5     $k \leftarrow k + 1$;
6 end
7 return $\boldsymbol{\theta}_k$
```

The complete procedure is summarized in Algorithm 1. At each iteration, the algorithm computes the exact gradient over the entire training set, takes a step proportional to the negative gradient, and repeats until the gradient norm falls below a prescribed tolerance ϵ. The simplicity of this loop belies its power: as we shall prove in Section 4, for convex loss functions this procedure is guaranteed to converge to the global optimum.

3.2 Deriving the Gradient for OLS

We now apply gradient descent to OLS. The update rule requires the gradient $\nabla L(\boldsymbol{\beta})$, which we derive formally.

Recall the OLS loss function:

$$L(\boldsymbol{\beta}) = \frac{1}{n}\|\mathbf{X}\boldsymbol{\beta} - \mathbf{y}\|_2^2 = \frac{1}{n}(\mathbf{X}\boldsymbol{\beta} - \mathbf{y})^T(\mathbf{X}\boldsymbol{\beta} - \mathbf{y}).$$

Expanding the quadratic form gives

$$\begin{aligned} L(\boldsymbol{\beta}) &= \frac{1}{n}((\mathbf{X}\boldsymbol{\beta})^T - \mathbf{y}^T)(\mathbf{X}\boldsymbol{\beta} - \mathbf{y}) \\ &= \frac{1}{n}(\boldsymbol{\beta}^T\mathbf{X}^T - \mathbf{y}^T)(\mathbf{X}\boldsymbol{\beta} - \mathbf{y}) \\ &= \frac{1}{n}(\boldsymbol{\beta}^T\mathbf{X}^T\mathbf{X}\boldsymbol{\beta} - \boldsymbol{\beta}^T\mathbf{X}^T\mathbf{y} - \mathbf{y}^T\mathbf{X}\boldsymbol{\beta} + \mathbf{y}^T\mathbf{y}). \end{aligned}$$

We note that $(\boldsymbol{\beta}^T\mathbf{X}^T\mathbf{y})^T = \mathbf{y}^T\mathbf{X}\boldsymbol{\beta}$. Since a scalar equals its own transpose, these two middle terms are equal, giving

$$L(\boldsymbol{\beta}) = \frac{1}{n}(\boldsymbol{\beta}^T(\mathbf{X}^T\mathbf{X})\boldsymbol{\beta} - 2\boldsymbol{\beta}^T(\mathbf{X}^T\mathbf{y}) + \mathbf{y}^T\mathbf{y}). \tag{1}$$

This is a standard quadratic form $L(\boldsymbol{\beta}) = \frac{1}{n}(\boldsymbol{\beta}^T\mathbf{A}\boldsymbol{\beta} - 2\boldsymbol{\beta}^T\mathbf{c} + k)$, where $\mathbf{A} = \mathbf{X}^T\mathbf{X}$, $\mathbf{c} = \mathbf{X}^T\mathbf{y}$, and $k = \mathbf{y}^T\mathbf{y}$.

To find the gradient, we must differentiate this scalar function with respect to the vector $\boldsymbol{\beta}$.

Definition 3.1 (Gradient Notation)**.** The gradient of a scalar function $f\colon \mathbb{R}^p \to \mathbb{R}$ with respect to a column vector $\boldsymbol{\beta} \in \mathbb{R}^p$ is itself a column vector of partial derivatives:

$$\nabla L(\boldsymbol{\beta}) \equiv \nabla_{\boldsymbol{\beta}} L(\boldsymbol{\beta}) = \begin{bmatrix} \frac{\partial L}{\partial \beta_1} \\ \vdots \\ \frac{\partial L}{\partial \beta_p} \end{bmatrix}.$$

Theorem 3.2 (Matrix Calculus Identities)**.** *Under the denominator layout convention, the following identities hold for $\boldsymbol{\beta} \in \mathbb{R}^p$, $\mathbf{c} \in \mathbb{R}^p$, and $\mathbf{A} \in \mathbb{R}^{p\times p}$:*

1. $\nabla_{\boldsymbol{\beta}}(\mathbf{c}^T\boldsymbol{\beta}) = \nabla_{\boldsymbol{\beta}}(\boldsymbol{\beta}^T\mathbf{c}) = \mathbf{c}$.
2. $\nabla_{\boldsymbol{\beta}}(\boldsymbol{\beta}^T\mathbf{A}\boldsymbol{\beta}) = (\mathbf{A} + \mathbf{A}^T)\boldsymbol{\beta}$.

In particular, if $\mathbf{A}$ is symmetric, identity (2) simplifies to $\nabla_{\boldsymbol{\beta}}(\boldsymbol{\beta}^T\mathbf{A}\boldsymbol{\beta}) = 2\mathbf{A}\boldsymbol{\beta}$.

Proof. For identity (1), the inner product $\mathbf{c}^T\boldsymbol{\beta} = \boldsymbol{\beta}^T\mathbf{c} = \sum_{i=1}^p c_i\beta_i$ is linear in $\boldsymbol{\beta}$. Its j-th partial derivative is

$$\frac{\partial}{\partial \beta_j}\sum_{i=1}^p c_i\beta_i = c_j,$$

so the gradient vector is $\mathbf{c}$.

For identity (2), write

$$\boldsymbol{\beta}^T\mathbf{A}\boldsymbol{\beta} = \sum_{i=1}^p\sum_{k=1}^p A_{ik}\beta_i\beta_k.$$

Differentiating with respect to β_j via the product rule gives

$$\frac{\partial}{\partial \beta_j}(\boldsymbol{\beta}^T\mathbf{A}\boldsymbol{\beta}) = \sum_{k=1}^p A_{jk}\beta_k + \sum_{i=1}^p A_{ij}\beta_i = (\mathbf{A}\boldsymbol{\beta})_j + (\mathbf{A}^T\boldsymbol{\beta})_j.$$

Collecting over all j yields $\nabla_{\boldsymbol{\beta}}(\boldsymbol{\beta}^T\mathbf{A}\boldsymbol{\beta}) = (\mathbf{A}+\mathbf{A}^T)\boldsymbol{\beta}$. When $\mathbf{A} = \mathbf{A}^T$, this reduces immediately to $2\mathbf{A}\boldsymbol{\beta}$. □

Example 3.3. Let $f(\boldsymbol{\beta}) = \boldsymbol{\beta}^T\mathbf{A}\boldsymbol{\beta} - 2\boldsymbol{\beta}^T\mathbf{c}$ with $\mathbf{A} = \begin{pmatrix} 3 & 1 \\ 1 & 2 \end{pmatrix}$ and $\mathbf{c} = \begin{pmatrix} 1 \\ 4 \end{pmatrix}$. Since $\mathbf{A}$ is symmetric, identity (2) gives $\nabla_{\boldsymbol{\beta}}(\boldsymbol{\beta}^T\mathbf{A}\boldsymbol{\beta}) = 2\mathbf{A}\boldsymbol{\beta}$, and identity (1) gives $\nabla_{\boldsymbol{\beta}}(\boldsymbol{\beta}^T\mathbf{c}) = \mathbf{c}$. Therefore,

$$\nabla f(\boldsymbol{\beta}) = 2\mathbf{A}\boldsymbol{\beta} - 2\mathbf{c} = 2\begin{pmatrix} 3 & 1 \\ 1 & 2 \end{pmatrix}\boldsymbol{\beta} - 2\begin{pmatrix} 1 \\ 4 \end{pmatrix}.$$

Setting $\nabla f = \mathbf{0}$ yields $\boldsymbol{\beta}^* = \mathbf{A}^{-1}\mathbf{c} = \frac{1}{5}\begin{pmatrix} 2 & -1 \\ -1 & 3 \end{pmatrix}\begin{pmatrix} 1 \\ 4 \end{pmatrix} = \frac{1}{5}\begin{pmatrix} -2 \\ 11 \end{pmatrix}$.

We apply Theorem 3.2 to the expanded loss (1):

$$L(\boldsymbol{\beta}) = \frac{1}{n}\underbrace{(\boldsymbol{\beta}^T(\mathbf{X}^T\mathbf{X})\boldsymbol{\beta})}_{\text{Form } \boldsymbol{\beta}^T\mathbf{A}\boldsymbol{\beta}} - \frac{2}{n}\underbrace{(\boldsymbol{\beta}^T(\mathbf{X}^T\mathbf{y}))}_{\text{Form } \boldsymbol{\beta}^T\mathbf{c}} + \underbrace{\frac{1}{n}(\mathbf{y}^T\mathbf{y})}_{\text{Constant}}.$$

Differentiating term by term gives

$$\begin{aligned}\nabla L(\boldsymbol{\beta}) &= \nabla_{\boldsymbol{\beta}}\left[\frac{1}{n}(\boldsymbol{\beta}^T(\mathbf{X}^T\mathbf{X})\boldsymbol{\beta}) - \frac{2}{n}(\boldsymbol{\beta}^T(\mathbf{X}^T\mathbf{y})) + \frac{1}{n}(\mathbf{y}^T\mathbf{y})\right] \\ &= \frac{1}{n}\nabla_{\boldsymbol{\beta}}(\boldsymbol{\beta}^T(\mathbf{X}^T\mathbf{X})\boldsymbol{\beta}) - \frac{2}{n}\nabla_{\boldsymbol{\beta}}(\boldsymbol{\beta}^T(\mathbf{X}^T\mathbf{y})) + \nabla_{\boldsymbol{\beta}}\left(\frac{1}{n}\mathbf{y}^T\mathbf{y}\right). \qquad (2)\end{aligned}$$

We analyze each term in (2):

- **Term 1:** $\mathbf{A} = \mathbf{X}^T\mathbf{X}$ is symmetric, so identity (2) gives $\nabla_{\boldsymbol{\beta}}(\boldsymbol{\beta}^T(\mathbf{X}^T\mathbf{X})\boldsymbol{\beta}) = 2(\mathbf{X}^T\mathbf{X})\boldsymbol{\beta}$.
- **Term 2:** This has the form $\boldsymbol{\beta}^T\mathbf{c}$ with $\mathbf{c} = \mathbf{X}^T\mathbf{y}$, so identity (1) gives $\nabla_{\boldsymbol{\beta}}(\boldsymbol{\beta}^T(\mathbf{X}^T\mathbf{y})) = \mathbf{X}^T\mathbf{y}$.
- **Term 3:** This is constant with respect to $\boldsymbol{\beta}$; its gradient is the zero vector.

Substituting gives

$$\begin{aligned}\nabla L(\boldsymbol{\beta}) &= \frac{1}{n}(2(\mathbf{X}^T\mathbf{X})\boldsymbol{\beta}) - \frac{2}{n}(\mathbf{X}^T\mathbf{y}) + \mathbf{0} \\ &= \frac{2}{n}(\mathbf{X}^T\mathbf{X}\boldsymbol{\beta} - \mathbf{X}^T\mathbf{y}) \\ &= \frac{2}{n}\mathbf{X}^T(\mathbf{X}\boldsymbol{\beta} - \mathbf{y}).\end{aligned}$$

The batch gradient descent update rule for OLS is therefore as follows:

$$\boldsymbol{\beta}_{k+1} = \boldsymbol{\beta}_k - \eta\nabla L(\boldsymbol{\beta}_k) = \boldsymbol{\beta}_k - \frac{2\eta}{n}\mathbf{X}^T(\mathbf{X}\boldsymbol{\beta}_k - \mathbf{y}).$$

3.3 The Critical Role of the Learning Rate

The learning rate η is the single most important hyperparameter in training machine learning models. Its value dictates the behavior of the optimization process.

The learning rate measures how much we *trust* the local linear approximation provided by the gradient. The Taylor expansion is only accurate in a small neighborhood around $\boldsymbol{\theta}_k$, and the size of this neighborhood is related to the local curvature (the Hessian).

- **If η is too small:** The algorithm takes excessively small steps. While the loss reliably decreases at each iteration, convergence is impractically slow.
- **If η is too large:** The step overshoots the minimum, landing in a region of higher loss. This can cause the loss to oscillate or diverge to infinity.
- **If η is well-chosen:** The learning rate balances speed with stability, taking steps large enough for meaningful progress yet small enough to remain in a region where the gradient is a reliable guide.

Example 3.4 (Gradient Descent for OLS)**.** We apply the algorithm to ordinary least squares.

1. **Loss Function:** $L(\boldsymbol{\beta}) = \frac{1}{n}\|\mathbf{X}\boldsymbol{\beta} - \mathbf{y}\|_2^2 = \frac{1}{n}(\mathbf{y} - \mathbf{X}\boldsymbol{\beta})^T(\mathbf{y} - \mathbf{X}\boldsymbol{\beta})$.
2. **Gradient Calculation:** We expand and apply matrix calculus identities (with $\mathbf{X}^T\mathbf{X}$ symmetric):

$$\begin{aligned}\nabla L(\boldsymbol{\beta}) &= \nabla_{\boldsymbol{\beta}}\left[\frac{1}{n}\left(\mathbf{y}^T\mathbf{y} - 2\boldsymbol{\beta}^T(\mathbf{X}^T\mathbf{y}) + \boldsymbol{\beta}^T(\mathbf{X}^T\mathbf{X})\boldsymbol{\beta}\right)\right] \\ &= \frac{1}{n}\left[-2\mathbf{X}^T\mathbf{y} + 2(\mathbf{X}^T\mathbf{X})\boldsymbol{\beta}\right] \\ &= \frac{2}{n}\mathbf{X}^T(\mathbf{X}\boldsymbol{\beta} - \mathbf{y}).\end{aligned}$$

The term $(\mathbf{X}\boldsymbol{\beta} - \mathbf{y})$ is the residual vector, and $\mathbf{X}^T$ projects these residuals back onto the parameter space to determine the update direction.

3. **Update Rule:** The batch gradient descent update for OLS is as follows:

$$\boldsymbol{\beta}_{k+1} = \boldsymbol{\beta}_k - \frac{2\eta}{n}\mathbf{X}^T(\mathbf{X}\boldsymbol{\beta}_k - \mathbf{y}).$$

   At each iteration, we compute the predictions $\mathbf{X}\boldsymbol{\beta}_k$, find the residuals, compute the gradient, and update $\boldsymbol{\beta}$. We repeat until convergence.

Example 3.5 (Batch Gradient Descent: Convergence to the Optimum)**.** We trace batch gradient descent on a concrete dataset to observe its convergence. Consider the model $y = \beta x$ (regression through the origin, $p = 1$) with the following $n = 4$ data points:

i	1	2	3	4
x_i	1	2	3	4
y_i	2	4	5	8

The closed-form solution is $\beta^* = \frac{\sum x_i y_i}{\sum x_i^2} = \frac{57}{30} = 1.9$, with optimal loss $L(\beta^*) = 0.175$. The MSE loss is $L(\beta) = \frac{1}{4}\sum_{i=1}^{4}(\beta x_i - y_i)^2 = 7.5\beta^2 - 28.5\beta + 27.25$, and its gradient simplifies to

$$\nabla L(\beta) = \frac{1}{2}\sum_{i=1}^{4} x_i(\beta x_i - y_i) = 15\beta - 28.5.$$

We initialize $\beta_0 = 0$ and set $\eta = 0.05$. Each iteration applies $\beta_{k+1} = \beta_k - 0.05 \cdot \nabla L(\beta_k)$:

k	β_k	$\nabla L(\beta_k)$	$L(\beta_k)$
0	0.0000	−28.500	27.2500
1	1.4250	−7.125	1.8672
2	1.7813	−1.781	0.2808
3	1.8703	−0.445	0.1816
4	1.8926	−0.111	0.1754
5	1.8981	−0.028	0.1750
6	1.8995	−0.007	0.1750
7	1.8999	−0.002	0.1750
8	1.9000	−0.000	0.1750

The loss decreases *monotonically* at every iteration—a hallmark of batch GD with a well-chosen learning rate. By iteration 3, β has already reached 1.870, capturing over 98% of the distance to β^*. By iteration 8, $\beta \approx 1.900$ and $L \approx 0.175$, essentially converged.

Key strength: reliability. Every step is guaranteed to reduce the loss because the gradient is computed exactly from all n samples. The convergence path is perfectly smooth with no oscillation. This deterministic behavior makes batch GD easy to analyze theoretically and to debug in practice. Figure 1 illustrates this smooth, monotonic decrease of the loss over eight iterations toward the optimum $L(\beta^*) = 0.175$.

Example 3.6 (Batch GD: Instability and Computational Cost)**.** We illustrate the two principal weaknesses of batch GD using the same dataset as the previous example: $n = 4$, $\mathbf{x} = [1, 2, 3, 4]^\top$, $\mathbf{y} = [2, 4, 5, 8]^\top$, with loss $L(\beta) = 7.5\beta^2 - 28.5\beta + 27.25$ and gradient $\nabla L(\beta) = 15\beta - 28.5$.

Part 1: Instability from an Excessive Learning Rate. The Hessian of the OLS loss is constant: $H_L = \frac{2}{n}\mathbf{X}^\top\mathbf{X} = 15$. Batch GD with a fixed learning rate η converges if and only if $|1 - \eta H_L| < 1$, which requires

$$\eta < \frac{2}{H_L} = \frac{2}{15} \approx 0.133.$$

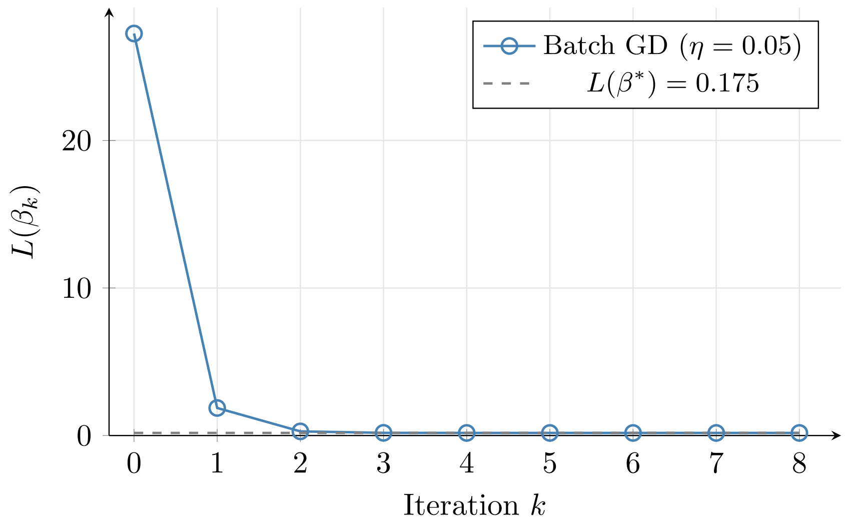

Figure 1: Batch Gradient Descent convergence ($\eta = 0.05$). The loss decreases smoothly and monotonically toward $L(\beta^*) = 0.175$ (dashed line). The rapid initial drop followed by gradual refinement is characteristic of gradient descent on a convex quadratic.

The previous example used $\eta = 0.05$, giving a contraction factor $|1 - 0.05 \times 15| = 0.25$. We now set $\eta = 0.15$, which violates the convergence condition: the contraction factor becomes $|1 - 0.15 \times 15| = 1.25 > 1$. Starting from $\beta_0 = 0$, the iterates are as follows:

$$\begin{aligned}
\beta_1 &= 0 - 0.15(-28.5) = 4.275,\\
\beta_2 &= 4.275 - 0.15(35.625) = -1.069,\\
\beta_3 &= -1.069 - 0.15(-44.535) = 5.611,\\
\beta_4 &= 5.611 - 0.15(55.665) = -2.739.
\end{aligned}$$

The corresponding gradient and loss at each iterate are shown below.

Iteration k	β_k	$\nabla L(\beta_k)$	$L(\beta_k)$
0	0.000	−28.500	27.250
1	4.275	35.625	42.482
2	−1.069	−44.535	66.289
3	5.611	55.665	103.459
4	−2.739	−69.585	161.577

The loss grows by a factor of approximately 1.56 at each step and β alternates in sign, indicating geometric divergence. The update overshoots the minimum, crosses to the other side, and overshoots again with greater magnitude. No amount of additional iterations recovers convergence once the learning rate violates the bound.

Part 2: Computational Cost at Scale. Even when η is chosen correctly, batch GD requires computing the full gradient $\nabla L(\boldsymbol{\beta}) = \frac{2}{n}\mathbf{X}^\top(\mathbf{X}\boldsymbol{\beta} - \mathbf{y})$ at every iteration. This matrix-vector product costs $O(np)$ floating-point operations. In a realistic setting with $n = 10^6$ observations and $p = 10^3$ features, the cost per gradient evaluation is

$$n \times p = 10^6 \times 10^3 = 10^9 \text{ operations.}$$

Convergence typically requires on the order of 10^3 iterations, giving a total cost of

$$10^3 \times 10^9 = 10^{12} \text{ operations per training run.}$$

A standard desktop CPU sustains approximately 10^9 floating-point operations per second, so a single training run requires roughly 10^3 seconds, or about 17 minutes. A modest hyperparameter search over ten values of η multiplies this to approximately 170 minutes of wall-clock time—for a single model.

Key weakness. Batch GD computes an exact gradient at every step, but exactness is expensive: every parameter update requires a full pass over the entire dataset, regardless of how well the gradient is estimated. The variants introduced in Section 4 address this cost directly by trading gradient exactness for computational efficiency.

4 Convexity and Convergence Guarantees

We have derived an algorithm guaranteed to take steps *downhill* on the loss landscape. However, this local guarantee does not promise that we will reach the lowest point on the entire surface. A non-convex landscape can contain many local minima—small valleys that are not the global minimum—in which gradient descent can become trapped.

The OLS problem is special: its loss function is *convex.* This property guarantees that the landscape is a single bowl with no misleading local minima. Convex optimization provides the guarantee that gradient descent will find the single best solution.

4.1 Formal Definitions of Convexity

Definition 4.1 (Convex Set)**.** A set $C \subseteq \mathbb{R}^p$ is **convex** if for any two points $\mathbf{x}, \mathbf{y} \in C$ and any scalar $\lambda \in [0,1]$, the convex combination $\lambda\mathbf{x} + (1-\lambda)\mathbf{y}$ is also in C:

$$\forall \mathbf{x}, \mathbf{y} \in C,\ \forall \lambda \in [0,1] \implies \lambda\mathbf{x} + (1-\lambda)\mathbf{y} \in C.$$

Geometrically, the line segment connecting any two points in C lies entirely within C.

Example 4.2 (Convex and Non-Convex Sets)**.** The Euclidean ball $\{\mathbf{x} \in \mathbb{R}^p : \|\mathbf{x}\|_2 \leq r\}$ is convex: for any $\mathbf{x}, \mathbf{y}$ with $\|\mathbf{x}\|_2 \leq r$ and $\|\mathbf{y}\|_2 \leq r$, the triangle inequality gives

$$\|\lambda\mathbf{x} + (1-\lambda)\mathbf{y}\|_2 \leq \lambda\|\mathbf{x}\|_2 + (1-\lambda)\|\mathbf{y}\|_2 \leq r.$$

The set $\{(x_1, x_2) : x_1^2 + x_2^2 \geq 1\}$ (the exterior of the unit circle) is not convex: the points $(1,0)$ and $(-1,0)$ are both in the set, but their midpoint $(0,0)$ is not.

Definition 4.3 (Convex Function)**.** A function $f\colon C \to \mathbb{R}$ defined on a convex set C is **convex** if for any $\mathbf{x}, \mathbf{y} \in C$ and any $\lambda \in [0,1]$:

$$f(\lambda\mathbf{x} + (1-\lambda)\mathbf{y}) \leq \lambda f(\mathbf{x}) + (1-\lambda) f(\mathbf{y}).$$

Geometrically, the chord connecting $(\mathbf{x}, f(\mathbf{x}))$ and $(\mathbf{y}, f(\mathbf{y}))$ lies on or above the graph of f.

Example 4.4 (Convex and Non-Convex Functions)**.** The function $f(x) = x^2$ is convex on $\mathbb{R}$: for any $a, b \in \mathbb{R}$ and $\lambda \in [0,1]$, we have $(\lambda a + (1-\lambda)b)^2 \leq \lambda a^2 + (1-\lambda)b^2$, which follows from the identity $\lambda(1-\lambda)(a-b)^2 \geq 0$. The function $g(x) = \sin(x)$ is not convex on $\mathbb{R}$: for example, $g(0) = 0$ and $g(\pi) = 0$, but $g(\pi/2) = 1 > 0 = \frac{1}{2}g(0) + \frac{1}{2}g(\pi)$.

4.2 First and Second-Order Conditions for Convexity

For differentiable functions, convexity can be characterized by practical conditions involving the gradient and the Hessian.

Lemma 4.5 (First-Order Condition for Convexity). *Let $f\colon C \to \mathbb{R}$ be continuously differentiable on an open convex set C. Then f is convex if and only if for all $\mathbf{x}, \mathbf{y} \in C$:*

$$f(\mathbf{y}) \geq f(\mathbf{x}) + \nabla f(\mathbf{x})^T(\mathbf{y} - \mathbf{x}).$$

The first-order Taylor approximation at any point $\mathbf{x}$ (the tangent hyperplane) serves as a global underestimator of f.

Proof. We prove both directions of the equivalence.

($\Rightarrow$) Suppose f is convex. Fix $\mathbf{x}, \mathbf{y} \in C$ and let $\lambda \in (0, 1]$. By convexity,

$$f(\mathbf{x} + \lambda(\mathbf{y} - \mathbf{x})) = f(\lambda\mathbf{y} + (1-\lambda)\mathbf{x}) \leq \lambda f(\mathbf{y}) + (1-\lambda)f(\mathbf{x}).$$

Rearranging and dividing by $\lambda > 0$ gives

$$\frac{f(\mathbf{x} + \lambda(\mathbf{y} - \mathbf{x})) - f(\mathbf{x})}{\lambda} \leq f(\mathbf{y}) - f(\mathbf{x}).$$

Taking $\lambda \to 0^+$, the left side converges to the directional derivative $\nabla f(\mathbf{x})^T(\mathbf{y} - \mathbf{x})$, yielding $f(\mathbf{y}) \geq f(\mathbf{x}) + \nabla f(\mathbf{x})^T(\mathbf{y} - \mathbf{x})$.

($\Leftarrow$) Suppose the tangent hyperplane condition holds everywhere. Let $\mathbf{x}, \mathbf{y} \in C$, $\lambda \in [0, 1]$, and set $\mathbf{z} = \lambda\mathbf{x} + (1-\lambda)\mathbf{y} \in C$. Applying the condition at $\mathbf{z}$ with each of $\mathbf{x}$ and $\mathbf{y}$ as the second argument gives $f(\mathbf{x}) \geq f(\mathbf{z}) + \nabla f(\mathbf{z})^T(\mathbf{x} - \mathbf{z})$ and $f(\mathbf{y}) \geq f(\mathbf{z}) + \nabla f(\mathbf{z})^T(\mathbf{y} - \mathbf{z})$. Multiplying the first inequality by λ and the second by $(1-\lambda)$ and adding,

$$\lambda f(\mathbf{x}) + (1-\lambda)f(\mathbf{y}) \geq f(\mathbf{z}) + \nabla f(\mathbf{z})^T\big(\lambda(\mathbf{x} - \mathbf{z}) + (1-\lambda)(\mathbf{y} - \mathbf{z})\big).$$

Since $\lambda(\mathbf{x} - \mathbf{z}) + (1-\lambda)(\mathbf{y} - \mathbf{z}) = \lambda\mathbf{x} + (1-\lambda)\mathbf{y} - \mathbf{z} = \mathbf{0}$, the gradient term vanishes, and we obtain $\lambda f(\mathbf{x}) + (1-\lambda)f(\mathbf{y}) \geq f(\mathbf{z}) = f(\lambda\mathbf{x} + (1-\lambda)\mathbf{y})$, which is the definition of convexity. $\square$

Lemma 4.6 (Second-Order Condition for Convexity). *Let $f\colon C \to \mathbb{R}$ be twice continuously differentiable on an open convex set C. Then f is convex if and only if its Hessian is positive semidefinite everywhere:*

$$\mathbf{H}_f(\mathbf{x}) \succeq \mathbf{0} \quad \forall \mathbf{x} \in C.$$

This means f has non-negative curvature in all directions at every point. For a univariate function, this reduces to $f''(x) \geq 0$.

Proof. We prove both directions of the equivalence.

($\Rightarrow$) Suppose f is convex. Fix $\mathbf{x} \in C$ and a direction $\mathbf{v} \in \mathbb{R}^n$, and let $t > 0$ be small enough that $\mathbf{x} + t\mathbf{v} \in C$. By Lemma 4.5, convexity implies $f(\mathbf{x} + t\mathbf{v}) \geq f(\mathbf{x}) + t\nabla f(\mathbf{x})^T\mathbf{v}$. The second-order Taylor expansion gives

$$f(\mathbf{x} + t\mathbf{v}) = f(\mathbf{x}) + t\nabla f(\mathbf{x})^T\mathbf{v} + \tfrac{t^2}{2}\mathbf{v}^T\mathbf{H}_f(\mathbf{x})\mathbf{v} + o(t^2).$$

Subtracting the first-order lower bound from both sides and dividing by $t^2/2 > 0$ yields

$$\mathbf{v}^T\mathbf{H}_f(\mathbf{x})\mathbf{v} + \frac{o(t^2)}{t^2/2} \geq 0.$$

Taking $t \to 0^+$, the remainder term vanishes, giving $\mathbf{v}^T\mathbf{H}_f(\mathbf{x})\mathbf{v} \geq 0$ for all $\mathbf{v} \in \mathbb{R}^n$, i.e. $\mathbf{H}_f(\mathbf{x}) \succeq \mathbf{0}$.

($\Leftarrow$) Suppose $\mathbf{H}_f(\mathbf{x}) \succeq \mathbf{0}$ for all $\mathbf{x} \in C$. Fix $\mathbf{x}, \mathbf{y} \in C$. By the second-order Taylor expansion with integral remainder,

$$f(\mathbf{y}) = f(\mathbf{x}) + \nabla f(\mathbf{x})^T(\mathbf{y} - \mathbf{x}) + \int_0^1 (1-t)(\mathbf{y} - \mathbf{x})^T\mathbf{H}_f\big(\mathbf{x} + t(\mathbf{y} - \mathbf{x})\big)(\mathbf{y} - \mathbf{x})\,\mathrm{d}t.$$

Since $\mathbf{H}_f \succeq \mathbf{0}$ everywhere on C, the integrand $(1-t)(\mathbf{y} - \mathbf{x})^T\mathbf{H}_f(\cdots)(\mathbf{y} - \mathbf{x})$ is non-negative for all $t \in [0, 1]$. Therefore the integral is non-negative, giving $f(\mathbf{y}) \geq f(\mathbf{x}) + \nabla f(\mathbf{x})^T(\mathbf{y} - \mathbf{x})$. Since this tangent hyperplane inequality holds for all $\mathbf{x}, \mathbf{y} \in C$, Lemma 4.5 implies that f is convex. $\square$

4.3 The Cornerstone Theorem of Optimization

The profound implication of convexity for optimization is captured in the following theorem.

Theorem 4.7 (Global Optimality for Convex Functions). *Let $f\colon C \to \mathbb{R}$ be a convex function on a convex set C. If $\mathbf{x}^* \in C$ is a local minimum of f, then $\mathbf{x}^*$ is also a global minimum of f.*

Proof. Let $\mathbf{x}^*$ be a local minimum. Then there exists $\epsilon > 0$ such that $f(\mathbf{z}) \geq f(\mathbf{x}^*)$ for all $\mathbf{z} \in C$ with $\|\mathbf{z} - \mathbf{x}^*\|_2 < \epsilon$.

Assume for contradiction that $\mathbf{x}^*$ is not a global minimum. Then there exists $\mathbf{y} \in C$ with $f(\mathbf{y}) < f(\mathbf{x}^*)$.

Since C is convex, the point $\mathbf{z}_\lambda = \lambda\mathbf{y} + (1-\lambda)\mathbf{x}^*$ lies in C for all $\lambda \in (0, 1]$. By convexity of f:

$$f(\mathbf{z}_\lambda) = f(\lambda\mathbf{y} + (1-\lambda)\mathbf{x}^*) \leq \lambda f(\mathbf{y}) + (1-\lambda)f(\mathbf{x}^*).$$

Since $f(\mathbf{y}) < f(\mathbf{x}^*)$:

$$f(\mathbf{z}_\lambda) < \lambda f(\mathbf{x}^*) + (1-\lambda)f(\mathbf{x}^*) = f(\mathbf{x}^*).$$

The distance $\|\mathbf{z}_\lambda - \mathbf{x}^*\|_2 = \lambda\|\mathbf{y} - \mathbf{x}^*\|_2$ can be made arbitrarily small by choosing $\lambda < \epsilon/\|\mathbf{y} - \mathbf{x}^*\|_2$. For such λ, we have $\|\mathbf{z}_\lambda - \mathbf{x}^*\|_2 < \epsilon$.

This is a contradiction: $\mathbf{z}_\lambda$ lies within the ϵ-neighborhood of $\mathbf{x}^*$ yet satisfies $f(\mathbf{z}_\lambda) < f(\mathbf{x}^*)$, violating the local minimality of $\mathbf{x}^*$. Therefore, $\mathbf{x}^*$ is a global minimum. □

We now prove that the OLS loss function is convex, guaranteeing that gradient descent converges to the unique global minimum.

Theorem 4.8 (Convexity of OLS Loss). *The OLS loss function $L(\boldsymbol{\beta}) = \frac{1}{n}\|\mathbf{X}\boldsymbol{\beta} - \mathbf{y}\|_2^2$ is convex.*

Proof. We compute the Hessian and show it is positive semidefinite. The gradient of the OLS loss is

$$\nabla L(\boldsymbol{\beta}) = \frac{2}{n}\mathbf{X}^T(\mathbf{X}\boldsymbol{\beta} - \mathbf{y}) = \frac{2}{n}(\mathbf{X}^T\mathbf{X}\boldsymbol{\beta} - \mathbf{X}^T\mathbf{y}).$$

Differentiating once more gives the Hessian,

$$\mathbf{H}_L(\boldsymbol{\beta}) = \nabla_{\boldsymbol{\beta}^T}\left[\nabla L(\boldsymbol{\beta})\right] = \frac{2}{n}\mathbf{X}^T\mathbf{X}.$$

The Hessian is constant—it does not depend on $\boldsymbol{\beta}$, so the curvature of the OLS loss landscape is uniform. To verify positive semidefiniteness, let $\mathbf{z} \in \mathbb{R}^p$ be arbitrary. Then

$$\begin{aligned}
\mathbf{z}^T\mathbf{H}_L(\boldsymbol{\beta})\mathbf{z} &= \frac{2}{n}\mathbf{z}^T\mathbf{X}^T\mathbf{X}\mathbf{z} \\
&= \frac{2}{n}(\mathbf{X}\mathbf{z})^T(\mathbf{X}\mathbf{z}) \\
&= \frac{2}{n}\|\mathbf{X}\mathbf{z}\|_2^2 \geq 0.
\end{aligned}$$

Hence the Hessian is positive semidefinite everywhere, and by the second-order condition $L(\boldsymbol{\beta})$ is convex. If the columns of $\mathbf{X}$ are linearly independent ($\mathbf{X}^T\mathbf{X}$ is invertible), then $\|\mathbf{X}\mathbf{z}\|_2^2 > 0$ for all $\mathbf{z} \neq \mathbf{0}$, making the Hessian positive definite and the loss strictly convex with a unique global minimum. □

5 Scaling to Large Data

We have established batch gradient descent and proven that for convex problems like OLS, it finds the optimal solution. However, the *batch* computation becomes a bottleneck for large-scale machine learning.

5.1 The Computational Bottleneck of Batch Gradient Descent

Consider a general loss function that is a sum over n data points, $L(\boldsymbol{\theta}) = \frac{1}{n}\sum_{i=1}^{n} L_i(\boldsymbol{\theta})$, where L_i is the loss for the i-th data point. The batch gradient is as follows:

$$\nabla L(\boldsymbol{\theta}) = \frac{1}{n}\sum_{i=1}^{n} \nabla L_i(\boldsymbol{\theta}).$$

A single parameter update requires a full pass over the entire dataset. When n is in the millions or billions, this is prohibitively expensive.

5.2 The Idea of Stochastic Approximation

The solution comes from **stochastic approximation**: replacing an expensive, exact quantity with a cheap, noisy, but unbiased estimate.

The expensive quantity is the true gradient $\nabla L(\boldsymbol{\theta})$. The most extreme approximation uses the gradient from a *single*, randomly chosen data point. This is the essence of stochastic gradient descent (SGD).

This radical approximation works because of the **Data Redundancy Principle**. Large datasets are highly redundant: the gradient from one sample is similar to the gradient from another similar sample. The batch approach wastefully re-computes and averages nearly identical gradient information. SGD exploits this redundancy by using a single sample's gradient as an approximation of the full gradient.

5.3 Formalizing Stochastic Gradient Descent

Definition 5.1 (Update Step)**.** An **update step** (also called an **iteration**) is one step of a recurrence relation of the form

$$\boldsymbol{\theta}_{k+1} = f_k(\boldsymbol{\theta}_k), \qquad k = 0, 1, 2, \dots,$$

where $\boldsymbol{\theta}_k \in \mathbb{R}^p$ is the current parameter vector and $f_k \colon \mathbb{R}^p \to \mathbb{R}^p$ is the update rule at step k.

Definition 5.2 (Epoch)**.** An **epoch** is one complete sequential pass through all n training samples.

After each epoch, the data are typically reshuffled before the next pass. We index epochs by $t \in \{1, \dots, T\}$, where T is the total number of epochs. The number of update steps contained in one epoch depends on the algorithm: batch GD produces one update step per epoch, while SGD produces n.

Example 5.3 (Epochs vs. Update Steps in SGD)**.** Consider the running dataset with $n = 4$ samples, trained for $T = 2$ epochs. In batch GD, one epoch produces exactly one update step: the parameter is modified once using the gradient over all four samples. In SGD, one epoch produces $n = 4$ update steps, one per sample. For $T = 2$ epochs, SGD applies $k = 0, 1, \dots, 7$ update steps in total. The epoch boundary falls between $k = 3$ and $k = 4$: after processing all four samples in epoch $t = 1$, the data are reshuffled and the inner loop restarts. The learning rate schedule $\{\eta_k\}$ is indexed by the global counter k, not by t, so the step size decreases continuously across epoch boundaries rather than resetting at the start of each epoch.

The full SGD procedure is presented in Algorithm 2. The two nested loops correspond to the two time-scales of Definitions 5.1 and 5.2: the outer loop advances the epoch counter t, completing one full pass through the data, while the inner loop advances the update counter k, applying one gradient step per sample. Each epoch produces n update steps, compared to the single update step of batch gradient descent. The learning rate schedule $\{\eta_k\}$ allows the step size to decrease over time, which is essential for convergence as we discuss below.

Algorithm 2: Stochastic Gradient Descent (SGD)

```
   Input: Initial parameters $\boldsymbol{\theta}_0 \in \mathbb{R}^p$, learning rate schedule $\{\eta_k\}$, number of epochs $T$
   Result: Optimized parameters $\boldsymbol{\theta}^*$
1  $k \leftarrow 0$;
2  for $t = 1, 2, \ldots, T$ do
3      Randomly shuffle $\{1, 2, \ldots, n\}$ into a permutation $\pi$;
4      for $j = 1, 2, \ldots, n$ do
5          $i \leftarrow \pi(j)$;
6          Compute the stochastic gradient: $\nabla L_i(\boldsymbol{\theta}_k)$;
7          Update parameters: $\boldsymbol{\theta}_{k+1} \leftarrow \boldsymbol{\theta}_k - \eta_k \nabla L_i(\boldsymbol{\theta}_k)$;
8          $k \leftarrow k + 1$;
9      end
10 end
11 return $\boldsymbol{\theta}_k$
```

5.4 Analysis of the Stochastic Gradient

The stochastic gradient $\nabla L_{i_k}(\boldsymbol{\theta}_k)$ is a random variable whose value depends on the random choice of index i_k. The key property that makes SGD valid is that this estimator is **unbiased**.

Lemma 5.4 (Unbiasedness of the Stochastic Gradient)**.** *The expected value of the stochastic gradient equals the true batch gradient:*

$$\mathbb{E}_i[\nabla L_i(\boldsymbol{\theta})] = \nabla L(\boldsymbol{\theta}).$$

Proof. By the definition of expectation for a discrete random variable, we have:

$$\mathbb{E}_i[\nabla L_i(\boldsymbol{\theta})] = \sum_{j=1}^{n} P(i = j) \cdot \nabla L_j(\boldsymbol{\theta}).$$

Since i is chosen uniformly, $P(i = j) = 1/n$ for all j:

$$\mathbb{E}_i[\nabla L_i(\boldsymbol{\theta})] = \sum_{j=1}^{n} \frac{1}{n} \nabla L_j(\boldsymbol{\theta}) = \frac{1}{n} \sum_{j=1}^{n} \nabla L_j(\boldsymbol{\theta}) = \nabla L(\boldsymbol{\theta}).$$

This holds for any fixed $\boldsymbol{\theta}$. The stochastic gradient is an unbiased estimator of the true gradient. □

Example 5.5 (SGD: Speed Through Stochastic Updates)**.** We apply SGD to the same dataset ($y = \beta x$, $n = 4$, $\beta^* = 1.9$) with $\beta_0 = 0$ and $\eta = 0.02$. The per-sample gradient is $\nabla L_i(\beta) = 2x_i(\beta x_i - y_i)$.

Epoch 1 shuffles the indices into the order $i = 2, 4, 1, 3$. Each update uses a single sample:

1. **Sample** $i = 2$ ($x_2 = 2, y_2 = 4$): $\nabla L_2(0) = 2(2)(0 - 4) = -16$. The update gives $\beta_1 = 0 + 0.02(16) = 0.320$.

2. **Sample** $i = 4$ ($x_4 = 4, y_4 = 8$): $\nabla L_4(0.32) = 2(4)(1.28 - 8) = -53.76$. The update gives $\beta_2 = 0.320 + 1.075 = 1.395$.

3. **Sample** $i = 1$ ($x_1 = 1, y_1 = 2$): $\nabla L_1(1.395) = 2(1)(1.395 - 2) = -1.21$. The update gives $\beta_3 = 1.395 + 0.024 = 1.419$.

4. **Sample** $i = 3$ ($x_3 = 3, y_3 = 5$): $\nabla L_3(1.419) = 2(3)(4.258 - 5) = -4.45$. The update gives $\beta_4 = 1.419 + 0.089 = 1.508$.

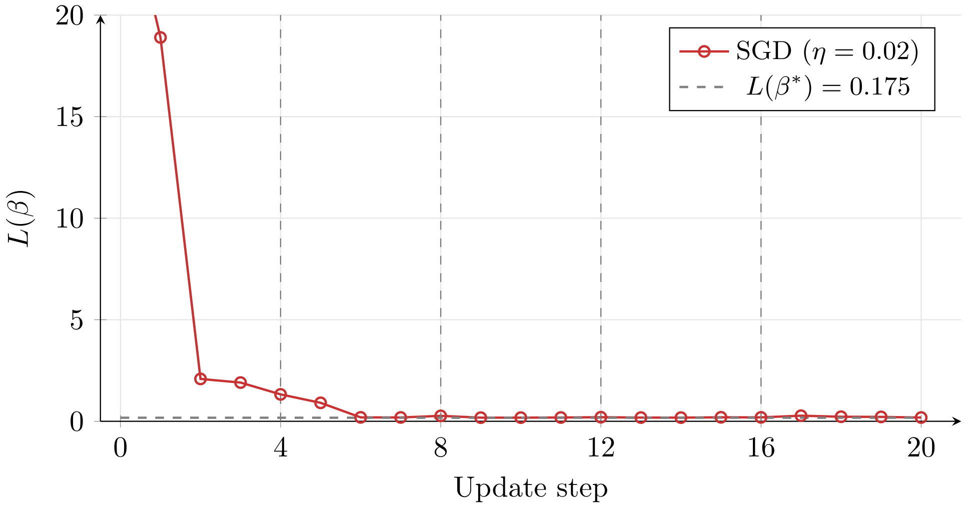

Figure 2: SGD convergence ($\eta = 0.02$). Loss is plotted at each of the 20 update steps across 5 epochs (dashed vertical lines mark epoch boundaries). The noisy, sawtooth-like trajectory is characteristic of SGD: rapid initial descent followed by oscillation near the optimum $L(\beta^*) = 0.175$.

After just one epoch, SGD reaches $\beta \approx 1.508$—already 79% of the way to β^*. By contrast, batch GD's single update per epoch reaches $\beta = 1.425$. The per-sample updates vary wildly: sample 4 moved β by 1.075, while sample 1 moved it by only 0.024.

Continuing for 4 more epochs (summarized at epoch boundaries):

Epoch	β	$L(\beta)$
0	0.000	27.250
1	1.508	1.325
2	1.789	0.268
3	1.841	0.201
4	1.850	0.193
5	1.937	0.185

By epoch 2, SGD is already near β^*. However, note the oscillation: β overshoots (epoch 5: $\beta = 1.937$) and undershoots ($\beta = 1.789$) around the optimum. Near the minimum, different samples push β in opposite directions—sample 3 with $y_3 = 5$ (below the trend) pushes β down, while sample 4 with $y_4 = 8$ (above the trend) pushes it up.

Key strength: speed. With $n = 4$ updates per epoch instead of one, SGD makes dramatically faster early progress. By epoch 2, SGD achieves a loss of 0.268, comparable to what batch GD needs 2 full iterations to reach. Additionally, the inherent noise in SGD updates can help escape shallow local minima in non-convex problems—a critical advantage for deep learning. Figure 2 captures this behavior: rapid initial descent followed by oscillation near the optimum, with the characteristic noisy, sawtooth-like pattern visible across epochs.

Example 5.6 (SGD: Gradient Variance and Non-Monotonic Loss)**.** We demonstrate the two principal weaknesses of SGD using the same dataset ($y = \beta x$, $n = 4$, $\beta^* = 1.9$) with $\eta = 0.02$, starting from $\beta_0 = 1.5$—close enough to the optimum that the weaknesses are immediately visible.

Part 1: Gradient variance. At $\beta_0 = 1.5$, the four per-sample gradients are as follows:

$$\begin{aligned}
\nabla L_1(1.5) &= 2(1)(1.5 - 2) = -1.000,\\
\nabla L_2(1.5) &= 2(2)(3.0 - 4) = -4.000,\\
\nabla L_3(1.5) &= 2(3)(4.5 - 5) = -3.000,\\
\nabla L_4(1.5) &= 2(4)(6.0 - 8) = -16.000.
\end{aligned}$$

The batch gradient is $\nabla L(1.5) = \frac{1}{4}(-1 - 4 - 3 - 16) = -6.0$. The individual estimates span a 16-fold range: sample 1 underestimates the true descent direction by a factor of 6, while sample 4 overestimates it by a factor of $8/3 \approx 2.67$. Any single-sample update is therefore a noisy surrogate for the batch gradient.

Part 2: Non-monotonic loss. With sample order $i = 4, 1, 3, 2$ (one shuffled epoch), each step applies $\beta_{k+1} = \beta_k - 0.02 \cdot \nabla L_i(\beta_k)$. The four updates are traced below.

Step k	Sample i	β_k	$\nabla L_i(\beta_k)$	β_{k+1}	$L(\beta_{k+1})$
0	4	1.500	−16.000	1.820	0.223
1	1	1.820	−0.360	1.827	0.214
2	3	1.827	+2.886	1.769	0.303
3	2	1.769	−1.848	1.806	0.241

At step $k = 2$, sample $i = 3$ produces a positive gradient ($\nabla L_3 = +2.886$), pulling β from 1.827 past the optimum to 1.769 and increasing the loss from 0.214 to 0.303. This behaviour is impossible in batch GD, where every step is guaranteed to reduce the loss. Here, sample 3 acts against the population trend: its residual pushes β in the wrong direction.

Key weakness. SGD with a constant learning rate does not converge to β^*. Near the optimum, high-variance updates overshoot in alternating directions, sustaining persistent oscillation rather than convergence. Convergence to a point requires a decaying learning rate schedule satisfying $\sum_{k=1}^{\infty} \eta_k = \infty$ and $\sum_{k=1}^{\infty} \eta_k^2 < \infty$, which forces step sizes to shrink and the oscillations to subside.

This property ensures that while any individual SGD step may not point in the direction of steepest descent, the *average* direction over many iterations is correct. The algorithm makes noisy but statistically correct progress toward the minimum.

5.5 Properties and Convergence of SGD

The stochastic nature of the updates gives SGD a distinct set of properties.

- **Erratic Convergence Path:** Unlike the smooth path of batch GD on a convex surface, SGD follows an erratic trajectory. The high variance of the single-sample gradient causes the parameters to oscillate around the descent path. Near the minimum, this oscillation prevents the algorithm from settling at the exact optimum.

- **Learning Rate Schedule:** To achieve convergence, the learning rate η_k must decrease over time. A standard condition is $\sum_{k=1}^{\infty} \eta_k = \infty$ and $\sum_{k=1}^{\infty} \eta_k^2 < \infty$. Common choices include $\eta_k \propto 1/k$ or $\eta_k \propto 1/\sqrt{k}$. The decaying step size allows oscillations to shrink, enabling the algorithm to settle closer to the true minimum.

- **Escaping Local Minima:** In non-convex optimization, the noise in SGD can be advantageous. The random perturbations can help the algorithm escape shallow local minima and continue searching for deeper, better minima. Batch GD, with its deterministic steps, would remain trapped in the first local minimum it encounters.

6 The Modern Workhorse

While SGD solves the computational bottleneck of batch GD, its high-variance updates can lead to slow final convergence and unstable training. This motivates **Mini-Batch Gradient Descent**, the de facto standard for training modern large-scale machine learning models.

6.1 Formalizing Mini-Batch Gradient Descent

Instead of using all n samples (Batch) or 1 sample (SGD), we use a small random batch of b samples, where $1 \ll b \ll n$.

Algorithm 3: Mini-Batch Gradient Descent

```
Input: Initial parameters $\boldsymbol{\theta}_0 \in \mathbb{R}^p$, learning rate $\eta > 0$, mini-batch size $b$, number of
       epochs $T$
Result: Optimized parameters $\boldsymbol{\theta}^*$
1  $k \leftarrow 0$;
2  for $t = 1, 2, \ldots, T$ do
3      Randomly shuffle and partition $\{1, 2, \ldots, n\}$ into mini-batches $\mathcal{B}_1, \mathcal{B}_2, \ldots, \mathcal{B}_{\lceil n/b \rceil}$;
4      for $m = 1, 2, \ldots, \lceil n/b \rceil$ do
5          Compute the mini-batch gradient: $\mathbf{g}_k \leftarrow \frac{1}{b} \sum_{i \in \mathcal{B}_m} \nabla L_i(\boldsymbol{\theta}_k)$;
6          Update parameters: $\boldsymbol{\theta}_{k+1} \leftarrow \boldsymbol{\theta}_k - \eta\, \mathbf{g}_k$;
7          $k \leftarrow k + 1$;
8      end
9  end
10 return $\boldsymbol{\theta}_k$
```

The mini-batch gradient descent procedure is formalized in Algorithm 3. The structure mirrors SGD, but the inner loop iterates over mini-batches of size b rather than individual samples, producing $\lceil n/b \rceil$ updates per epoch. Each mini-batch gradient $\mathbf{g}_k$ averages over b per-sample gradients, reducing the variance of the estimate by a factor of b relative to SGD while preserving the unbiasedness property. Setting $b = 1$ recovers SGD, and setting $b = n$ recovers batch gradient descent, making mini-batching the natural interpolation between the two extremes.

Example 6.1 (Mini-Batch GD: Balancing Speed and Stability)**.** We apply mini-batch gradient descent with $b = 2$ to the same dataset ($y = \beta x$, $n = 4$, $\beta^* = 1.9$) with $\beta_0 = 0$ and $\eta = 0.03$. One epoch consists of $n/b = 2$ updates.

Epoch 1 partitions the shuffled data into mini-batches $\mathcal{B}_0 = \{2, 4\}$ and $\mathcal{B}_1 = \{1, 3\}$.

1. **Mini-batch $\mathcal{B}_0 = \{2, 4\}$:** The mini-batch gradient averages two per-sample gradients:

$$\mathbf{g}_0 = \tfrac{1}{2}[\nabla L_2(0) + \nabla L_4(0)] = \tfrac{1}{2}[-16 + (-64)] = -40.$$

   The update gives $\beta_1 = 0 - 0.03(-40) = 1.200$.

2. **Mini-batch $\mathcal{B}_1 = \{1, 3\}$:** Using the updated parameter $\beta_1 = 1.200$, we compute the gradient:

$$\mathbf{g}_1 = \tfrac{1}{2}[\nabla L_1(1.2) + \nabla L_3(1.2)] = \tfrac{1}{2}[-1.6 + (-8.4)] = -5.0.$$

   The update gives $\beta_2 = 1.200 - 0.03(-5.0) = 1.350$.

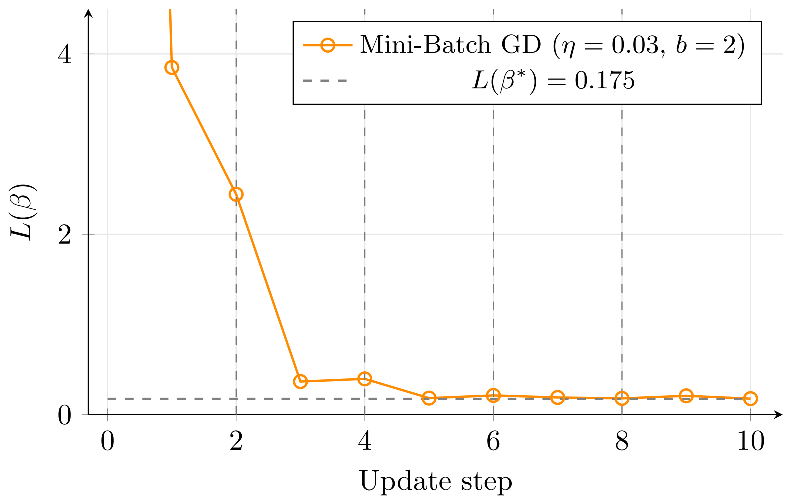

Figure 3: Mini-Batch GD convergence ($\eta = 0.03$, $b = 2$). Loss is plotted at each of the 10 update steps across 5 epochs. The trajectory is smoother than SGD but faster than Batch GD, reflecting the variance-reducing effect of averaging over mini-batches.

After one epoch, mini-batch GD reaches $\beta \approx 1.350$—between batch GD's $\beta = 1.425$ (one update) and SGD's $\beta = 1.508$ (four updates). The step sizes (1.200 and 0.150) are more uniform than SGD's (which ranged from 0.024 to 1.075), thanks to the variance reduction from averaging two samples.

Continuing for 4 more epochs:

Epoch	β	$L(\beta)$
0	0.000	27.250
1	1.350	2.444
2	1.728	0.397
3	1.829	0.213
4	1.874	0.180
5	1.918	0.177

By epoch 5, $\beta \approx 1.918$ and $L \approx 0.177$, very close to the optimum. The convergence is smoother than SGD but faster than batch GD per epoch, since mini-batch makes 2 updates per epoch instead of 1.

Key strength: balance. The mini-batch approach inherits the multiple-updates-per-epoch advantage of SGD while reducing variance through averaging. In practice, the mini-batch computation maps perfectly to GPU SIMD parallelism: a mini-batch of size $b = 32$ or 64 can be processed in roughly the same wall-clock time as a single sample, making mini-batch GD the dominant choice for modern deep learning. Figure 3 shows the resulting convergence profile: the loss trajectory is markedly smoother than SGD's erratic path while converging more rapidly per epoch than batch GD.

Example 6.2 (Non-Monotonic Loss and Hyperparameter Sensitivity)**.** We demonstrate the two principal weaknesses of mini-batch GD using the same dataset ($y = \beta x$, $n = 4$, $\beta^* = 1.9$) with $b = 2$ and $\eta = 0.03$, starting from $\beta_0 = 1.5$. The per-sample gradient is $\nabla L_i(\beta) = 2x_i(\beta x_i - y_i)$; the mini-batch gradient for batch $\mathcal{B}_m = \{i, j\}$ is $g_k = \frac{1}{2}(\nabla L_i(\beta_k) + \nabla L_j(\beta_k))$.

Part 1: Non-monotonic loss. Two consecutive epochs use different random shufflings, producing four updates. The trace is shown below, with epochs separated by a rule.

Step k	Batch $\mathcal{B}_m$	β_k	g_k	β_{k+1}	$L(\beta_{k+1})$
0	$\{3,4\}$	1.500	-9.500	1.785	0.274
1	$\{1,2\}$	1.785	-1.075	1.817	0.226
2	$\{2,4\}$	1.817	-3.660	1.927	0.180
3	$\{1,3\}$	1.927	$+2.270$	1.859	0.187

At step $k = 3$, batch $\mathcal{B}_3 = \{1, 3\}$ yields a *positive* mini-batch gradient $g_3 = +2.270$. Sample 3 has residual $\beta_3 x_3 - y_3 = 1.927 \times 3 - 5 = 0.781 > 0$, contributing gradient $+4.686$, which dominates the small negative contribution -0.146 from sample 1. The net gradient pushes β downward from 1.927 to 1.859—past the optimum—and the loss rises from 0.180 to 0.187. Averaging over $b = 2$ samples reduces but does not eliminate variance; a single sample whose gradient contradicts the true descent direction can reverse the net mini-batch gradient.

Part 2: Dual hyperparameter sensitivity. The mini-batch approach requires tuning two hyperparameters rather than one. The learning rate η must satisfy the same convergence condition as batch GD. The batch size b controls the variance–cost trade-off: The mini-batch gradient has variance

$$\mathrm{Var}(g_k) = \frac{1}{b}\,\mathrm{Var}(\nabla L_i(\beta))\,,$$

so doubling b halves the gradient variance but doubles the computational cost per update. For $b = 1$ the method reduces to SGD (maximal variance, minimal cost); for $b = n$ it reduces to batch GD (zero variance, maximal cost). The practitioner must select b from this continuum, typically guided by GPU memory constraints and empirical validation rather than analytical criteria.

Key weakness. For any $b < n$, the residual gradient variance $\mathrm{Var}(g_k) = \mathrm{Var}(\nabla L_i)/b > 0$ sustains oscillation near the optimum under a constant learning rate, preventing exact convergence. The required decaying schedule $\sum_k \eta_k = \infty$, $\sum_k \eta_k^2 < \infty$ applies equally to mini-batch GD. Furthermore, the two-dimensional hyperparameter space (η, b) roughly squares the cost of hyperparameter search relative to batch GD.

6.2 Comparing the Three Variants

Having applied all three gradient descent variants to the same dataset, we can now visualize their convergence profiles side by side. Figure 4 plots the loss at the end of each epoch for all three methods.

Three key observations emerge from this comparison. First, **SGD converges fastest per epoch** because it performs n parameter updates per pass through the data, versus n/b for mini-batch and just 1 for Batch. After epoch 1, SGD has already reached $L \approx 1.33$, compared to 1.87 for batch GD. Second, **Batch GD converges most smoothly**: its loss curve is monotonically decreasing with no oscillation, making it the easiest to analyze and debug. Third, **mini-batch GD offers the best practical trade-off**: it converges nearly as fast as SGD per epoch while maintaining a smoother trajectory. In production settings with GPUs, mini-batch GD additionally benefits from hardware parallelism that makes each mini-batch update nearly as fast as a single-sample SGD update.

6.3 Analysis of Benefits

This variant combines the advantages of both batch GD and SGD.

1. **More Stable Gradient Estimate:** Averaging over b samples (e.g., $b = 32, 64, 128$) reduces the variance of the gradient estimate by a factor of $1/b$ compared to SGD. This leads to a smoother convergence path and allows a larger learning rate.

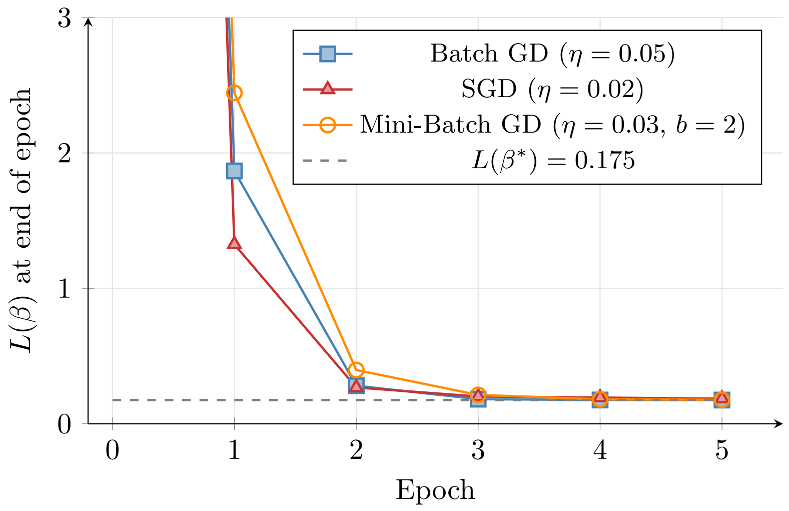

Figure 4: Comparison of Batch GD, SGD, and Mini-Batch GD on the same dataset over 5 epochs. All three methods start at $L(\beta_0) = 27.25$ and converge toward $L(\beta^*) = 0.175$ (dashed line), but with distinct profiles. SGD reaches the lowest loss after epoch 1 due to its $n = 4$ updates per epoch, but oscillates near the optimum. Batch GD converges most smoothly but uses only one update per epoch. Mini-Batch GD ($b = 2$) balances both behaviors.

2. **Computational Efficiency:** Updates are far more frequent than in batch GD: n/b updates per epoch instead of one. This produces much faster progress in the early stages of training.

3. **Hardware Parallelization:** The matrix operations required for a mini-batch are highly parallelizable. Modern GPUs, with thousands of cores optimized for SIMD (Single Instruction, Multiple Data) operations, process a mini-batch of size 32 or 64 in roughly the same time as a single sample. SGD cannot saturate GPU parallelism; batch GD often exceeds GPU memory. Mini-batching provides the optimal balance.

The dominance of mini-batch GD is a consequence of both its algorithmic properties and its compatibility with modern hardware. The algorithm is perfectly suited to GPU architectures, and the co-evolution of mini-batch methods and GPU computing has been central to the rise of deep learning. Table 1 summarizes the trade-offs among the three variants.

7 Conclusion

This lecture has bridged multivariable calculus and the core machinery of modern machine learning. We framed *learning from data* as a concrete mathematical problem: minimizing a high-dimensional loss function.

We used the gradient and Hessian to understand the local geometry of the loss landscape, then derived gradient descent as a principled procedure rooted in the first-order Taylor approximation. We proved the cornerstone theorem of convex optimization—that any local minimum of a convex function is a global minimum—and applied it to the OLS loss, establishing convergence guarantees for gradient descent.

Finally, we addressed the computational realities of large-scale data. The prohibitive cost of batch gradient descent motivated stochastic and mini-batch gradient descent, where we analyzed the trade-offs between computational cost, gradient variance, and hardware parallelism. This

Table 1: Comparison of Gradient Descent Variants

Property	**Batch GD**	**SGD**	**Mini-Batch GD**
Gradient Uses	All n samples	1 random sample	b random samples $(1 \ll b \ll n)$
Update Frequency	Once per epoch	n times per epoch	n/b times per epoch
Cost per Update	$O(np)$	$O(p)$	$O(bp)$
Memory	High (entire dataset)	Low (one sample)	Medium (one mini-batch)
Convergence Path	Smooth (for convex)	Noisy, erratic	Smoother than SGD
Hardware Utilization	Poor (memory bottleneck)	Poor (no vectorization)	Excellent (GPU parallelism)
Key Advantage	Convergence guarantee (convex)	Fast updates; escapes local minima	Optimal efficiency–stability balance

framework—from abstract calculus to practical, scalable algorithms—is the engine of modern machine learning.