Machine Learning: 1-4

◷ Created	@2025年12月4日 10:36
⊙ Class	Fundamentals of Computer Science

1 Syllabus(课程大纲) & Introduction

Applications With and Without Machine Learning

Credit Risk Assessment

Taditional (Non-ML) Credit Risk Assessment: A rule-based scoring system is used, where human-designed rules assign fixed points. Examples include giving extra points for high salary or mid-range age, and subtracting(减) points for previous defaults(违约记录). A loan(贷款) is approved only if the total score reaches a threshold. **The main limitation is that these rigid rules cannot capture complex interactions, making the system overly simplistic.**

ML-Based Credit Risk Assessment: Machine learning models such as logistic regression and random forests are trained on historical client data. These models estimate the probability of default and automatically adapt when new patterns appear. Approval is based on whether the predicted probability falls below a certain cutoff(截断点). **A key advantage is the ability to discover hidden nonlinear relationships and reduce human bias.**

Supervised and Unsupervised Learning

Machine learning paradigms differ mainly by **how much information is given to the model during learning**.

- **Reinforcement Learning** provides very sparse scalar(标量) rewards — only a *few bits* per sample.
- **Supervised Learning** provides explicit human-labeled answers — typically *10–10,000 bits per sample*.
- **Self-Supervised Learning** uses the input itself to generate labels, giving *millions of bits per sample* and enabling large-scale modern models.

Supervised Learning

Supervised learning learns a mapping

$$f(x) \approx y$$

from a dataset of labeled pairs (x_i, y_i).

Its goal is to **minimize the error between predictions and ground-truth labels**.

Core Idea: Supervised learning uses **labeled data** with clearly defined input → output relationships. The model extracts patterns that allow generalization to new, unseen inputs.

Types of Supervised Tasks

- **Classification:** Classification assigns input data to discrete categories.
  - **Binary Classification**
  - **Multi-class Classification**
- **Regression:** Regression predicts **continuous numerical values**, not categories.

Unsupervised Learning

Unsupervised learning discovers structure in **unlabeled data**, with no predefined targets. It aims to:

- Reveal hidden **patterns**, **clusters**, or **representations**
- Understand data distribution
- Reduce dimensionality
- Serve as a foundation for many generative and self-supervised models

Dimensionality Reduction

High-dimensional data (images, text, genomics) is difficult to analyze or visualize. Dimension reduction seeks low-dimensional representations that preserve essential information.

$$z = f(x), \quad x \in \mathbb{R}^n,\ z \in \mathbb{R}^m,\ n \gg m$$

Techniques:

- **PCA**: directions of maximum variance
- **t-SNE**: preserve local neighborhoods

- **Autoencoders**: learn compact latent encodings

The goal is **information-preserving compression**, not merely discarding dimensions.

Autoencoders

Autoencoders learn a mapping that compresses data into a latent vector and reconstructs it:

- Input → Encoder → Latent vector z
- z → Decoder → Reconstructed data

Self-Supervised Learning (SSL)

Self-supervised learning generates labels **from the data itself** through pretext tasks(伪任务). The strategic goal is to learn rich representations **without manual annotation**.

Examples of SSL tasks:

- Predict missing words in sentences (LLMs)
- Predict masked image patches (Vision Transformers)

SSL is the foundation of modern large models such as **GPT**, **CLIP**, **SimCLR**, **BERT**, etc. It effectively acts as *supervised learning with self-generated labels*.

Generative Models

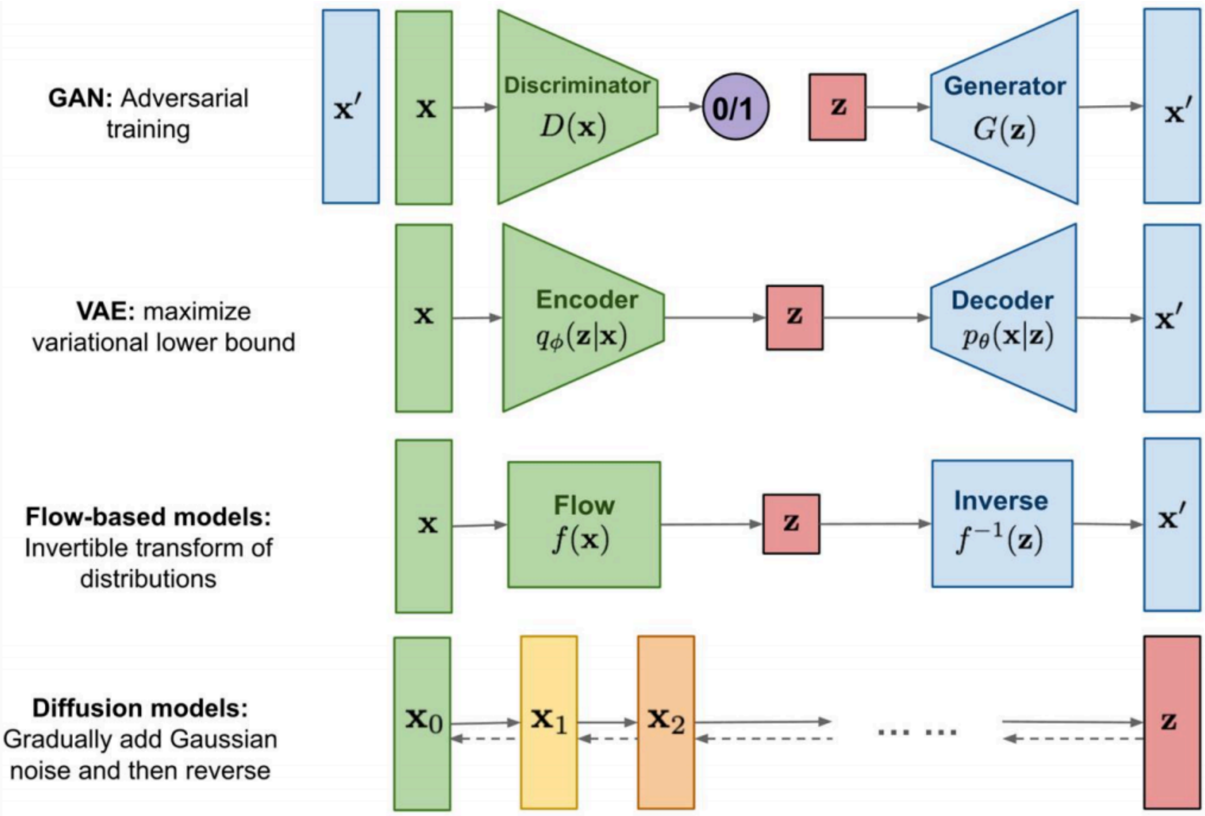
Generative modeling focuses on understanding and sampling from the data distribution $P(X)$.

Everything begins with estimating or learning the underlying(潜在) distribution.

Examples:

- Autoencoders: learn manifold(流形) structures $G(d, p)$
- Probabilistic generative models: $x_i \sim G(\mu, \sigma)$

Generative models can also perform clustering or dimension reduction as a byproduct (e.g., β-VAE, InfoGAN).



Reinforcement Learning

Reinforcement Learning (RL) is a learning paradigm in which an agent learns by **interacting with an environment** and improving behavior through **trial and error**. Unlike supervised learning, RL does not receive labeled answers. Instead, it receives **rewards** that indicate how good each action was. The agent's objective is to learn a **policy** $\pi(a \mid s)$ that selects actions maximizing **expected cumulative reward** over time.

Core Idea: The central process of RL is iterative interaction:

1. The agent observes a **state** (s).
2. Chooses an **action** (a).
3. The environment responds with a **reward** and a new **state**.
4. The agent updates its policy to increase future rewards.

The essence(本质) is continuous feedback-based learning, not static data learning.

Goal of Reinforcement Learning: The ultimate aim is to **maximize cumulative reward**, also called the *return*. This encourages the agent to consider *long-term consequences*, not just immediate gains.

The Multi-Armed Bandit Problem

The multi-armed bandit is the simplest RL setting and captures the essence of **exploration vs. exploitation**.

Scenario: Imagine being in a casino(赌场) facing several slot machines(老虎机) ("one-armed bandits"), but you don’t know which machine yields the highest payout.

Example machines:

- **Machine A**: Pays out **30%** of the time
- **Machine B**: Pays out **50%** of the time
- **Machine C**: Pays out **10%** of the time

You must choose repeatedly, aiming to maximize winnings over time. Therefore, the bandit problem forces the agent to balance:

- **Exploration** — trying different machines to gather information
- **Exploitation** — selecting the machine that seems best based on current knowledge

This balance is fundamental to all RL algorithms.

2 Linear Model

Machine Learning Pipeline

The machine learning pipeline describes the end-to-end workflow of turning raw data into a trained and deployable model. Although textbooks often present this process as linear, in practice it is iterative: decisions at later stages may require revisiting earlier steps.

Data Preparation

Machine learning begins with data. Raw data must be collected, cleaned, and transformed into a form usable by algorithms. This stage defines the quality ceiling of the entire system. Typical operations include:

- Handling missing values and outliers
- Normalizing or standardizing features
- Constructing the feature matrix
- Splitting data into training and testing sets

If the dataset is supervised, each training example consists of attribute–label pairs:

$$(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)$$

Here x_i represents the input attributes, and y_i is the supervision signal (label).

Outputs of this stage include:

- Cleaned dataset
- Feature matrix
- Training and testing datasets

Machine Learning Algorithm

Once data is prepared, the next step is to choose and configure the appropriate algorithm. The decision depends on problem type (regression, classification, etc.), data scale, interpretability requirements, and computational budget.

The learning goal is to find a model f_θ whose predictions minimize a loss function over the dataset:

$$\mathcal{L}(f_\theta(x^{(i)}), y^{(i)})$$

This is where theoretical assumptions meet empirical reality. Algorithms range from simple linear models to highly expressive neural networks. Choosing well is both an art and a science.

Outputs of this stage include:

- Selected algorithm
- Algorithm configuration
- Baseline model

Training Recipes

The final stage focuses on how to optimize the model. The “recipe” metaphor is quite apt: even with the same ingredients (data + model), the outcome depends heavily on the cooking process.

Training strategies include:

- Hyperparameter tuning
- Choosing optimization algorithms (e.g., closed-form solutions or numerical methods)
- Applying validation techniques to avoid overfitting

- Deciding stopping criteria

Closed-form solutions exist for certain simple models (e.g., linear regression under specific assumptions), but most modern ML relies on numerical optimization such as gradient descent.

Key questions answered here:

- How do we optimize the model effectively?
- When should we stop training?
- How do we evaluate generalization performance?

Outputs of this stage include:

- Trained model
- Validation results
- Optimized hyperparameters

Linear Models

Linear models provide one of the simplest yet most foundational ways to understand supervised learning. Despite their simplicity, they capture key ideas: assumptions, loss minimization, optimization, and interpretability.

Basic Assumption of a Linear Model

A linear regression model assumes a linear relationship between the input variable x and the output y:

$$y = \beta_0 + \beta_1 x + \epsilon,$$

where:

- β_0 is the intercept(截距) (bias term),
- β_1 is the slope,
- ϵ is the error term capturing all unexplained variability.

The task in linear regression is to estimate parameters β_0 and β_1 from data so that the model fits as closely as possible to the observed points.

Loss Function and Sum of Squared Errors (SSE)

To find the best parameters, we introduce a loss function measuring the mismatch between predictions and true values.

Given data points (x_i, y_i), predictions are:

$$\hat{y}_i = \beta_0 + \beta_1 x_i.$$

The **sum of squared errors (SSE)** is defined as:

$$\mathcal{L}(\beta_0, \beta_1) = \sum_i (\hat{y}_i - y_i)^2.$$

Here the loss function $\mathcal{L}$ is a function **of the parameters**, not of the data.

Minimizing SSE ensures that the regression line stays as close as possible to the entire dataset, balancing overestimates and underestimates.

The optimization problem is:

$$(\beta_0^*, \beta_1^*) = \arg\min_{\beta_0, \beta_1} \mathcal{L}(\beta_0, \beta_1).$$

General Form and Interpretation

A slightly more general model might include nonlinear basis functions while still being linear in parameters:

$$f(x) = \beta_1 x + \beta_2 x^2 + c.$$

Even though the model includes x^2, regression remains *linear in parameters* β_1, β_2, c.

The loss function becomes:

$$\ell(\beta) = \sum_i (\hat{y}_i - y_i)^2, \qquad \hat{y}_i = f(x_i).$$

Again, ℓ is a function **of the parameters**.

Solving for Optimal Parameters

To find the minimum of the loss function, we compute partial derivatives:

$$\frac{\partial \ell(\beta_0, \beta_1)}{\partial \beta_1} = 0, \qquad \frac{\partial \ell(\beta_0, \beta_1)}{\partial \beta_0} = 0.$$

Solving these equations gives:

$$\beta_1 = \frac{\sum x_i y_i - n\bar{x}\bar{y}}{\sum x_i^2 - n\bar{x}^2} = \frac{\sum (x_i - \bar{x})(y_i - \bar{y})}{\sum (x_i - \bar{x})^2}.$$

This leads to the elegant statistical form:

$$\beta_1 = \frac{\mathrm{Cov}(x, y)}{\mathrm{Var}(x)}.$$

The intercept is:

$$\beta_0 = \bar{y} - \beta_1 \bar{x}.$$

Intuition Behind the Closed-Form Solution

Even though the algebra looks mechanical, conceptually the solution has a clean meaning:

- If (x) and (y) are strongly positively correlated, covariance is large → slope is large.
- If (x) has little variance, a tiny change in (x) must cause a large change in (y) to match the observations → slope grows.
- The intercept ensures the fitted line is centered with the data mean.

Summary

Linear regression teaches the full ML workflow in miniature(缩影):

- **Model assumption:** linearity.
- **Loss function:** squared error.
- **Optimization:** closed-form solution via calculus.
- **Interpretation:** covariance structure explains slope.

Math Recap for Linear Models

Partial Derivatives in Linear Models

Consider a linear model with multiple outputs:

$$y_i = \sum_{j=1}^{n} x_{ij} w_j$$

Each output y_i depends on all weights $w_1, \ldots, w_n$. When taking derivatives(求导), we always ask a precise question: **Which variable are we differentiating with respect to(对……求导)?**

For a fixed weight w_k:

$$\frac{\partial y_i}{\partial w_k} = \frac{\partial}{\partial w_k} \left(\sum_{j=1}^{n} x_{ij} w_j \right) = x_{ik}.$$

This result is simple but fundamental: **the derivative of the output with respect to a weight is the corresponding input feature.**

Stacking all derivatives together gives:

$$\frac{\partial \mathbf{y}}{\partial \mathbf{w}} = X^\top.$$

This explains why the data matrix X repeatedly appears in gradient-based learning rules.

Jacobian Matrix: Derivatives of Vector-Valued Functions

When a function maps vectors to vectors, derivatives are no longer scalars.

For a function:

$$f : \mathbb{R}^n \to \mathbb{R}^m, \quad f(\mathbf{x}) = \begin{bmatrix} f_1(\mathbf{x}) \\ \vdots \\ f_m(\mathbf{x}) \end{bmatrix},$$

the derivative is the **Jacobian matrix**:

$$J = \begin{pmatrix} \frac{\partial f_1}{\partial x_1} & \cdots & \frac{\partial f_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial f_m}{\partial x_1} & \cdots & \frac{\partial f_m}{\partial x_n} \end{pmatrix}.$$

Each row corresponds to one output, and each column corresponds to one input dimension.

In linear models, the Jacobian has a particularly clean structure, which makes optimization tractable(易处理的).

Hessian Matrix: Second-Order Structure

For a scalar-valued function:

$$f : \mathbb{R}^n \to \mathbb{R},$$

the **Hessian matrix** captures second-order derivatives:

$$H(f) = \begin{pmatrix} \frac{\partial^2 f}{\partial x_1^2} & \cdots & \frac{\partial^2 f}{\partial x_1 \partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial^2 f}{\partial x_n \partial x_1} & \cdots & \frac{\partial^2 f}{\partial x_n^2} \end{pmatrix}.$$

The Hessian describes **curvature(曲率)**. In optimization:

- Positive definite(正定) Hessian → unique global minimum
- Indefinite(不定) Hessian → saddle points(鞍点) may exist

For squared loss in linear regression, the Hessian turns out to be constant(常数) and positive semidefinite(半正定), which guarantees convexity(凸性).

From Scalar Equations to Vector Form

Instead of writing predictions one by one:

$$\hat{y}_i = w_0 + w_1 x_i,$$

we rewrite the model in vectorized form:

$$\hat{\mathbf{Y}} = XW,$$

where:

- $W = [w_0, w_1]^\top$
- $X = \begin{pmatrix} 1 & x_0 \\ 1 & x_1 \\ \vdots & \vdots \\ 1 & x_n \end{pmatrix}$
- $\hat{\mathbf{Y}} = [\hat{y}_0, \ldots, \hat{y}_n]^\top$

Vectorization is not cosmetic(表面的) — it reveals the geometry of learning.

Loss Function in Matrix Form

Using squared error, the loss becomes:

$$\ell(W) = ||Y - XW||_F^2.$$

This compact expression encodes all data points simultaneously(同时).

Taking the derivative with respect to W:

$$\frac{\partial \ell(W)}{\partial W} = -2X^\top (Y - XW).$$

😈 Let $X \in \mathbb{R}^{n\times d}, W \in \mathbb{R}^{d\times k}, Y \in \mathbb{R}^{n\times k}$. Define the residual(残差)

$$E(W) = Y - XW \quad (\in \mathbb{R}^{n\times k}).$$

The loss is the squared Frobenius norm(F-范数的平方):

$$\ell(W) = |Y - XW|_F^2 = |E(W)|_F^2.$$

Step 1: Rewrite the Frobenius norm using trace
A key identity is

$$|A|_F^2 = \text{tr}(A^\top A).$$

So

$$\ell(W) = \text{tr}\left(E^\top E\right) = \text{tr}\left((Y - XW)^\top (Y - XW)\right).$$

Expand:

$$\begin{aligned}(Y - XW)^\top (Y - XW) &= (Y^\top - W^\top X^\top)(Y - XW) \\ &= Y^\top Y - Y^\top XW - W^\top X^\top Y + W^\top X^\top XW.\end{aligned}$$

Take trace (and use linearity of trace):

$$\ell(W) = \text{tr}(Y^\top Y) - \text{tr}(Y^\top XW) - \text{tr}(W^\top X^\top Y) + \text{tr}(W^\top X^\top XW).$$

Now use the cyclic property(循环不变性) $\text{tr}(ABC) = \text{tr}(BCA) = \text{tr}(CAB)$, and note that the middle two terms are equal scalars($tr(A) = tr(A^\top)$):

$$\text{tr}(Y^\top XW) = \text{tr}(W^\top X^\top Y).$$

So

$$\ell(W) = \text{tr}(Y^\top Y) - 2\text{tr}(W^\top X^\top Y) + \text{tr}(W^\top X^\top XW).$$

Step 2: Differentiate term by term w.r.t. W

1. $\text{tr}(Y^\top Y)$'s derivative is 0.
2. For the linear trace term, use the standard result:

$$\frac{\partial}{\partial W}\mathrm{tr}(W^\top A) = A \quad (\text{same shape as } W).$$

Here $A = X^\top Y$. Therefore:

$$\frac{\partial}{\partial W}\left[-2\mathrm{tr}(W^\top X^\top Y)\right] = -2X^\top Y.$$

3. For the quadratic term(二次项), use:

$$\frac{\partial}{\partial W}\mathrm{tr}(W^\top A W) = (A + A^\top)W.$$

Here $A = X^\top X$, which is symmetric because $(X^\top X)^\top = X^\top X$. So $(A + A^\top) = 2A$, giving

$$\frac{\partial}{\partial W}\mathrm{tr}(W^\top X^\top X W) = 2X^\top X W.$$

Combine all pieces:

$$\frac{\partial \ell(W)}{\partial W} = -2X^\top Y + 2X^\top X W = 2X^\top (XW - Y).$$

Setting the gradient to zero yields the **normal equation**:

$$X^\top X W = X^\top Y.$$

😈 Normal Equation(正规方程): The predicted value XW is the projection of Y onto the subspace spanned by the column space of X in the sample space.

Closed-Form Solution and the Pseudoinverse

If $X^\top X$ is invertible, the optimal solution is:

$$W^* = (X^\top X)^{-1} X^\top Y.$$

This expression is known as the **least-squares solution(最小二乘解)**, and $(X^\top X)^{-1} X^\top$ is the **pseudoinverse(伪逆) of** X.

> 😈 Pseudoinverse: When a matrix is non-invertible (either non-square(非方阵) or singular(奇异)), we can define a matrix that serves as the “closest possible inverse.”

Key interpretation:

- The solution projects Y onto the column space of X.
- Learning is equivalent to geometric projection in feature space.

Linear Regression Viewed Probabilistically

Assume the model

$$y = Xw + \varepsilon$$

where the noise is Gaussian:

$$\varepsilon \sim \mathcal{N}(0, \sigma^2).$$

This assumption is not innocent; it *hard-codes* a loss function, whether we like it or not.

Likelihood and log-likelihood

For a single observation y, conditioned on x and w, the likelihood is

$$p(y \mid x, w) = \frac{1}{\sigma\sqrt{2\pi}} \exp\left(-\frac{1}{2\sigma^2}(y - x^\top w)^2\right).$$

For i.i.d. samples, the joint likelihood is the product over all data points. Since products are annoying and numerically unstable, we immediately take logs:

$$\log p(y \mid X, w) = -\frac{n}{2}\log(2\pi\sigma^2) - \frac{1}{2\sigma^2}\sum_{i=1}^{n}(y_i - x_i^\top w)^2.$$

> 😈 这两步公式是怎么来的?

The first term does not depend on w. Therefore:

$$\arg\max_{w} \log p(y \mid X, w) \iff \arg\min_{w} \sum_{i=1}^{n} (y_i - x_i^{\top} w)^2.$$

This is the **origin of Mean Squared Error (MSE)**. MSE is not “natural”; it is *Gaussian noise in disguise*.

Loss functions as noise models

This is the big conceptual jump many slides gloss over(略过不谈):

- Gaussian noise ⇒ squared loss
- Laplace noise(拉普拉斯噪声) ⇒ absolute loss (L1)
- Heavy-tailed noise ⇒ robust losses

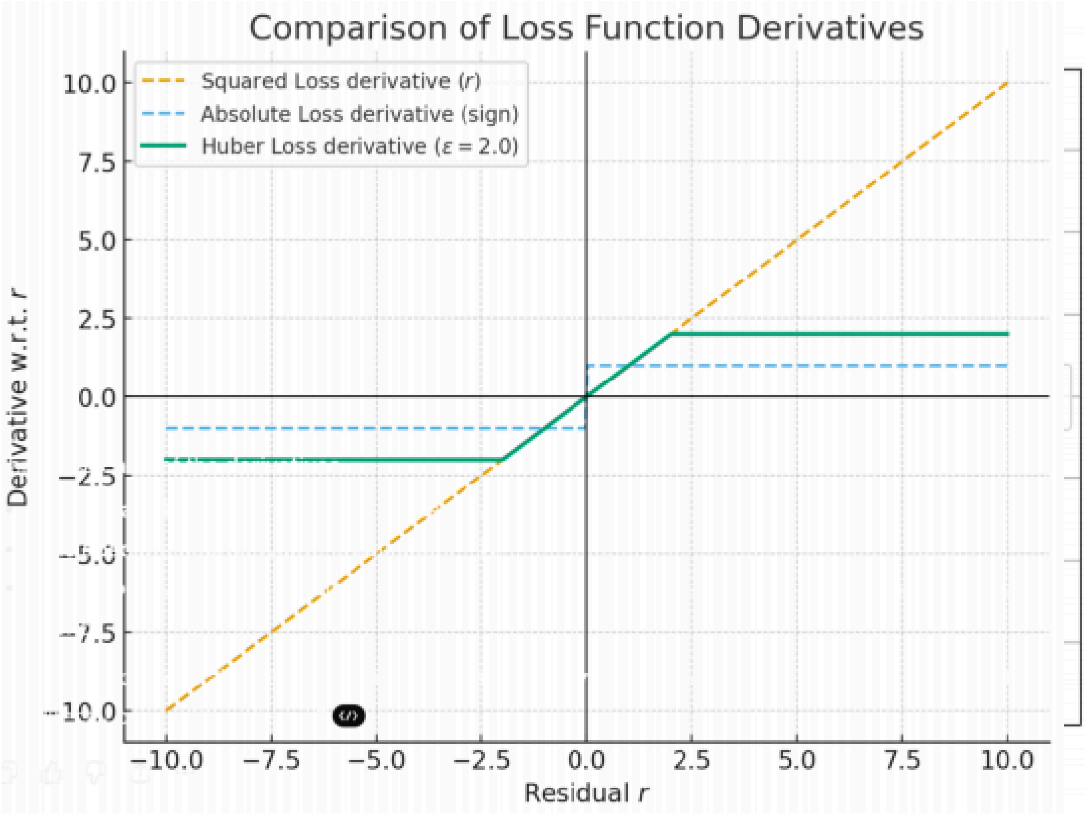
Choosing a loss function is choosing a **statistical assumption**, whether or not you admit it.

Why Huber loss exists

Squared loss is smooth but fragile: large residuals dominate(占据主导地位).

Absolute loss is robust but non-smooth at zero, which causes optimization headaches.

Huber loss is a compromise(妥协).



Define the residual

$$r_i = f_\beta(x_i) - y_i.$$

The Huber loss with parameter $\varepsilon > 0$ is

$$\ell_\varepsilon(r) = \begin{cases} \frac{1}{2}r^2, & |r| \le \varepsilon \\ \varepsilon\left(|r| - \frac{\varepsilon}{2}\right), & |r| > \varepsilon \end{cases}$$

Interpretation:

- Near zero: behaves like L2 (smooth, efficient)
- Far away: behaves like L1 (robust to outliers)

This is not magic. It is *deliberate curvature engineering*.

Empirical risk with Huber loss

Given data $X \in \mathbb{R}^{n\times p}$, $y \in \mathbb{R}^n$, define

$$L(\beta) = \sum_{i=1}^{n} \ell_\varepsilon(f_\beta(x_i) - y_i).$$

Assume a linear model:

$$f_\beta(x_i) = x_i^\top \beta.$$

So the residual is

$$r_i = x_i^\top \beta - y_i.$$

First-order derivative(一阶导数) of the Huber objective

First compute the derivative of the scalar Huber loss with respect to its argument r:

$$\ell'_\varepsilon(r) = \begin{cases} r, & |r| \le \varepsilon \\ \varepsilon \ \mathrm{sign}(r), & |r| > \varepsilon \end{cases}$$

Now apply the chain rule:

$$\nabla_\beta r_i = x_i.$$

So the gradient of the full objective is

$$\nabla_\beta L(\beta) = \sum_{i=1}^{n} \ell'_\varepsilon(r_i) x_i.$$

Explicitly:

$$\nabla_\beta L(\beta) = \sum_{i:|r_i| \le \varepsilon} r_i x_i + \sum_{i:|r_i| > \varepsilon} \varepsilon \ \mathrm{sign}(r_i) x_i.$$

Why Huber beats pure L1 in practice

From an optimization perspective:

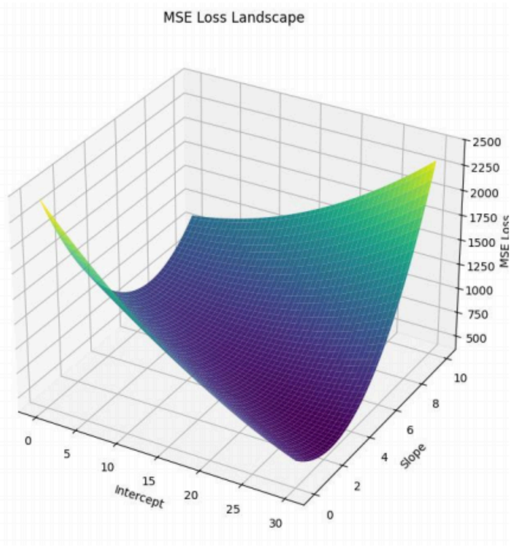
- L2: smooth but exploding gradients
- L1: bounded(有界) gradients but non-differentiable
- Huber: smooth *and* bounded

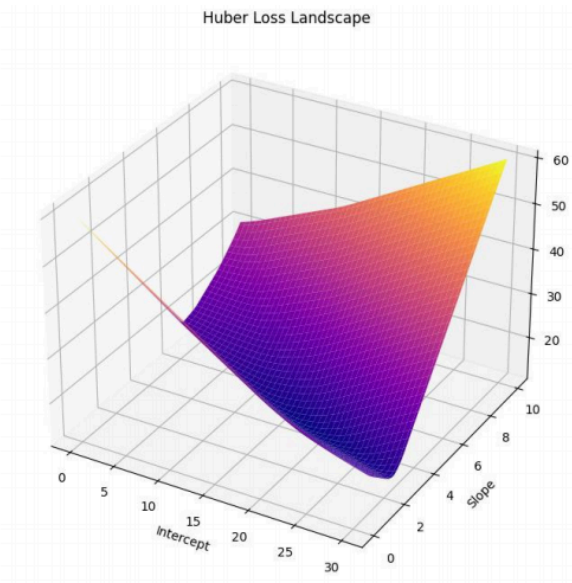
From a statistical perspective:

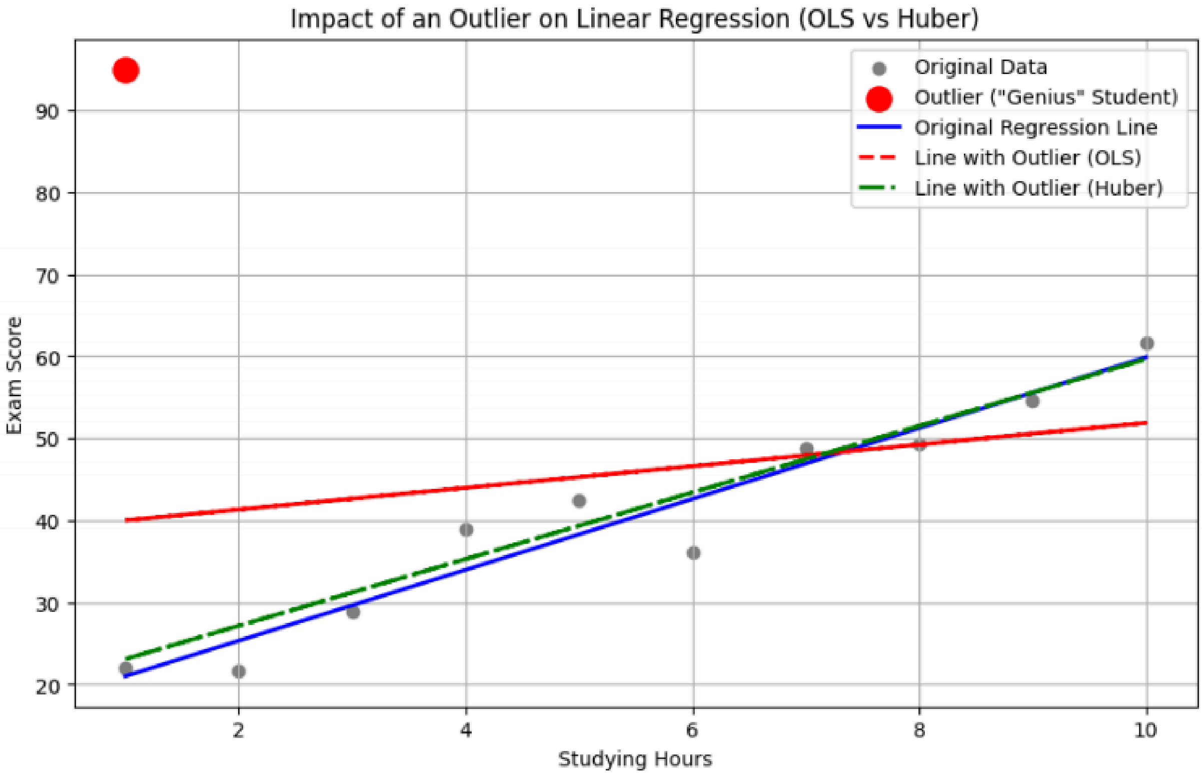
- Small errors are trusted
- Large errors are distrusted but not ignored

- Influence function is clipped

In robust statistics language, Huber loss implements **influence capping(影响函数截断)**.







Conceptual checkpoint (important)

At this point, notice something subtle but deep: There is **no closed-form solution** for Huber regression in general. The moment you leave pure L2, you also leave normal equations behind.

Multivariate Linear Regression

We begin by extending simple linear regression to multiple input dimensions. Instead of a single feature x, we now consider multiple features:

$$y = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \cdots + \beta_k x_k.$$

Geometrically:

- With one feature, the model is a line.
- With two features, the model becomes a plane(平面).
- With k features, the model is a hyperplane in $\mathbb{R}^{k+1}$.

Each additional feature introduces a new dimension in the input space.

Vectorized Form of Multivariate Regression

For a dataset with n samples and k features, predictions can be written as:

$$\hat{y}_i = \omega_0 + \omega_1 x_{i1} + \cdots + \omega_k x_{ik}, \quad i = 1, \ldots, n.$$

By augmenting each input with a constant 1 (to absorb the bias term), we define:

- Design matrix:

$$X = \begin{bmatrix} 1 & x_{01} & \cdots & x_{0k} \\ 1 & x_{11} & \cdots & x_{1k} \\ \vdots & \vdots & & \vdots \\ 1 & x_{n1} & \cdots & x_{nk} \end{bmatrix} \in \mathbb{R}^{n \times k}$$

- Parameter vector:

$$W = \begin{bmatrix} \omega_0 \\ \omega_1 \\ \vdots \\ \omega_k \end{bmatrix} \in \mathbb{R}^{k \times 1}$$

- Output vector:

$$Y = \begin{bmatrix} y_0 \\ y_1 \\ \vdots \\ y_n \end{bmatrix} \in \mathbb{R}^{n\times 1}$$

The entire system is compactly written as:

$$\hat{Y} = XW.$$

Linear Systems Viewpoint

The regression problem is equivalent to solving the linear system:

$$XW = Y.$$

However, the solvability depends on the relationship between:

- number of equations: n
- number of unknowns: k
- rank of X

This leads to three fundamentally different regimes(情形).

Overdetermined(超定) Systems ($n > k$)

When the number of data points exceeds the number of parameters:

$$n > k,$$

the system is overdetermined. In general:

- There is **no exact solution** satisfying $XW = Y$.
- The system is inconsistent due to noise.

The standard approach is **least squares**, minimizing:

$$||XW - Y||_2^2.$$

If X has full column rank, the solution is:

$$W = (X^\top X)^{-1} X^\top Y.$$

This is the classical **normal equation** solution.

Underdetermined(欠定) Systems ((n < k))

When there are fewer equations than unknowns:

$$n < k,$$

the system is underdetermined. In this case:

- There are **infinitely many exact solutions**.
- $X^\top X$ is singular and not invertible.
- The normal equation formula does not apply.

The question becomes: “Which solution should we choose?”

Moore–Penrose Pseudoinverse(伪逆)

The Moore–Penrose pseudoinverse X^+ provides a unified solution for **any matrix**, whether:

- square
- tall (overdetermined)
- wide (underdetermined)
- singular

It allows us to write:

$$W = X^+ Y$$

as a generalized inverse-based solution.

Special cases:

- If $n > k$ and X has full column rank(满列秩，即矩阵的**列向量线性无关**，秩等于列数**)**:

$$X^+ = (X^\top X)^{-1} X^\top$$

- If $n < k$ and X has full row rank:

$$X^+ = X^\top (X X^\top)^{-1}$$

General Solution for Underdetermined Systems

For underdetermined systems, the full solution set can be decomposed as:

$$W = W_p + W_h$$

where:

- $W_p = X^\top (XX^\top)^{-1} Y$ is a **particular solution(特解)**
- $W_h \in \mathrm{Null}(X)$ is any vector in the null space of X(零空间分量)

More explicitly:

$$W = X^\top (XX^\top)^{-1} Y - \big(I - X^\top (XX^\top)^{-1} X\big) z, \quad z \in \mathbb{R}^k$$

This shows that infinitely many solutions exist, parameterized by z.

Minimum-Norm Solution

Among all possible solutions, the pseudoinverse selects a very special one:

> The solution with minimum Euclidean norm.

That is:

$$W^* = \arg\min_{W} ||W||_2 \quad \text{s.t. } XW = Y.$$

This solution is exactly:

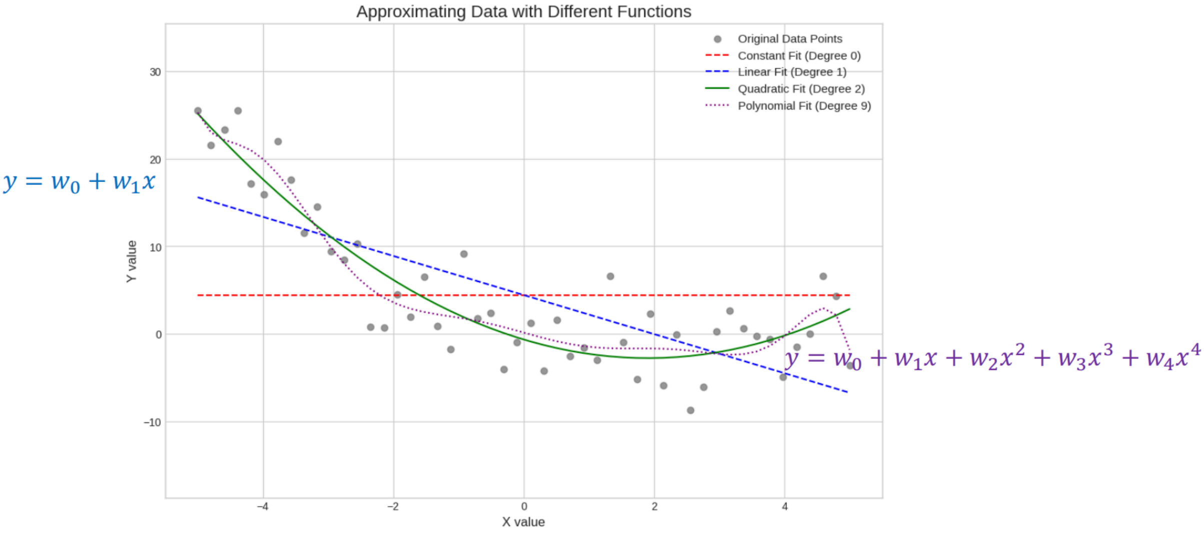
$$W^* = X^+ Y.$$

All other solutions differ by adding null-space components that increase the norm.

3 Optimization II

Fit, Gradient Descent and Learning Rate

Given a set of input-output pairs (x_i, y_i), we want to find a function $f(x)$ that captures the underlying trend. Different choices of function classes lead to very different behaviors: A constant model (degree 0) ignores input variation entirely. A linear model (degree 1) assumes a straight-line relationship. Higher-degree polynomials introduce curvature and can fit complex patterns. As model complexity increases, expressive power increases, but so does the risk of overfitting. This illustrates the fundamental **bias–variance tradeoff**.



$y = w_0 + w_1 x$

$y = w_0 + w_1 x + w_2 x^2 + w_3 x^3 + w_4 x^4$

Linear Models as Weighted Feature Aggregators

A linear model is best understood not as “a line”, but as a **linear combination of features**. We start from an input vector $x \in \mathbb{R}^d$. A feature mapping $\phi : \mathbb{R}^d \to \mathbb{R}^p$ transforms the input into a feature vector:

$$\phi(x) = (\phi_1(x), \phi_2(x), \ldots, \phi_p(x))$$

The model output is:

$$f(x) = \sum_{j=1}^{p} w_j \phi_j(x)$$

This is linear in parameters w, even if $\phi(x)$ itself is highly nonlinear. Stacking linear layers without nonlinearity does **not** increase expressiveness; the composition of linear functions is still linear.

Nonlinear Feature Mappings

To overcome the expressive limits of basic linear models, we introduce **nonlinear feature mappings**. Examples include polynomial features such as:

$$\phi(x) = (1, x_1, x_1 x_2, x_1^2 x_2^5, \ldots)$$

Or periodic features:

$$\phi(x) = (1, \sin(2\pi x), \sin(4\pi x), \ldots)$$

The key idea is simple but powerful: We keep the model linear in parameters, but nonlinear in input space.

Least Squares and the Closed-Form Solution

Using Mean Squared Error (MSE) as the loss function:

$$L(w) = ||Xw - Y||^2$$

The optimal solution, assuming $X^T X$ is invertible, is:

$$w = (X^T X)^{-1} X^T Y$$

However, this approach has serious limitations. Matrix inversion costs $O(n^3)$, which does not scale. If $X^T X$ is singular or ill-conditioned, the solution does not exist or is unstable. The method lacks flexibility for large-scale or streaming data.

Motivation for Iterative Optimization

To address these issues, we move to **iterative optimization methods**. Instead of solving for w in one step, we start from an initial guess and gradually improve it. At iteration t, we update:

$$w^{(t+1)} = w^{(t)} + \mu v$$

Here:

- μ is the step size (learning rate)
- v is the update direction

This reframes learning as a controlled search in parameter space.

Choosing the Descent Direction

To minimize the loss, we want to move in the direction of steepest decrease. In one dimension, this leads to:

- If $L'(w) < 0$, increasing w decreases the loss.
- If $L'(w) > 0$, decreasing w decreases the loss.

Thus, the descent direction is:

$$v = -\text{sign}(L'(w))$$

In higher dimensions, this generalizes to the **negative gradient**.

Step Size and Gradient Descent

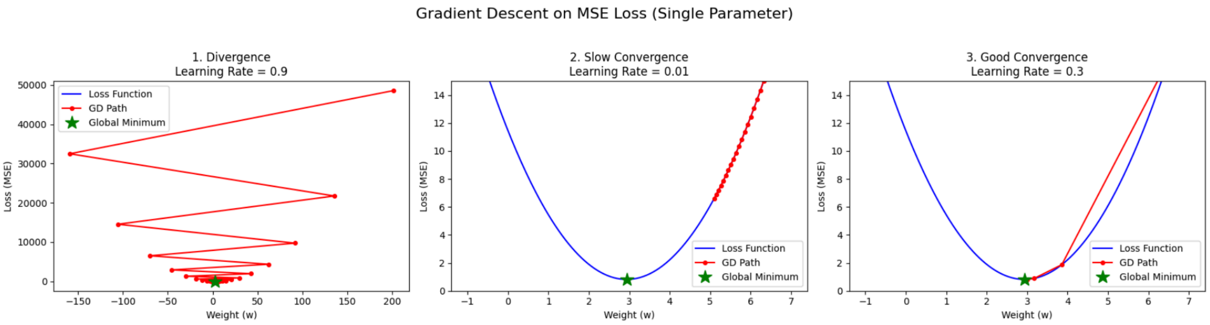
A naive fixed step size can cause divergence or slow convergence. To stabilize updates, we normalize by gradient magnitude:

$$w^{(t+1)} = w^{(t)} - \alpha \nabla L(w^{(t)})$$

This is the standard **gradient descent update rule**.

- Choosing α is critical:
- Too large leads to divergence.
- Too small leads to painfully slow convergence.

The plots clearly show divergence, slow convergence, and well-behaved convergence depending on the learning rate.



When to Stop Iterating

Since iterative methods do not have a natural endpoint, stopping criteria must be defined. Common conditions include:

- The gradient norm is below a threshold.
- The number of iterations exceeds a maximum limit.
- The improvement in loss between iterations is sufficiently small.

Stopping too early leads to underfitting; stopping too late wastes computation and may overfit.

From 1D to High Dimensions

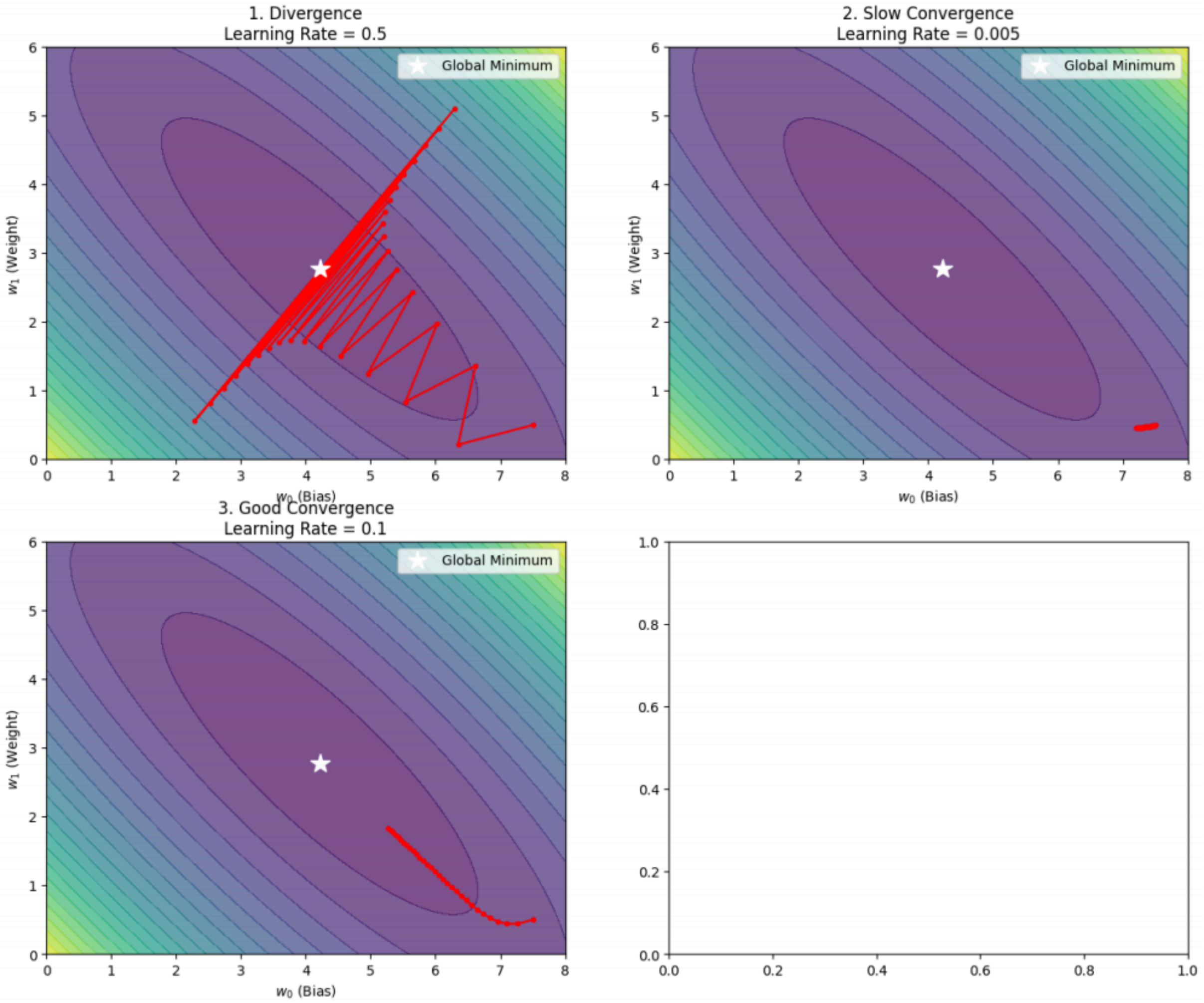
In one-dimensional optimization, choosing a descent direction is trivial: there are only two options, left or right. In higher-dimensional parameter spaces, however, we face infinitely many possible directions. Despite this increase in complexity, the core intuition remains unchanged: **At each iteration, we want to move in a direction that reduces the loss function most efficiently.** Thus, the same three questions still govern the algorithm:

- Which direction should we move?
- How far should we move?
- When should we stop?

The challenge lies entirely in defining "direction" when the parameter vector w lives in $\mathbb{R}^n$.

Step Size Behavior in Multiple Dimensions

When visualizing gradient descent in two dimensions, the loss surface becomes a contour(等高线) map. Each contour line represents points with equal loss value. Different learning rates lead to qualitatively different behaviors: **A large learning rate may overshoot the minimum and cause divergence. A very small learning rate results in slow convergence. A well-chosen learning rate allows smooth and stable descent toward the minimum.** These behaviors mirror the one-dimensional case, but the geometry makes the consequences more visible.



Steepest Descent as a Geometric Principle

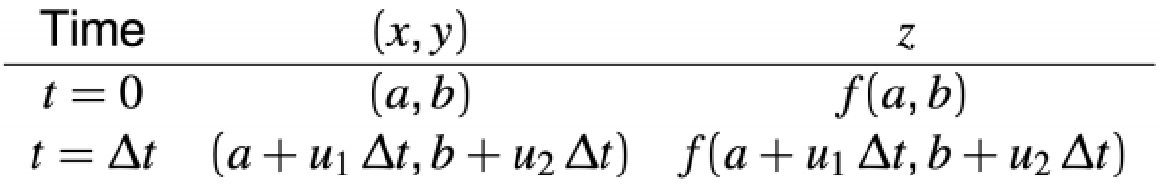
At any point on the loss surface, there exists a direction in which the loss decreases most rapidly. This direction is known as the **steepest descent direction**. Geometrically, the steepest descent direction is always orthogonal(正交) to the contour lines of the loss function. This is not an arbitrary choice; it follows directly from the geometry of level sets. The optimization path produced by gradient descent therefore cuts across contour lines at right angles, instead of sliding along them.

Directional Change and the Gradient

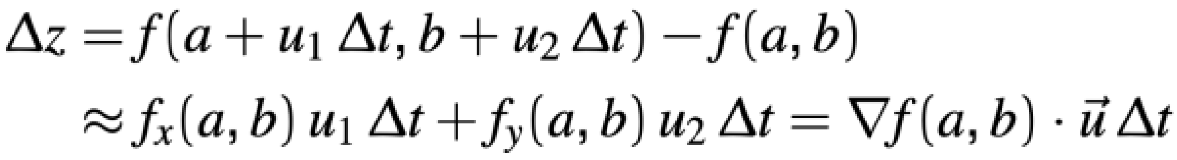
To formalize the notion of "which direction decreases the function fastest," we introduce the idea of **directional derivatives**. Let $f(x, y)$ be a scalar-valued function and let $\mathbf{u}$ be a unit vector in the plane(平面). We examine how the function value changes if we move a small distance in direction $\mathbf{u}$. Using a first-order Taylor approximation:

$$\Delta z \approx \nabla f(a, b) \cdot \mathbf{u}, \Delta t$$

This shows that the rate of change of the function along direction $\mathbf{u}$ depends on the dot product between the gradient and the direction vector.



Time	(x, y)	z
$t = 0$	(a, b)	$f(a, b)$
$t = \Delta t$	$(a + u_1 \Delta t, b + u_2 \Delta t)$	$f(a + u_1 \Delta t, b + u_2 \Delta t)$



$$\begin{aligned}\Delta z &= f(a + u_1 \Delta t, b + u_2 \Delta t) - f(a, b) \\ &\approx f_x(a, b)\, u_1 \Delta t + f_y(a, b)\, u_2 \Delta t = \nabla f(a, b) \cdot \vec{u}\, \Delta t\end{aligned}$$

Directional Derivative

Formally, the directional derivative of f at point $\mathbf{x}$ in direction $\mathbf{u}$ is defined as:

$$D_{\mathbf{u}} f(\mathbf{x}) = \nabla f(\mathbf{x}) \cdot \mathbf{u}$$

This quantity measures how fast the function increases or decreases when moving along $\mathbf{u}$. If the dot product is positive, the function increases. If it is negative, the function decreases.

Why the Gradient Points to Steepest Ascent

Since $\mathbf{u}$ is a unit vector, the directional derivative satisfies:

$$D_{\mathbf{u}} f = ||\nabla f|| \cos(\theta)$$

where θ is the angle between ∇f and $\mathbf{u}$. As $\mathbf{u}$ varies, $\cos(\theta)$ ranges between -1 and 1. Therefore, the maximum possible directional derivative is $||\nabla f||$, achieved when $\mathbf{u}$ points in the same direction as the gradient. This proves that: The gradient vector points in the direction of **steepest ascent**. Its negative points in the direction of **steepest descent**.

The Final Answer: Gradient Descent Direction

To decrease the loss function as fast as possible at the current point, we should choose:

$$\mathbf{u} = -\frac{\nabla f}{|\nabla f|}$$

In practice, the normalization is absorbed into the learning rate, leading to the standard update rule:

$$w^{(t+1)} = w^{(t)} - \alpha \nabla L(w^{(t)})$$

This is not a heuristic(启发式). It is a direct consequence of multivariable calculus and geometry.

Convergence

Convergence of Gradient Descent for Linear Regression

We consider linear regression with mean squared error loss:

$$J(w) = \frac{1}{2m}(Xw - y)^T(Xw - y)$$

This objective is a **convex quadratic function(凸二次函数)**, which makes it ideal for convergence analysis. The gradient is:

$$\nabla J(w) = \frac{1}{m}X^T(Xw - y)$$

Gradient descent updates the parameters as:

$$w_{k+1} = w_k - \alpha \nabla J(w_k)$$

The global minimizer w^* satisfies:

$$\nabla J(w^*) = 0 \quad \Rightarrow \quad w^* = (X^TX)^{-1}X^Ty$$

Error Dynamics and Linear Recurrence(递推)

Define the error vector:

$$e_k = w_k - w^*$$

Substituting(代入) the update rule and using $X^Ty = X^TXw^*$, we obtain:

$$e_{k+1} = \left(I - \frac{\alpha}{m}X^TX\right)e_k$$

This shows that gradient descent behaves like a **linear dynamical system** in the error space. Convergence depends entirely on the spectral properties(谱性质) of the matrix:

$$I - \frac{\alpha}{m} X^T X$$

Spectral Norm(谱范数) and Convergence Condition

Using the spectral (operator) norm:

$$||A||_2 = \sup_{z \neq 0} \frac{||Az||_2}{||z||_2}$$

We obtain the bound:

$$||e_{k+1}||_2 \leq \left|\left|I - \frac{\alpha}{m} X^T X\right|\right|_2 ||e_k||_2$$

For convergence, we require:

$$|1 - \alpha \lambda_i| < 1 \quad \text{for all eigenvalues(特征值) } \lambda_i \text{ of } \frac{1}{m} X^T X$$

This yields the fundamental step-size condition(步长条件):

$$0 < \alpha < \frac{2}{\lambda_{\max}}$$

This is not a heuristic—it is a **necessary and sufficient condition** for convergence in this quadratic case.

Conditioning and Convergence Speed

Even when gradient descent converges, it may do so very slowly. If the eigenvalues of $X^T X$ vary widely (poor conditioning), the loss contours(等高线) are highly elongated(拉长). Gradient descent then zig-zags across narrow valleys(狭窄山谷) instead of moving directly to the minimum. Well-conditioned problems converge quickly. Ill-conditioned problems converge slowly, even with an optimal step size. This explains why learning rate tuning alone often cannot fix slow convergence.

How to Choose a Better Step Size

A fixed learning rate is fundamentally limited because it ignores local curvature. One remedy(补救方法) is **line search**, which adapts the step size dynamically at each iteration. The goal is not to find the perfect step, but a step that guarantees sufficient decrease in the objective.

Line Search and Armijo–Goldstein Condition

Given a descent direction $-\nabla f(w_t)$, the Armijo–Goldstein condition requires:

$$J(w_t - \alpha\nabla f(w_t)) \leq J(w_t) - \frac{1}{2}\alpha||\nabla f(w_t)||^2$$

Interpretation(解释): The actual decrease in loss must be at least a fixed fraction(固定比例) of the predicted first-order decrease.

In practice: Start with a candidate α. If the condition fails, reduce α (often by halving). Repeat until the condition is satisfied. This guarantees stability without knowing the Hessian.

Newton’s Method: Second-Order Optimization

Gradient descent uses only first-order information. Newton’s method exploits **second-order structure**. Using a second-order Taylor expansion around x_t:

$$Q_t(x) = f(x_t) + \nabla f(x_t)^T(x - x_t) + \frac{1}{2}(x - x_t)^T H(x_t)(x - x_t)$$

Here, $H(x_t)$ is the Hessian matrix. The next iterate is defined as:

$$x_{t+1} = \arg\min_x Q_t(x) \quad \Rightarrow \quad x_{t+1} = x_t - [H(x_t)]^{-1}\nabla f(x_t)$$

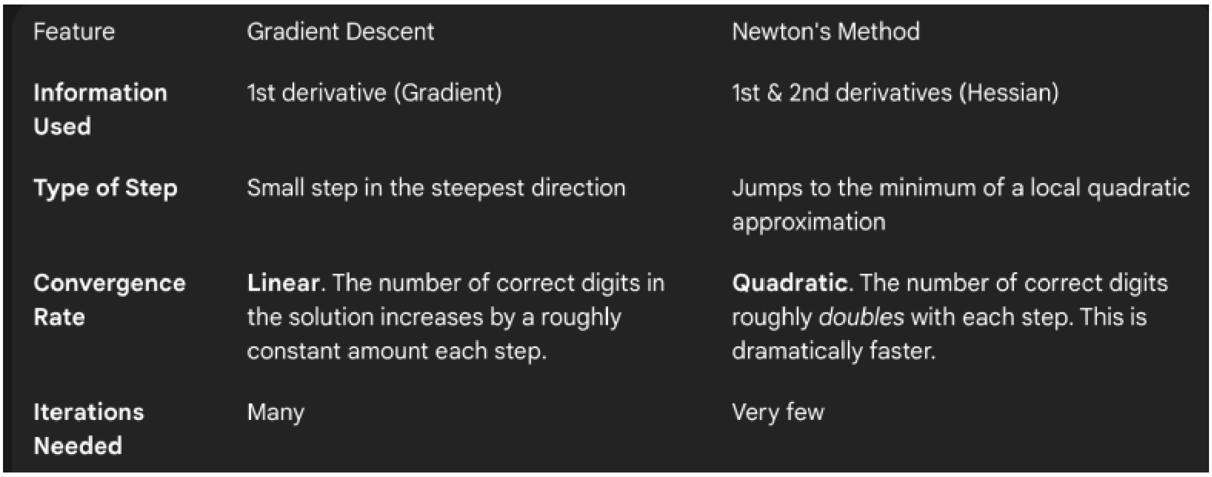
This step jumps directly to the minimum of the local quadratic approximation.

Newton’s method requires strong assumptions:

- The function must be continuous C^0,
- Differentiable C^1,
- And twice differentiable C^2.

Not all continuous functions are differentiable. All differentiable functions are continuous. These distinctions matter for optimization guarantees.

Gradient Descent vs. Newton’s Method



Feature	Gradient Descent	Newton's Method
Information Used	1st derivative (Gradient)	1st & 2nd derivatives (Hessian)
Type of Step	Small step in the steepest direction	Jumps to the minimum of a local quadratic approximation
Convergence Rate	**Linear**. The number of correct digits in the solution increases by a roughly constant amount each step.	**Quadratic**. The number of correct digits roughly *doubles* with each step. This is dramatically faster.
Iterations Needed	Many	Very few

From Gradient Descent to Mini-batch SGD

What Is Wrong with Vanilla Gradient Descent?

Standard (batch) Gradient Descent computes the gradient using **all training examples** before each parameter update. At iteration t, the update rule is:

$$w_{t+1} = w_t - \alpha \frac{1}{m} \sum_{i=1}^{m} \nabla \ell(w_t; x_i, y_i)$$

This has two nice theoretical properties:

- The update direction is smooth and deterministic.
- For convex objectives, convergence to the global minimum is guaranteed.

However, the major drawback is computational: **Each update requires a full pass over the dataset.** When m is large, a single iteration becomes extremely expensive. As a result, batch gradient descent is **impractical for large-scale learning**, even though it looks mathematically clean.

Stochastic(随机) Gradient Descent (SGD)

Stochastic Gradient Descent addresses this bottleneck by updating parameters using **only one training example at a time**. For each epoch, the dataset is shuffled, and for each example:

$$w \leftarrow w - \alpha \nabla \ell(w; x_i, y_i)$$

This drastically(大幅度地) reduces the cost per update and allows learning to begin immediately. The key conceptual shift is: We replace the exact gradient with a **noisy**

but unbiased estimate. This introduces variance into the optimization process. The trajectory becomes noisy and oscillatory(震荡的) rather than smooth. In theory, SGD converges more slowly in terms of objective value per iteration, but in practice it often makes faster progress per unit time.

Mini-batch Stochastic Gradient Descent

Mini-batch SGD interpolates between batch GD and pure SGD. Instead of using one example or the full dataset, gradients are computed over a small batch of size B:

$$w \leftarrow w - \alpha \frac{1}{B} \sum_{(x_i, y_i) \in \text{batch}} \nabla \ell(w; x_i, y_i)$$

This achieves several goals simultaneously:

- Variance of the gradient estimate is reduced compared to SGD.
- Computation can be vectorized and parallelized efficiently.
- The algorithm maps naturally to modern hardware such as GPUs and TPUs.

Mini-batch SGD is therefore not a compromise—it is an **engineering optimum**.

Accuracy–Speed Trade-off

The choice of optimization method reflects a fundamental trade-off. Batch GD prioritizes accuracy of each update but sacrifices speed. SGD prioritizes speed and scalability but sacrifices smoothness. Mini-batch SGD balances both by controlling the noise level through batch size. In practice, **increasing batch size reduces gradient noise but increases computation per update.** Decreasing batch size does the opposite. There is no universally optimal batch size; it depends on the model, data, and hardware.

In modern machine learning papers, the term "SGD" almost always refers to **mini-batch SGD**, not true single-sample SGD. This is a linguistic shortcut, not a theoretical statement. Failing to notice this causes confusion when reading papers or reproducing results.

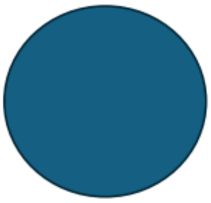
4 Optimization II

Convex Sets, Convex Functions, and Convexity

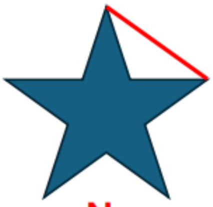
A set $X \subseteq \mathbb{R}^n$ is **convex** if for any two points $x_1, x_2 \in X$, the entire line segment between them stays inside the set:

$$\forall \lambda \in [0,1], \quad \lambda x_1 + (1-\lambda) x_2 \in X.$$

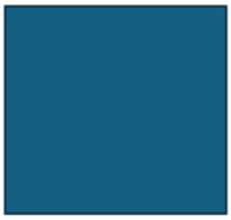
Exercise



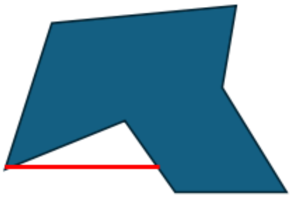
Yes



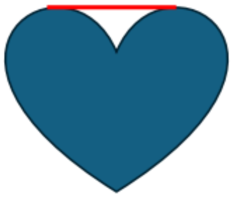
No



Yes



No

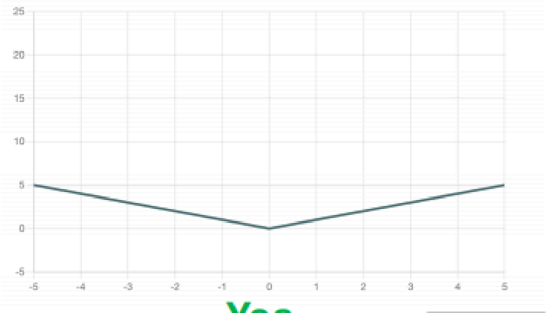


No

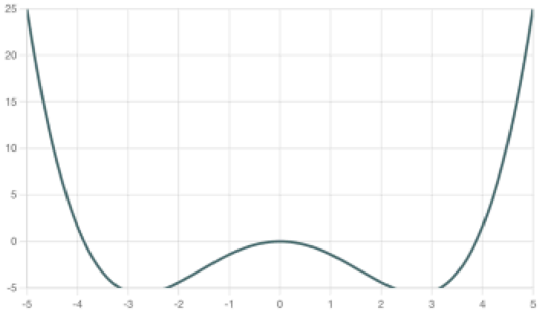
A function $f : X \to \mathbb{R}$ (with **convex domain** X) is **convex** if the function value at any point on the segment between x_1 and x_2 is no larger than the linear interpolation of the endpoint values:

$$\forall x_1, x_2 \in X, \ \forall \lambda \in [0,1], \quad f(\lambda x_1 + (1-\lambda) x_2) \le \lambda f(x_1) + (1-\lambda) f(x_2).$$

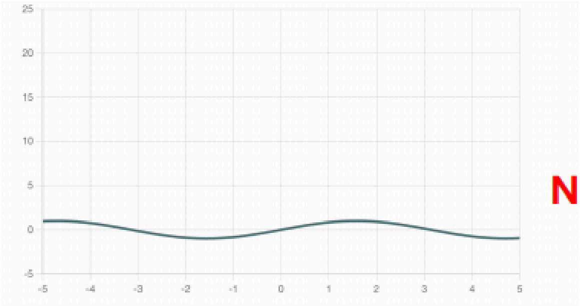
Exercise



Yes



No



No

For **convex minimization** (minimize a convex function over a convex set), convexity implies: **Any local minimum is also a global minimum.** This is the key reason convex problems are “nice”: you don’t get fake local valleys that trap you away from the best solution. Dually(对偶地), for **maximization**, concave functions(凹函数) have the analogous “local max = global max” behavior.

Let (X, d_X) be a metric space and $f : X \to \mathbb{R}$.

- **Global minimum:** $x^\star \in X$ is a global minimum if

$$\forall x \in X, \quad f(x^\star) \le f(x).$$

- **Local minimum:** $x^\star \in X$ is a local minimum if

$$\exists \varepsilon > 0 \text{ s.t. } \forall x \in X,\ d_X(x, x^\star) < \varepsilon \Rightarrow f(x^\star) \le f(x).$$

The “local = global” guarantee is exactly what convexity buys you for minimization.

Equivalent Conditions for Convexity (0th / 1st / 2nd Order)

(A) 0th-order condition (Definition / Jensen form)

$$f(\lambda x_1 + (1-\lambda)x_2) \le \lambda f(x_1) + (1-\lambda) f(x_2).$$

This is the core definition and does not require derivatives.

(B) 1st-order condition (Supporting hyperplane)

If f is differentiable, convexity is equivalent to:

$$f(x_2) \ge f(x_1) + \nabla f(x_1)^\top (x_2 - x_1), \quad \forall x_1, x_2 \in X.$$

Meaning: the tangent plane(切平面) at x_1 is a **global underestimator** of f.

(C) 2nd-order condition (Hessian test)

If f is twice differentiable, convexity is equivalent to:

$$\nabla^2 f(x) \succeq 0 \quad \text{for all } x \in X,$$

i.e., the Hessian is **positive semidefinite(半正定，PSD)** everywhere.

$f(x,y) = 2x^2 + 3y^2$	$f(x,y) = e^x$	$f(x,y) = -(x^2 + y^2)$
convex	convex	Concave
$f(x,y) = x^3$	$f(x) = log(x)$	$f(x,y) = x^2 - y^2$
Neither	Concave	Neither

The “Hidden Rule” of Convex Functions

The Hidden Rule: A Convex Function Requires a Convex Domain

In convex optimization, we usually treat “f is convex” as shorthand(简称) for:

1. **Domain is convex**: $\mathrm{dom}(f)$ is a convex set
2. **Chord inequality holds(弦不等式成立)**:

$$f(\lambda x_1 + (1-\lambda)x_2) \;\le\; \lambda f(x_1) + (1-\lambda)f(x_2), \\ \forall x_1, x_2 \in \mathrm{dom}(f),\ \forall \lambda \in [0,1].$$

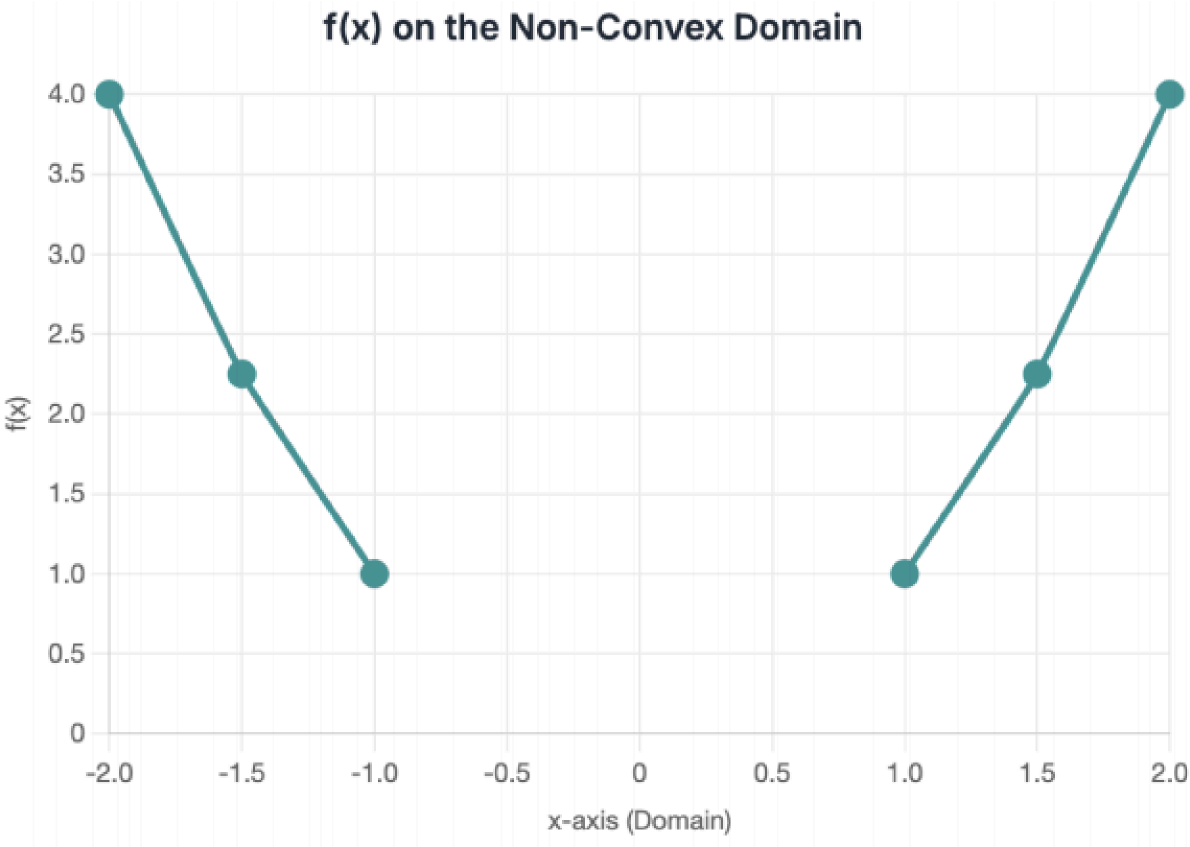
Why the domain matters: If the domain has a “hole” or is disconnected, the point $\lambda x_1 + (1-\lambda)x_2$ may fall outside the domain, so the convexity inequality(凸性不等式) becomes meaningless under the standard definition. Even if the formula looks convex, the *optimization problem* can become non-convex because the feasible region(可行域) is non-convex.

Case Study 1: Convex Formula on a Disconnected Domain(不连通域)

Consider $f(x) = x^2$, which is convex on $\mathbb{R}$. Now restrict the domain to a disconnected set, e.g.

$$D = [-2, -1] \cup [1, 2].$$

Pick $x_1 = -1$ and $x_2 = 1$. Their midpoint is 0, but $0 \notin D$. So the line segment between two feasible points is not fully feasible → **the domain is not convex** → under the standard convex-analysis setup, we do **not** call f convex on that domain.



Case Study 2: A Convex Paraboloid(抛物面) on a “Donut(甜甜圈)” (Annulus(环形)) Domain

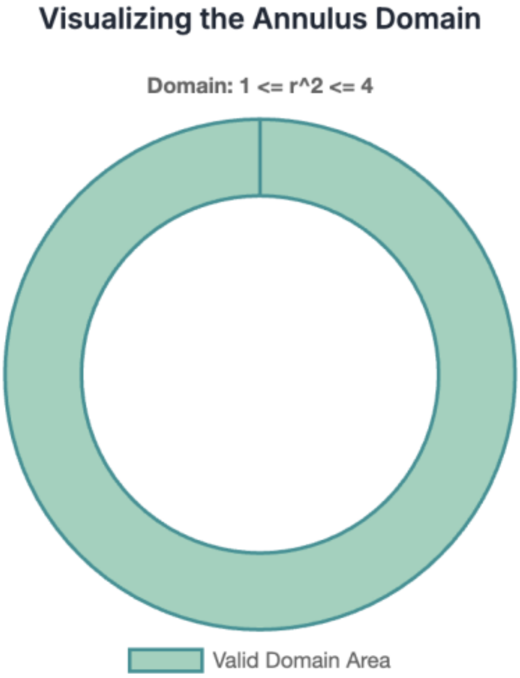
Let

$$f(x, y) = x^2 + y^2$$

(convex on $\mathbb{R}^2$), but restrict the domain to an annulus:

$$D = (x, y) : 1 \leq x^2 + y^2 \leq 4.$$

This set is **not convex** because a segment connecting two points on opposite sides typically passes through the center hole, which is excluded. Therefore, the function *as an optimization object on that domain* is not convex in the standard sense (again: **package deal**).



Closure Properties: How to Build New Convex Functions

Positive Linear Combination

If f and g are convex on a convex set X, then for any $\alpha, \beta > 0$:

$$h(x) = \alpha f(x) + \beta g(x)$$

is convex on X. (Actually $\alpha, \beta \geq 0$ is enough; strict positivity isn't required for convexity.)

Maximum of Convex Functions

If f and g are convex on X, then

$$h(x) = \max\{f(x), g(x)\}$$

is convex on X. This is a key trick: many "piecewise(分段)" convex functions are max-of-affines(仿射函数), hence convex.

Composition Rule

Let $f \circ g = f(g(x))$.

- If g is **affine** and f is convex, then $f \circ g$ is convex.
- More generally, if f is **convex and non-decreasing** (monotone increasing(单调递增)) and g is convex, then $f \circ g$ is convex.

This monotonicity condition is not decoration; it prevents the composition(复合) from flipping curvature(曲率反转).

e.g. Square loss:

$$\ell(y, f_w(x)) = (y - f_w(x))^2, \qquad L(w) = \sum_i \ell(y_i, f_w(x_i)).$$

- **(a) "$L(w)$ is convex" — False in general.**

  If $f_w(x)$ is a nonlinear model in w, the squared error can become non-convex.

- **(b) "$L(w)$ is convex for $f_w(x) = w^\top x$" — True.**

  Then each term is $(y_i - w^\top x_i)^2$, a convex quadratic in w. Sums of convex functions are convex.

- **(c) "$L(w)$ is convex for $f_w(x) = \mathrm{ReLU}(w^\top x) = \max(0, w^\top x)$" — Not true in general.**

The key subtlety(微妙之处): the loss is $(y - \max(0, a))^2$ with $a = w^\top x$. This function is piecewise:

- constant $= y^2$ when $a \leq 0$,
- quadratic $(y - a)^2$ when $a \geq 0$,

  and the "kink(折点)" at $a = 0$ is convex **only for certain values of** y. For common cases like $y = 1$, the slope jumps the wrong way → the function is not convex.

e.g.

1. **"All convex functions have only one local minimum that is a global minimum."**

   False. Convex functions can have **multiple** global minimizers (e.g., a flat bottom).

   > 😈 A more precise statement would be: **If a convex function has a local minimizer, then every local minimizer is also a global minimizer.**

   > 😈 If a function fff has the property that **every local minimum is also a global minimum**, does that necessarily mean fff is convex? ❌ **Not necessarily.**
   > Consider a **non-convex** function that still has no "spurious(伪)" local minima: $f(x) = x^3$. For this function, $f''(x) = 6x$, which is **negative** for $x < 0$ and **positive** for $x > 0$. Thus, the function is **concave(凹) on the left** and **convex on the right**, so overall it is **not convex**.
   > However, $f(x) = x^3$ has **no local minima or maxima at all**.
   > Therefore, the statement "every local minimum is a global minimum" is **vacuously true** (a *vacuum truth*(空真命题)) — it holds simply because there are **no local minima** to begin with.

2. **"A strictly convex function on all of $\mathbb{R}$ is guaranteed to have one unique global minimum."**

   False as stated. Strict convexity gives **uniqueness if a minimizer exists**, but existence is not guaranteed (e.g., a strictly convex function may approach an infimum(下确界) without attaining it).

3. **“For differentiable f, $\nabla f(x^*) = 0$ is sufficient to prove x^* is a global minimum.”**

   False in general. This becomes **true if f is convex** (and differentiable), but without convexity a stationary point can be a max or saddle.

Takeaway Geometry: Stationary Points, Local Minima, Global Minima

- **Stationary point(驻点):** $\nabla f(x^*) = 0$ (flat slope).
- **Local minimum:** $f(x^*) \le f(x)$ for all x in a neighborhood of x^*.
- **Global minimum:** $f(x^*) \le f(x)$ for all x in the entire domain.

In general:

$$\text{Global minima} \subseteq \text{Local minima} \subseteq \text{Stationary points (for smooth } f\text{)}.$$

Convexity collapses the hierarchy (for differentiable convex f):

$$\nabla f(x^*) = 0 \quad \iff \quad x^* \text{ is a global minimizer.}$$

Stationary Point vs Saddle Point

A **saddle point** is a stationary point that is neither a local min nor a local max: the function curves up in some directions and down in others. Intuitively, it looks like a mountain pass: “minimum along one axis, maximum along another.” This is why “gradient = 0” alone is not enough unless convexity (or other stronger structure) is present.

Convergence of Gradient Descent

Problem Setup

We want to solve the unconstrained optimization problem

$$\min_{x \in \mathbb{R}^n} f(x)$$

using Gradient Descent (GD):

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

where $\alpha > 0$ is the step size and x^* denotes a global minimizer. Define the **function error**

$$e_k := f(x_k) - f(x^*).$$

Core Assumptions

Convexity

A differentiable function f is convex if for all x, y:

$$f(y) \geq f(x) + \nabla f(x)^\top (y - x).$$

This implies (for minimization) that local minima are global minima, and the tangent plane(切平面) is a global under-estimator(全局下估计).

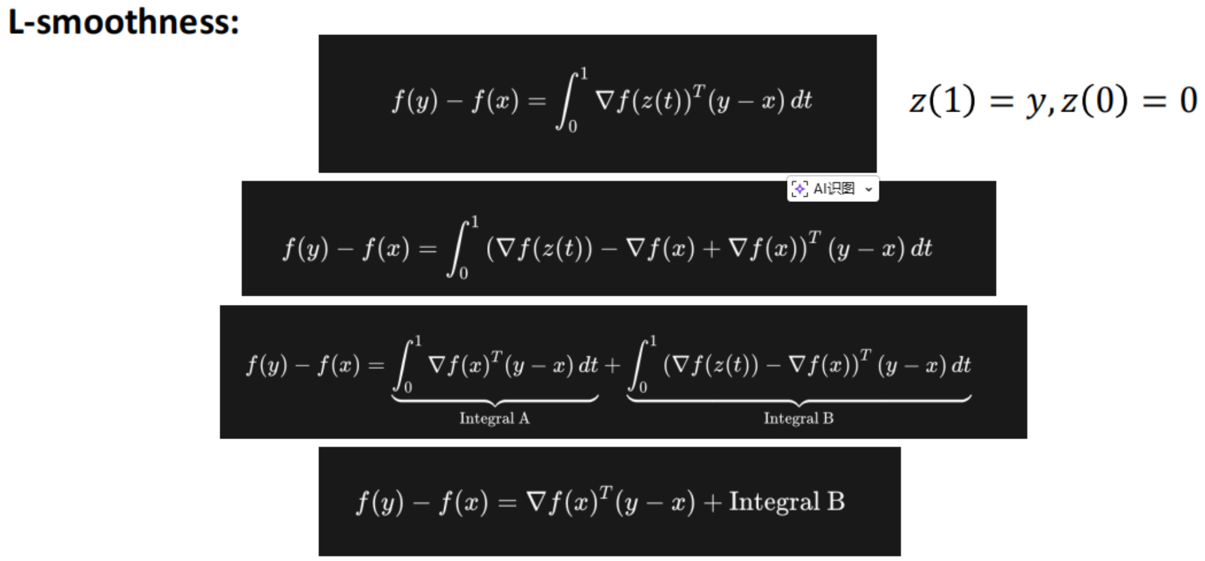
L-Smoothness (Lipschitz Gradient)

f is **L-smooth** if its gradient is Lipschitz:

$$||\nabla f(x) - \nabla f(y)|| \leq L||x - y||.$$

A key equivalent inequality(等价不等式) (often called the **Descent Lemma(下降引理)**) is:

$$f(y) \leq f(x) + \nabla f(x)^\top (y - x) + \frac{L}{2}||y - x||^2.$$



L-smoothness:

$$f(y) - f(x) = \int_0^1 \nabla f(z(t))^T (y - x)\, dt$$

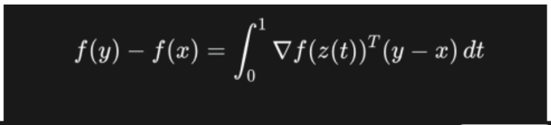
$z(1) = y, z(0) = 0$

$$f(y) - f(x) = \int_0^1 \left(\nabla f(z(t)) - \nabla f(x) + \nabla f(x)\right)^T (y - x)\, dt$$

$$f(y) - f(x) = \underbrace{\int_0^1 \nabla f(x)^T (y - x)\, dt}_{\text{Integral A}} + \underbrace{\int_0^1 \left(\nabla f(z(t)) - \nabla f(x)\right)^T (y - x)\, dt}_{\text{Integral B}}$$

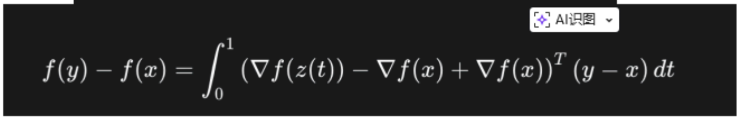
$$f(y) - f(x) = \nabla f(x)^T (y - x) + \text{Integral B}$$

L-smoothness:

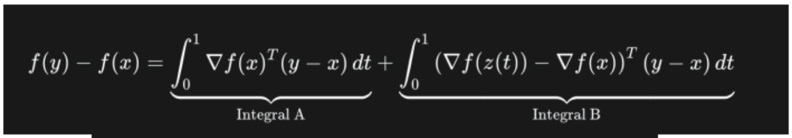


$$f(y) - f(x) = \int_0^1 \nabla f(z(t))^T (y - x)\, dt$$

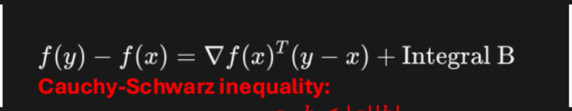
$z(1) = y, z(0) = 0$



$$f(y) - f(x) = \int_0^1 \left(\nabla f(z(t)) - \nabla f(x) + \nabla f(x)\right)^T (y - x)\, dt$$



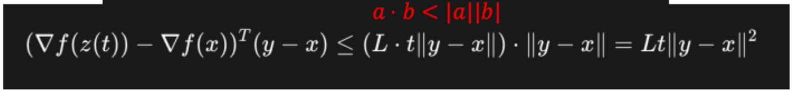
$$f(y) - f(x) = \underbrace{\int_0^1 \nabla f(x)^T (y - x)\, dt}_{\text{Integral A}} + \underbrace{\int_0^1 \left(\nabla f(z(t)) - \nabla f(x)\right)^T (y - x)\, dt}_{\text{Integral B}}$$



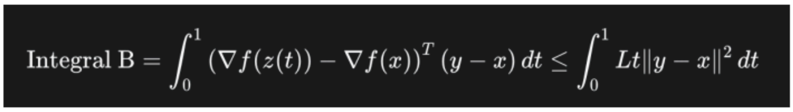
$$f(y) - f(x) = \nabla f(x)^T (y - x) + \text{Integral B}$$

Cauchy-Schwarz inequality:

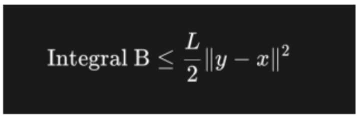
$a \cdot b < |a||b|$



$$(\nabla f(z(t)) - \nabla f(x))^T (y - x) \leq (L \cdot t\|y - x\|) \cdot \|y - x\| = Lt\|y - x\|^2$$



$$\text{Integral B} = \int_0^1 \left(\nabla f(z(t)) - \nabla f(x)\right)^T (y - x)\, dt \leq \int_0^1 Lt\|y - x\|^2\, dt$$

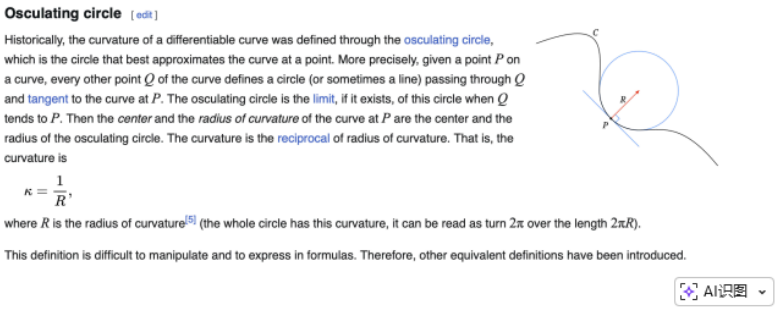


$$\text{Integral B} \leq \frac{L}{2}\|y - x\|^2$$

Strong Convexity

$$\overbrace{\underbrace{f(x)+\nabla f(x)^T(y-x)+\frac{\mu}{2}\|y-x\|^2}_{\text{Lower Bound (Floor)}}}^{1^{\text{st}}\text{ order condition}} \le f(y) \le \underbrace{f(x)+\nabla f(x)^T(y-x)+\frac{L}{2}\|y-x\|^2}_{\text{Upper Bound (Ceiling)}}$$

2^{nd} order condition $\nabla^2 f(x) \ge \mu I$, positive curvature at every x



Osculating circle [edit]

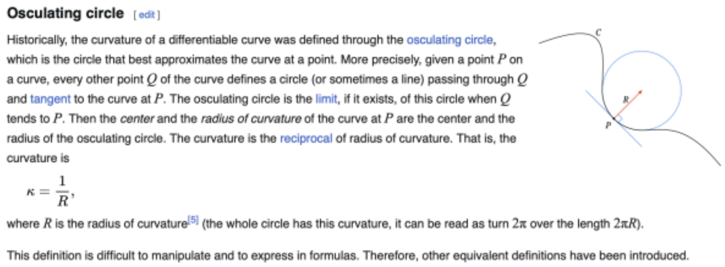
Historically, the curvature of a differentiable curve was defined through the osculating circle, which is the circle that best approximates the curve at a point. More precisely, given a point P on a curve, every other point Q of the curve defines a circle (or sometimes a line) passing through Q and tangent to the curve at P. The osculating circle is the limit, if it exists, of this circle when Q tends to P. Then the *center* and the *radius of curvature* of the curve at P are the center and the radius of the osculating circle. The curvature is the reciprocal of radius of curvature. That is, the curvature is

$$\kappa = \frac{1}{R},$$

where R is the radius of curvature[5] (the whole circle has this curvature, it can be read as turn 2π over the length $2\pi R$).

This definition is difficult to manipulate and to express in formulas. Therefore, other equivalent definitions have been introduced.

Strong C



Osculating circle [edit]

Historically, the curvature of a differentiable curve was defined through the osculating circle, which is the circle that best approximates the curve at a point. More precisely, given a point P on a curve, every other point Q of the curve defines a circle (or sometimes a line) passing through Q and tangent to the curve at P. The osculating circle is the limit, if it exists, of this circle when Q tends to P. Then the *center* and the *radius of curvature* of the curve at P are the center and the radius of the osculating circle. The curvature is the reciprocal of radius of curvature. That is, the curvature is

$$\kappa = \frac{1}{R},$$

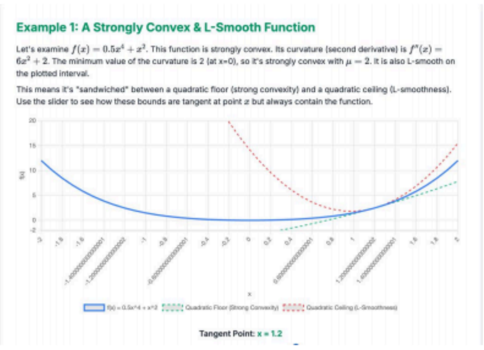
where R is the radius of curvature[5] (the whole circle has this curvature, it can be read as turn 2π over the length $2\pi R$).

This definition is difficult to manipulate and to express in formulas. Therefore, other equivalent definitions have been introduced.

$+\frac{L}{2}\|y-x\|^2$

ing)

2 order condition $\nabla^2 f(x) \ge \mu I$, positive curvature at every x

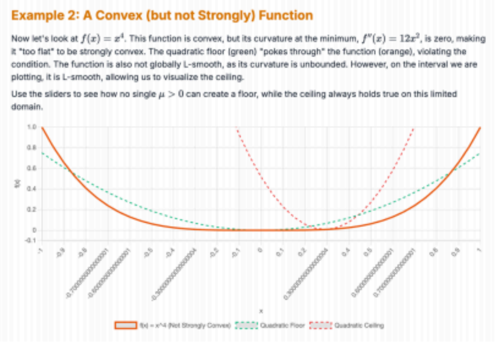


Example 1: A Strongly Convex & L-Smooth Function

Let's examine $f(x) = 0.5x^4 + x^2$. This function is strongly convex. Its curvature (second derivative) is $f''(x) = 6x^2 + 2$. The minimum value of the curvature is 2 (at x=0), so it's strongly convex with $\mu = 2$. It is also L-smooth on the plotted interval.

This means it's "sandwiched" between a quadratic floor (strong convexity) and a quadratic ceiling (L-smoothness). Use the slider to see how these bounds are tangent at point x but always contain the function.

Tangent Point: x = 1.2



Example 2: A Convex (but not Strongly) Function

Now let's look at $f(x) = x^4$. This function is convex, but its curvature at the minimum, $f''(x) = 12x^2$, is zero, making it "too flat" to be strongly convex. The quadratic floor (green) "pokes through" the function (orange), violating the condition. The function is also not globally L-smooth, as its curvature is unbounded. However, on the interval we are plotting, it is L-smooth, allowing us to visualize the ceiling.

Use the sliders to see how no single $\mu > 0$ can create a floor, while the ceiling always holds true on this limited domain.

Guaranteed One-Step Progress (Choosing a Step Size)

Apply the Descent Lemma with $x = x_k, y = x_{k+1} = x_k - \alpha\nabla f(x_k)$:

$$f(x_{k+1}) \le f(x_k) + \nabla f(x_k)^\top (x_{k+1} - x_k) + \frac{L}{2}||x_{k+1} - x_k||^2.$$

Substitute $x_{k+1} - x_k = -\alpha\nabla f(x_k)$:

$$\begin{aligned} f(x_{k+1}) &\leq f(x_k) - \alpha||\nabla f(x_k)||^2 + \frac{L\alpha^2}{2}||\nabla f(x_k)||^2 \\ &= f(x_k) - \alpha\Big(1 - \frac{L\alpha}{2}\Big)||\nabla f(x_k)||^2. \end{aligned}$$

So if $\alpha \leq \frac{1}{L}$, then $1 - \frac{L\alpha}{2} \geq \frac{1}{2}$ and we get the clean bound (taking $\alpha = \frac{1}{L}$):

$$f(x_{k+1}) \leq f(x_k) - \frac{1}{2L}||\nabla f(x_k)||^2.$$

This shows the objective is **monotonically decreasing** with that step size. Also rearrange it into:

$$||\nabla f(x_k)||^2 \leq 2L\big(f(x_k) - f(x_{k+1})\big) = 2L(e_k - e_{k+1}). \tag{1}$$

Connecting Error to Distance (The Key Bridge)

From convexity, set $x = x_k, y = x^*$:

$$\begin{aligned} f(x^*) &\geq f(x_k) + \nabla f(x_k)^\top(x^* - x_k) \\ \Rightarrow e_k &= f(x_k) - f(x^*) \leq \nabla f(x_k)^\top(x_k - x^*). \end{aligned}$$

Now expand the squared distance after one GD step:

$$\begin{aligned} ||x_{k+1} - x^*||^2 &= ||x_k - \alpha\nabla f(x_k) - x^*||^2 \\ &= ||x_k - x^*||^2 - 2\alpha\nabla f(x_k)^\top(x_k - x^*) + \alpha^2||\nabla f(x_k)||^2. \end{aligned}$$

Rearrange to isolate the inner product:

$$\nabla f(x_k)^\top(x_k - x^*) = \frac{1}{2\alpha}\Big(||x_k - x^*||^2 - ||x_{k+1} - x^*||^2\Big) + \frac{\alpha}{2}||\nabla f(x_k)||^2.$$

Combine with $e_k \leq \nabla f(x_k)^\top(x_k - x^*)$:

$$e_k \leq \frac{1}{2\alpha}\Big(||x_k - x^*||^2 - ||x_{k+1} - x^*||^2\Big) + \frac{\alpha}{2}||\nabla f(x_k)||^2.$$

When $\alpha = 1/L$:

$$e_k \leq \frac{L}{2}\Big(||x_k - x^*||^2 - ||x_{k+1} - x^*||^2\Big) + \frac{1}{2L}||\nabla f(x_k)||^2. \tag{2}$$

The Cancellation Trick (Telescoping)

Substitute equation (1) into equation (2):

$$e_k \leq \frac{L}{2}\Big(||x_k - x^*||^2 - ||x_{k+1} - x^*||^2\Big) + (e_k - e_{k+1}).$$

Cancel e_k from both sides:

$$e_{k+1} \leq \frac{L}{2}\Big(||x_k - x^*||^2 - ||x_{k+1} - x^*||^2\Big).$$

Sum this inequality for $k = 0$ to $K - 1$ (telescoping sum):

$$\begin{aligned}\sum_{k=0}^{K-1} e_{k+1} = \sum_{k=1}^{K} e_k \leq \sum_{k=0}^{K-1} \frac{L}{2}\Big(||x_k - x^*||^2 - ||x_{k+1} - x^*||^2\Big) \\ = \frac{L}{2}\Big(||x_0 - x^*||^2 - ||x_K - x^*||^2\Big) \leq \frac{L}{2}||x_0 - x^*||^2.\end{aligned}$$

Since $f(x_k)$ is decreasing, e_k is non-increasing, so $e_K \leq \frac{1}{K}\sum_{k=1}^{K} e_k$. Therefore:

$$e_K = f(x_K) - f(x*) \leq \frac{L||x_0 - x^*||^2}{2K}.$$

To ensure $e_K \leq \varepsilon$, it suffices that

$$K \geq \frac{L||x_0 - x^*||^2}{2\varepsilon},$$

which is the classical $O(1/\varepsilon)$ convergence rate for GD on convex, L-smooth functions.

Strong Convexity

If f is μ-**strongly convex**, then:

$$f(y) \geq f(x) + \nabla f(x)^\top (y - x) + \frac{\mu}{2}||y - x||^2,$$

equivalently (when twice differentiable) $\nabla^2 f(x) \succeq \mu I$. With both L-smoothness and strong convexity, GD can achieve **linear convergence** (geometric decay) rather than $O(1/K)$.

Mini-batch SGD, Momentum, NAG, RMSProp, Adam

Mini-batch SGD (The Practical Default)

We often minimize an empirical risk

$$L(w) = \frac{1}{N} \sum_{i=1}^{N} \ell_i(w).$$

Full-batch GD uses $\nabla L(w)$, which is expensive when N is large. **Mini-batch SGD** approximates the gradient using a small batch B:

$$g_t \approx \nabla L(w_t) = \frac{1}{|B|} \sum_{i \in B} \nabla \ell_i(w_t), \qquad w_{t+1} = w_t - \alpha g_t.$$

Why Stochasticity(随机性) Can Be Preferable

In convex problems, GD has clean guarantees and the landscape is "one bowl." In non-convex problems (typical deep nets), the landscape contains many critical points, including **saddle points**. The noise in SGD can help exploration:

- It can "kick" the iterate out of flat/unstable saddle regions.
- It can help avoid getting stuck in certain sharp basins(盆地) (informal but often observed).

Dampen the Oscillation(抑制振荡): Momentum Methods

Plain SGD can zig-zag, especially in "ravines(深谷)" where curvature is steep in one direction and flat in another. Momentum adds a memory of past gradients to smooth updates. One common form (velocity formulation):

$$v_{t+1} = \beta v_t + g_t, \qquad w_{t+1} = w_t - \alpha v_{t+1},$$

with $v_0 = 0$ and $\beta \in [0, 1)$ (often around 0.9–0.99).

Interpretation: v_t becomes an exponentially weighted moving average of gradients:

$$v_t = g_t + \beta g_{t-1} + \beta^2 g_{t-2} + \cdots$$

So directions that persist get amplified; directions that alternate(反复交替) (oscillations) cancel out(相互抵消).

Nesterov Accelerated Gradient (NAG): Look Ahead, Then Correct

Momentum can overshoot(超调) because it commits(笃信) strongly to the velocity. NAG modifies where you measure the gradient: evaluate it at a "lookahead" point. A typical NAG-like update:

$$\tilde{w}_t = w_t - \alpha\beta v_t, \qquad v_{t+1} = \beta v_t + \nabla L(\tilde{w}_t), \qquad w_{t+1} = w_t - \alpha v_{t+1}.$$

Intuition: "move in the momentum direction first, then compute the gradient there and correct." This often stabilizes training and can speed convergence compared to vanilla momentum.

Speeding Up Convergence: Step-size Heuristics

Move slowly in steep directions, fast in flat directions. This motivates per-parameter adaptive learning rates: scale(调整) updates differently in each coordinate(坐标分量) depending on gradient history.

RMSProp: Adaptive Step Sizes via Running Squared Gradients(平方梯度滑动量)

RMSProp keeps a moving average of squared gradients:

$$s_t = \beta s_{t-1} + (1-\beta)g_t^2 \quad \text{(elementwise square)},$$

and updates

$$w_{t+1} = w_t - \alpha \frac{g_t}{\sqrt{s_t + \varepsilon}}.$$

Effect: if a coordinate has consistently large gradients, s_t grows and the effective step size shrinks for that coordinate; if gradients are small, steps become larger.

Adam: Momentum + RMSProp (Plus Bias Correction)

Adam combines:

- a first-moment(一阶矩) estimate (momentum-like):

$$m_t = \beta_1 m_{t-1} + (1-\beta_1)g_t,$$

- a second-moment estimate (RMSProp-like):

$$v_t = \beta_2 v_{t-1} + (1-\beta_2)g_t^2.$$

Because m_t, v_t start at zero, they are biased early. Adam applies bias correction:

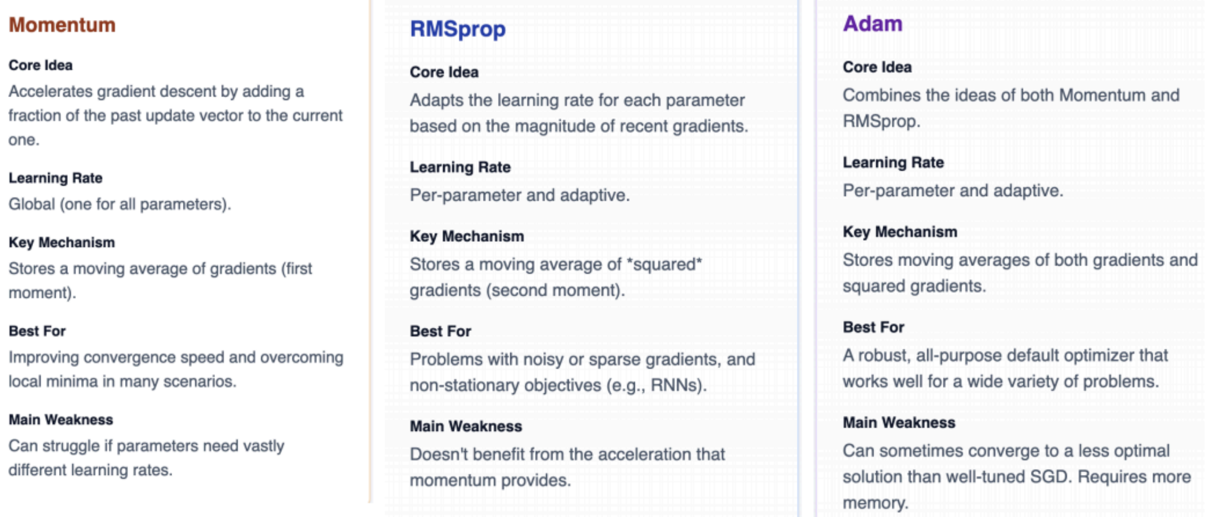
$$\hat{m}_t = \frac{m_t}{1-\beta_1^t}, \qquad \hat{v}_t = \frac{v_t}{1-\beta_2^t}.$$

Update:

$$w_{t+1} = w_t - \alpha \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \varepsilon}.$$

Common defaults: $\beta_1 = 0.9,\ \beta_2 = 0.999$, α tunable.

Comparison at a Glance (What problem each method targets)



Momentum

Core Idea
Accelerates gradient descent by adding a fraction of the past update vector to the current one.

Learning Rate
Global (one for all parameters).

Key Mechanism
Stores a moving average of gradients (first moment).

Best For
Improving convergence speed and overcoming local minima in many scenarios.

Main Weakness
Can struggle if parameters need vastly different learning rates.

RMSprop

Core Idea
Adapts the learning rate for each parameter based on the magnitude of recent gradients.

Learning Rate
Per-parameter and adaptive.

Key Mechanism
Stores a moving average of *squared* gradients (second moment).

Best For
Problems with noisy or sparse gradients, and non-stationary objectives (e.g., RNNs).

Main Weakness
Doesn't benefit from the acceleration that momentum provides.

Adam

Core Idea
Combines the ideas of both Momentum and RMSprop.

Learning Rate
Per-parameter and adaptive.

Key Mechanism
Stores moving averages of both gradients and squared gradients.

Best For
A robust, all-purpose default optimizer that works well for a wide variety of problems.

Main Weakness
Can sometimes converge to a less optimal solution than well-tuned SGD. Requires more memory.

Takeaway Messages

Reduced oscillation: Momentum makes the trajectory smoother, especially in "ravines(深谷)" where one direction is steep and the other is flat. It damps(抑制) the classic SGD zig-zag.

Faster convergence (often): smoothing usually creates a more direct path and builds speed in consistent downhill directions.

Nesterov's advantage (NAG): by "looking ahead," NAG tends to brake more appropriately near the minimum, reducing overshoot and making training more stable.

New hyperparameter: Momentum introduces β (momentum coefficient(系数)). Default values like 0.9 often work well, but the *interaction* between α and β matters—too large a learning rate plus high momentum can turn your optimizer into a pinball(弹珠) machine.

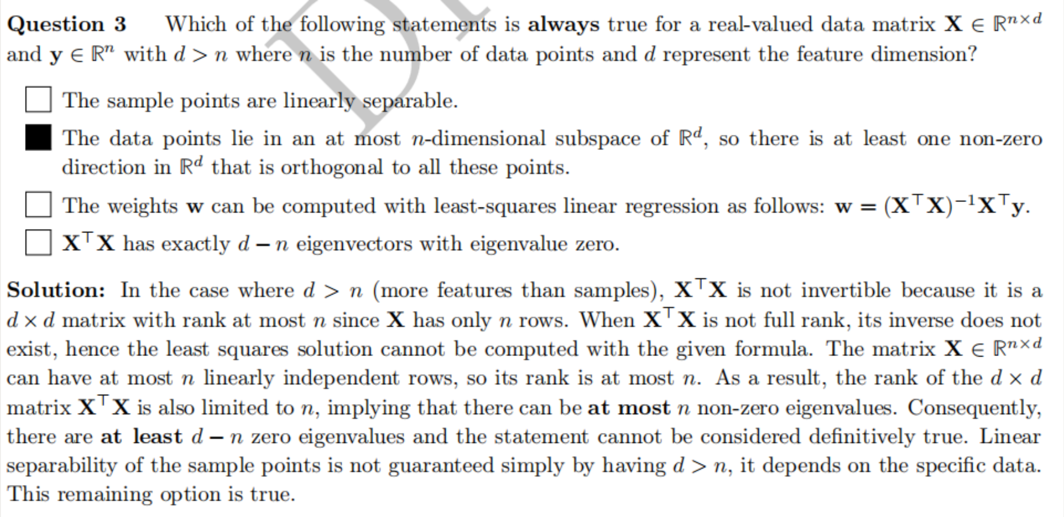
Practical Advice

Start with Adam for most deep learning tasks: it’s a strong baseline because it handles scaling differences and gradient noise with minimal tuning.

Consider RMSProp in settings that are highly non-stationary or historically tricky (e.g., some RNN / RL-style training loops). (Modern practice often still uses Adam variants here, but RMSProp remains conceptually relevant.)

Don’t forget SGD + Momentum: when you have time to tune and you care about best final generalization, SGD+Momentum can sometimes beat Adam—especially in supervised vision-style regimes(系统) with good schedules.

😈



Question 3 Which of the following statements is **always** true for a real-valued data matrix $\mathbf{X} \in \mathbb{R}^{n \times d}$ and $\mathbf{y} \in \mathbb{R}^{n}$ with $d > n$ where n is the number of data points and d represent the feature dimension?

☐ The sample points are linearly separable.

■ The data points lie in an at most n-dimensional subspace of $\mathbb{R}^d$, so there is at least one non-zero direction in $\mathbb{R}^d$ that is orthogonal to all these points.

☐ The weights $\mathbf{w}$ can be computed with least-squares linear regression as follows: $\mathbf{w} = (\mathbf{X}^\top \mathbf{X})^{-1}\mathbf{X}^\top \mathbf{y}$.

☐ $\mathbf{X}^\top \mathbf{X}$ has exactly $d - n$ eigenvectors with eigenvalue zero.

Solution: In the case where $d > n$ (more features than samples), $\mathbf{X}^\top \mathbf{X}$ is not invertible because it is a $d \times d$ matrix with rank at most n since $\mathbf{X}$ has only n rows. When $\mathbf{X}^\top \mathbf{X}$ is not full rank, its inverse does not exist, hence the least squares solution cannot be computed with the given formula. The matrix $\mathbf{X} \in \mathbb{R}^{n \times d}$ can have at most n linearly independent rows, so its rank is at most n. As a result, the rank of the $d \times d$ matrix $\mathbf{X}^\top \mathbf{X}$ is also limited to n, implying that there can be **at most** n non-zero eigenvalues. Consequently, there are **at least** $d - n$ zero eigenvalues and the statement cannot be considered definitively true. Linear separability of the sample points is not guaranteed simply by having $d > n$, it depends on the specific data. This remaining option is true.

给定：特征数多于样本数，这是典型的 **高维小样本（overparameterized）** 场景。

❌ **选项 1**

线性可分性不是只由维度决定的。反例很简单：所有点完全重合；或者标签随机但特征退化。线性可分性取决于**数据分布和标签**，不是“维度是否大于样本数”。

✅ **选项 2**

X 的行向量最多 n 个线性无关，所以所有数据点 **最多张成一个 n 维子空间**。整个空间是 $\mathbb{R}^d$，而子空间维数 ≤ n，且 $d > n$ ➡️ **必然存在非零的正交补空间**。

存在 $v \in \mathbb{R}^d, v \neq 0$，使得：$Xv = 0$。也就是说：这个方向对所有数据点“不可见”；数据在这个方向上的投影全为 0。

❌ **选项 3**

$$X^\top X \in \mathbb{R}^{d \times d}$$

但:

$$\operatorname{rank}(X^\top X) = \operatorname{rank}(X) \leq n < d$$

不是满秩矩阵，不可逆，所以这个公式 **根本不成立**。

正确做法应该是：Moore–Penrose 伪逆或加正则项（Ridge）。

❌ **选项 4**

我们只知道：$\operatorname{rank}(X^\top X) \leq n$，因此：非零特征值 **最多 n 个**零特征值 **至少 $d - n$ 个**。但注意：至少 ≠ 恰好。如果：X 的行本身就线性相关，那么 rank(X) < n，零特征值会 **多于 (d-n)**。