Machine Learning: 5-8

Created	@2025年12月16日 06:17
Class	Fundamentals of Computer Science

5 Model Evaluation and Selection

Understanding Model Generalization and Selection

A model's true success lies not in fitting the training data, but in generalizing to unseen reality.

Model selection involves making decisions that determine how a model learns and behaves. These decisions are encoded in **hyperparameters**, which control the model's complexity and performance.

Example 1: Ridge Regression Regularization strength λ controls how much we penalize large weights. Larger $\lambda \Rightarrow$ simpler model (less variance, more bias).

Example 2: Polynomial Features The **degree** d of polynomial expansion(多项式展开) determines how complex the feature space becomes. Higher $d \Rightarrow$ more expressive model, but higher risk of overfitting.

For neural networks, hyperparameter tuning becomes vastly more complex, including:

- **Optimization algorithm** (SGD, Adam, RMSProp, etc.)
- **Algorithm parameters** (learning rate, batch size, momentum)
- **Architecture design** (network type, depth, width)
- **Regularization strategies** (dropout, early stopping, data augmentation)

Because each decision interacts with others, **systematic validation** is crucial.

A Formal Framework: The Probabilistic Setup

To reason about model selection rigorously, we define the learning problem probabilistically.

1. **Data Distribution** D:

   There exists a true but unknown distribution over data points (x, y).

2. **Dataset Sampling** S:

   We only observe a finite sample $S = (x_n, y_n)_{n=1}^{N}$ drawn i.i.d. from D.

3. **Learning Algorithm** A:

   A function that maps a dataset S to a prediction function $f_S = A(S)$, possibly depending on hyperparameters λ.

The **true goal** is to minimize the *generalization error* (or *expected loss*):

$$L_D(f) = \mathbb{E}_{(x,y)\sim D}[\ell(y, f(x))]$$

This measures how well the model performs across the entire data distribution.

The Challenge: We **cannot compute** $L_D(f)$ **directly**, since D is unknown. Thus, we must estimate it using available data — leading us to **empirical error**.

We estimate the generalization error using the finite dataset S:

$$L_S(f) = \frac{1}{|S|} \sum_{(x_n,y_n)\in S} \ell(y_n, f(x_n))$$

As the dataset grows ($|S| \to \infty$), the empirical error converges to the true error $L_D(f)$ (Law of Large Numbers). However, for finite datasets, L_S is a random variable — an **unbiased but noisy estimator** of L_D. If we evaluate the trained model on its training set, $L_S(f_S)$ is optimistically biased because f_S was chosen to minimize L_S. This can hide poor generalization and contribute to overfitting.

The Solution: Train-Test Split

To obtain an *unbiased estimate* of generalization:

1. **Split the dataset** into independent sets:
   - S_{train}: for training
   - S_{test}: for evaluation
2. **Train the model** only on S_{train}.
3. **Compute test error**:

$$L_{S_{test}}(f_{S_{train}}) = \frac{1}{|S_{test}|} \sum_{(x_n,y_n)\in S_{test}} \ell(y_n, f_{S_{train}}(x_n))$$

This serves as an **unbiased estimate** of the model's generalization performance. Confidence bounds can further quantify how much uncertainty remains in this estimate.

If we evaluate the trained model on its training set, L_S(f_S) is optimistically biased because f_S was chosen to minimize L_S. This can hide poor generalization and contribute to overfitting.

Model Evaluation, Confidence Bounds, and Cross-Validation

Confidence Through Concentration Inequalities

Once we split data into train and test sets, we still face uncertainty: how close is our measured **test error** $L_{S_{test}}(f)$ to the *true generalization error* $L_D(f)$?

To answer this, we rely on **Hoeffding's Inequality**, which provides probabilistic guarantees.

$$L_D(f) \leq L_{S_{test}}(f) + \sqrt{\frac{c \ln(1/\delta)}{2N_{test}}}$$

- $L_D(f)$: True (unknown) generalization error
- $L_{S_{test}}(f)$: Measured test error
- N_{test}: Test set size
- δ: Risk level (e.g., 0.05 for 95% confidence)

This inequality states that the true error is very unlikely to exceed the test error by more than the additional bound term. As the test set grows, the uncertainty term $O(1/\sqrt{N_{test}})$ shrinks — giving **tighter confidence intervals** and more reliable performance estimates.

A Practical Use for the Bound

By combining this bound with our data split, we can make **quantified confidence statements(量化置信度陈述)** about model performance.

1. **Split the data:** Divide S into S_{train} and S_{test}.
2. **Train the model:** Fit $f_{S_{train}}$ using only S_{train}.
3. **Compute test error:** Evaluate on S_{test} to get $L_{S_{test}}(f_{S_{train}})$.

Confidence Statement: With 95% confidence $(\delta = 0.05)$, the true generalization error $L_D(f)$ is no worse than$L_{S_{test}}(f_{S_{train}}) + \sqrt{\frac{c \ln(1/\delta)}{2N_{test}}}$.

This provides a **probabilistic guarantee** rather than just a single test error — grounding our evaluation in statistical confidence.

Using the Test Set for Model Selection

Given multiple candidate hyperparameter settings $\lambda_1, \lambda_2, \ldots, \lambda_K$:

- **Data Splitting:** Divide data into S_{train} and S_{test}.

- **Iterative Training & Evaluation:**
  - For each λ_k, train a model f_{S_{train},λ_k} on S_{train}.
  - Evaluate its performance $L_{S_{test}}(f_{S_{train},\lambda_k})$.
- **Select the best** λ_k that minimizes the test error.

At first glance, this seems reasonable — but it hides a **critical flaw**. When we pick the "best" model based on test performance, we **indirectly use test data** during selection. This causes **information leakage** — the test set has influenced the choice of λ.

As a result:

- The test set is no longer *independent* of training decisions.
- The estimated generalization error becomes **optimistically biased** — the model appears better than it truly is on unseen data.

This violates the principle that the test set must *simulate new data*, untouched until the final evaluation.

The Better Solution: Train–Validation–Test Split

To prevent leakage, we introduce a third partition:

- **Training set** (S_{train}): Fit the model parameters.
- **Validation set** (S_{valid}): Tune hyperparameters λ.
- **Test set** (S_{test}): Final unbiased performance check.

Typical splits: 80% / 10% / 10%, or 50% / 25% / 25%.

The validation set acts as a *proxy test set* for hyperparameter tuning. However, since it's smaller, the reliability of validation results depends on data size — hence the question: *is this good enough?*

Cross-Validation: A More Robust Approach

Instead of a single validation split, **K-fold Cross-Validation** provides a statistically stronger evaluation. **Procedure:**

1. Split the dataset into K roughly equal folds $D_1, D_2, ..., D_K$.
2. For each fold j:
   - Train the model on all data except D_j.
   - Validate on D_j to compute $L_{D_j}(f_{D_{-j}})$.
3. Compute the **average validation error**:

$$CV_K(\lambda_k) = \frac{1}{K}\sum_{j=1}^{K} L_{D_j}(f_{D_{-j}}, \lambda_k)$$

This gives a stable, less noisy estimate of performance for each hyperparameter settingλ_k.

After computing CV errors for all hyperparameter candidates:

1. **Model selection:** Pick the λ_j with the lowest CV error.
2. **Final training:** Retrain the model $f^* = f_{S_{train}}(\lambda_j)$ on the *entire training+validation data*.
3. **Model evaluation:** Compute $L_{S_{test}}(f^*)$ on the untouched test set.

This yields a model that uses all available training data while maintaining an unbiased final evaluation.

The Intuition Behind Cross-Validation

Goal: Find parameters λ^* that minimize the true generalization error

$$L_{S_{test}}(f^*) = \text{smallest possible error on unseen data.}$$

How CV achieves this: By rotating which subset is used for validation, CV simulates testing on many unseen subsets, improving reliability.

Choosing K:

- If K is too large (e.g., leave-one-out CV), each training set is nearly the full dataset. The test portions are tiny, making variance large and results unstable.
- Typically, $K = 5$ or $K = 10$ balances bias and variance.

Bias–Variance Decomposition

We analyze the expected squared error at a fixed input point x_0:

$$\mathbb{E}\big[(y - \hat{f}(x_0))^2\big]$$

Assume the data-generating process:

$$y = f(x_0) + \varepsilon, \quad \mathbb{E}[\varepsilon] = 0, \quad \mathbb{E}[\varepsilon^2] = \sigma^2$$

Here: $f(x_0)$ is the true (unknown) target function, $\hat{f}(x_0)$ is the learned predictor, depending on the training dataset, ε is irreducible noise.

We expand the expected error:

$$\mathbb{E}\big[(y-\hat{f}(x_0))^2\big] = \mathbb{E}\big[(f(x_0)-\hat{f}(x_0))^2\big] + 2\mathbb{E}[\varepsilon(f(x_0)-\hat{f}(x_0)] + \mathbb{E}[\varepsilon^2]$$
$$\mathbb{E}[\varepsilon(f(x_0)-\hat{f}(x_0)] = \mathbb{E}[\varepsilon]\cdot\mathbb{E}[f(x_0)-\hat{f}(x_0)] = 0$$

Now decompose the first term by adding and subtracting $\mathbb{E}[\hat{f}(x_0)] = \mu$, $f(x_0) = f$:

$$\mathbb{E}\big[(f-\hat{f})^2\big] = \mathbb{E}\Big[(f-\mu+\mu-\hat{f})^2\Big]$$
$$= (f-\mu)^2 + \mathbb{E}[(\hat{f}-\mu)^2] + 2(f-\mu)\underbrace{\mathbb{E}[\mu-\hat{f}]}_{=0}.$$

Expanding the square and taking expectations, the cross-term vanishes because $f(x_0)$ and $\mathbb{E}[\hat{f}(x_0)]$ are constants w.r.t. dataset randomness. We obtain:

$$\mathbb{E}\big[(y-\hat{f}(x_0))^2\big] = \underbrace{(f(x_0)-\mathbb{E}[\hat{f}(x_0)])^2}_{\text{Bias}^2} - \underbrace{\mathbb{E}\big[(\hat{f}(x_0)+\mathbb{E}[\hat{f}(x_0)])^2\big]}_{\text{Variance}} + \underbrace{\sigma^2}_{\text{Irreducible noise}}$$

Definitions

Bias measures **systematic error**: how far the *average model* is from the true function.

$$\text{Bias}(x_0) = \mathbb{E}[\hat{f}(x_0)] - f(x_0)$$

Variance measures **sensitivity to the training dataset**: how much the learned model fluctuates(波动) across different datasets.

$$\text{Var}(x_0) = \mathbb{E}\big[(\hat{f}(x_0) - \mathbb{E}[\hat{f}(x_0)])^2\big]$$

Irreducible Noise is inherent randomness in the data and cannot be eliminated by any model.

$$\sigma^2 = \mathbb{E}[\varepsilon^2]$$

Consider training multiple models $\hat{f}_1, \hat{f}_2, \hat{f}_3$ on different datasets sampled from the same distribution. Define the average predictor:

$$\bar{f}(x) = \mathbb{E}_D[\hat{f}(x)] \approx \frac{1}{M}\sum_{i=1}^{M}\hat{f}_i(x)$$

Bias reflects the gap between the **true function** $f(x)$ and the **average learned function** $\bar{f}(x)$. High bias models are **too simple** and fail to capture the true structure of the data

Variance measures how much individual models $\hat{f}_i$ deviate from the average $\bar{f}$:

$$\text{Var}(x) = \mathbb{E}_D\big[(\hat{f}(x) - \bar{f}(x))^2\big]$$

High variance models fit the training data very closely and change dramatically with small dataset perturbations.

Model complexity controls the balance:

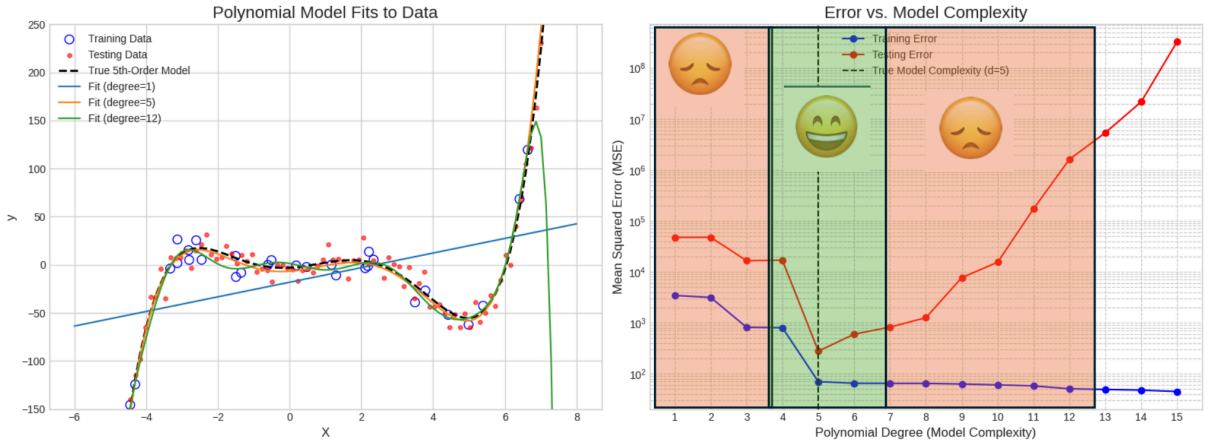
- **Low complexity models:** High bias, Low variance, Tend to underfit.
- **High complexity models:** Low bias, High variance, Tend to overfit.

There is typically an **optimal complexity region** where the total expected error is minimized. This directly explains the U-shaped test error curve observed as model complexity increases.

Underfitting vs Overfitting

- **Underfitting:** Training error high, Test error high, Dominated by bias.
- **Overfitting:** Training error very low, Test error high, Dominated by variance.

This explains why **smaller training error does NOT guarantee smaller test error**.



Relation to Regularization

Regularization is a tool to control bias and variance:

- Strong regularization: Increases bias, Reduces variance
- Weak regularization: Reduces bias, Increases variance

Choosing regularization strength is therefore a **bias–variance balancing act**, not a purely optimization problem.

Double Descent, Regularization, and Constrained Optimization

From Classical Bias–Variance to Double Descent

In classical learning theory, test risk as a function of model complexity follows a **U-shaped curve**:

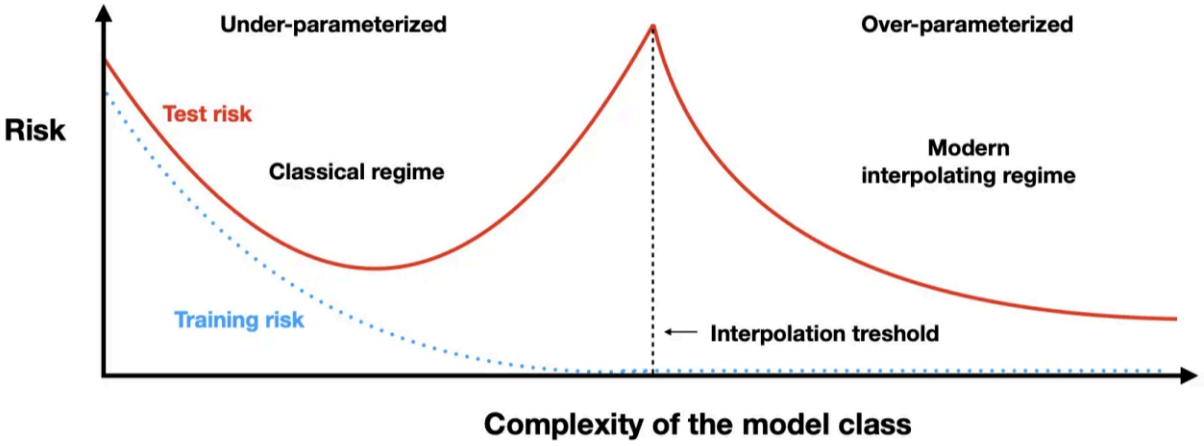
- Low complexity → high bias → underfitting
- High complexity → high variance → overfitting

However, modern machine learning models (e.g., deep neural networks) often operate in a **highly over-parameterized regime**, where this classical picture breaks down.

Empirically, we observe a **double descent curve**:

- Test risk decreases in the under-parameterized regime,
- Peaks near the **interpolation threshold** (where training error reaches zero),
- Then decreases again as model complexity continues to increase.

This phenomenon reconciles(使一致) modern practice with classical theory.



Regimes of Model Complexity

As model complexity increases, learning progresses through three regimes:

Under-parameterized regime

- Model capacity is insufficient to fit the training data.
- Training error is non-zero.
- Bias dominates the error.

Interpolation threshold

- The model just becomes powerful enough to fit the training data perfectly.
- Training risk approaches zero.
- Test risk often spikes due to extreme variance and instability.

Over-parameterized (interpolating) regime

- The model has many more parameters than data points.
- Surprisingly, test risk decreases again.
- Implicit regularization from optimization algorithms (e.g., SGD) plays a crucial role.

Polynomial Regression as a Canonical(典型的) Example

Recall the linear model:

$$\hat{y}_i = \omega_0 + \omega_1 x_{i1} + \cdots + \omega_k x_{ik}$$

For polynomial regression: k is the polynomial order, $x_{ik} = x_i^k$. The optimization problem is:

$$\arg\min_{\mathbf{w}} ||y - Xw||_F$$

A naive approach to model selection is to increase the polynomial degree one-by-one and evaluate performance. This works in low-dimensional settings but becomes unstable and computationally inefficient as complexity grows.

To control model complexity, we introduce **regularization**, which explicitly constrains the hypothesis space. A general regularized regression problem can be written as:

$$\min_{\mathbf{w}} ||y - Xw||_F \quad \text{s.t.} \quad ||\mathbf{w}||_p \leq R$$

where:

- $||\cdot||_p$ can be $\ell_1, \ell_2, \ell_0, \ell_4, \ldots$,
- R controls the strength of regularization.

Regularization restricts the solution to a **feasible region** in parameter space, reducing variance at the cost of increased bias.

Regularization as Constrained Optimization

Regularization transforms an unconstrained optimization problem into a constrained one. Geometrically:

- The loss function defines level sets (objective contours(等值线)),
- The constraint $||w||_p \leq R$ defines a feasible region,
- The optimal solution lies where the lowest loss contour touches the boundary of the feasible region.

This geometric interpretation explains:

- Why different norms lead to different solution structures,

- Why sparsity emerges naturally from ℓ_1-constraints.

Solving Constrained Problems: Projected Gradient Descent

One practical method to solve constrained optimization problems is **Projected Gradient Descent (PGD)**. The algorithm alternates between:

1. A gradient descent step that may leave the feasible region,
2. A projection step that maps the solution back onto the constraint set.

Formally:

$$w^{(t+1)} = \Pi_{\mathcal{C}}\left(w^{(t)} - \eta \nabla f(w^{(t)})\right)$$

where $\Pi_{\mathcal{C}}$ is the projection operator onto the feasible set $\mathcal{C}$. PGD makes the constraint explicit and controllable during optimization.

Ridge Regression, Sparsity, and Regularization Paths

Regularized Regression: Objective Function

We consider the general regularized regression problem:

$$\min_{\mathbf{w}} ||y - X\mathbf{w}||_F + \lambda ||\mathbf{w}||_p$$

This objective consists of two competing terms:

- **Data fidelity term** $||y - Xw||_F$: encourages accurate fitting to the observed data.
- **Regularization term** $\lambda||\mathbf{w}||_p$: encodes a prior belief about the model parameters.

The regularization parameter λ controls the trade-off between fitting the data and constraining model complexity.

Ridge Regression (ℓ_2-Regularization)

Ridge regression uses the ℓ_2-norm:

$$||\mathbf{w}||_2 = \left(\sum_j w_j^2\right)^{1/2}$$

Properties of ℓ_2-regularization:

- Penalizes large weights smoothly.
- Shrinks all coefficients continuously toward zero.
- Does **not** produce sparse solutions (weights rarely become exactly zero).

Ridge regression is especially effective when:

- Many features are correlated.
- We want stability and low variance rather than feature selection.

Sparsity and ℓ_1-Regularization (LASSO)

True sparsity is defined by the ℓ_0-norm:

$$||\mathbf{w}||_0 = \text{number of non-zero } w_j$$

However:

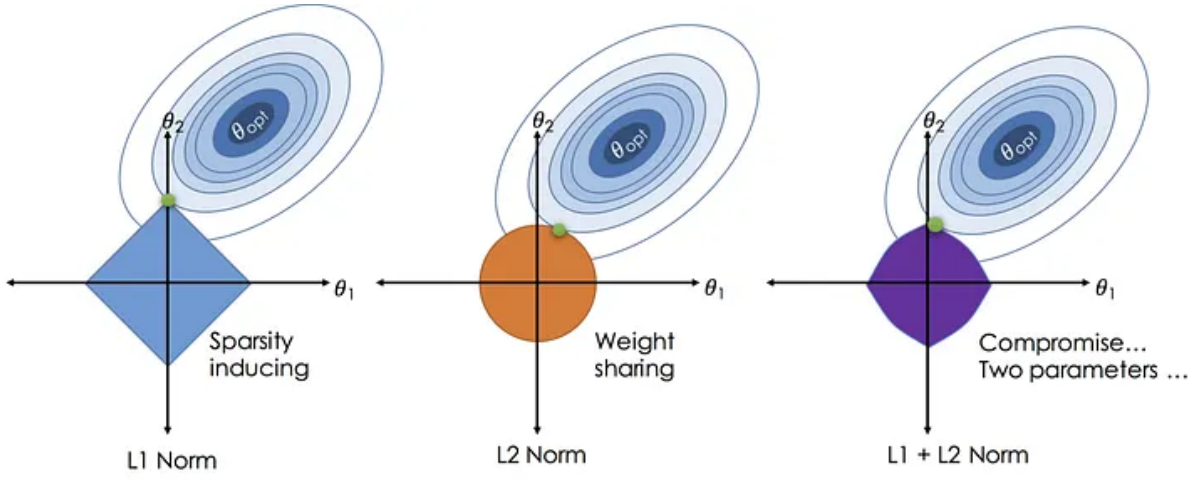
- ℓ_0-minimization is NP-hard.
- It is computationally infeasible for high-dimensional problems.

LASSO replaces ℓ_0 with the closest convex surrogate(最近凸代理):

$$||\mathbf{w}||_1 = \sum_j ||w_j||$$

Properties of ℓ_1-regularization:

- Encourages sparse solutions.
- Performs implicit feature selection.
- Solutions exhibit exact zeros.



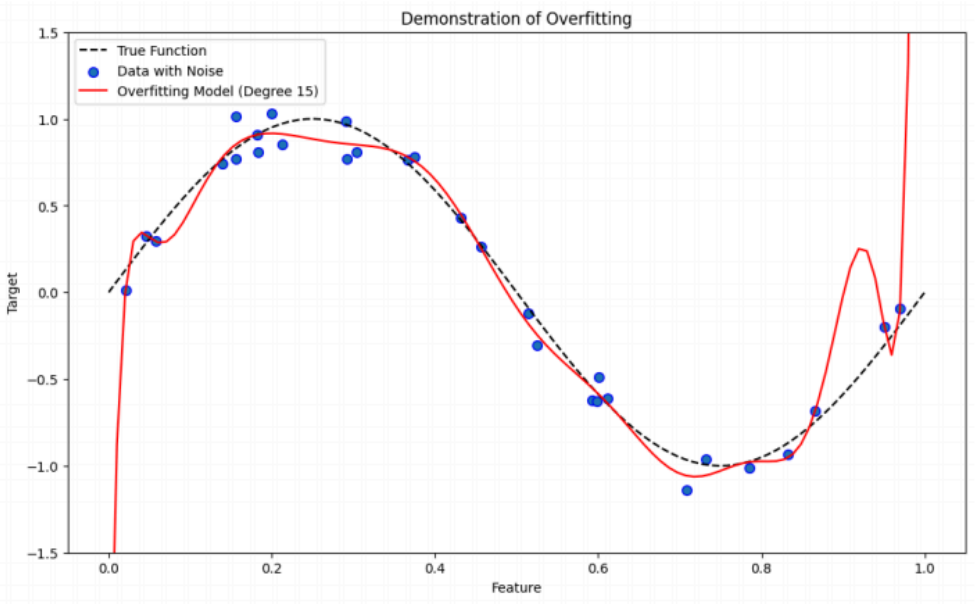
Synthetic Example: Polynomial Regression

We consider a synthetic dataset:

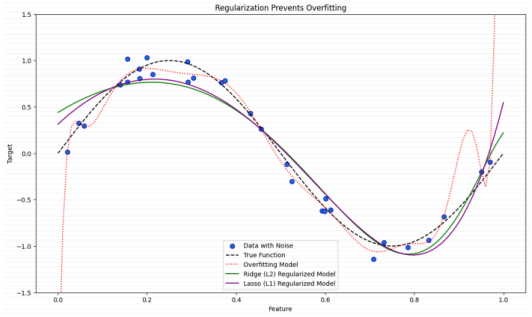
$$y_{\text{true}} = \sin(2\pi x)$$

with noisy observations. A high-degree polynomial model is fitted to the data.

No regularization



With Regularization



Without regularization:

- High-degree polynomial perfectly fits training points.
- Learned function oscillates wildly.
- Coefficients grow extremely large in magnitude.

Adding regularization stabilizes the solution:

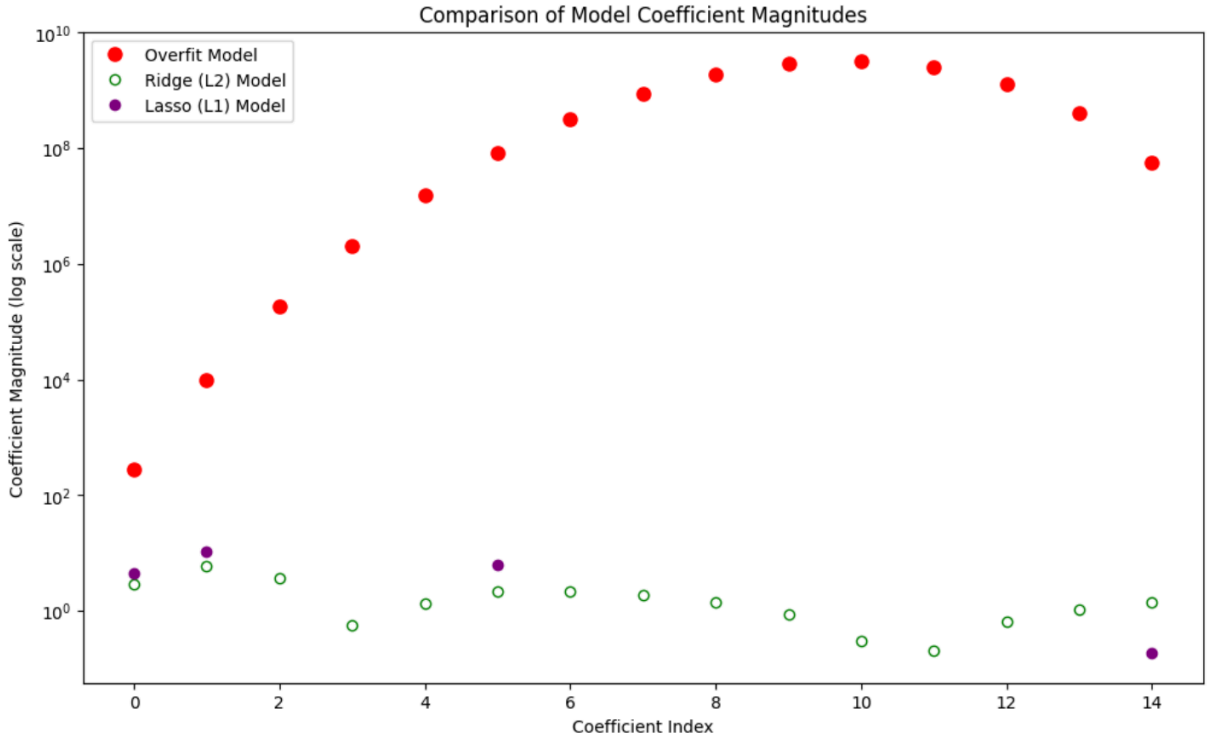
- **Ridge (ℓ_2):** Smooths the function; Keeps all coefficients small.
- **LASSO (ℓ_1):** Produces a simpler function; Removes unnecessary polynomial terms entirely.

Weight Magnitudes and Model Behavior

Examining the learned coefficients:

- Unregularized model: Extremely large coefficients (often orders of magnitude apart).
- Ridge: Coefficients shrink uniformly.
- LASSO: Many coefficients become exactly zero.

This directly explains why regularization improves numerical stability and generalization.



$$w = [w_0, w_1, \cdots, w_{14}]$$

Greedy Approximation: Orthogonal Matching Pursuit (OMP)

To approximate ℓ_0 minimization, greedy algorithms are often used. The OMP objective can be written as:

$$\min_{\mathbf{w}} ||y - X\mathbf{w}||_F + \lambda ||\mathbf{w}||_0$$

OMP constructs a sparse solution iteratively:

1. Initialize residual $r_0 = y$, support set $S = \emptyset$.
2. At each iteration:
   - Select the feature most correlated with the current residual.
   - Add its index to the support set S.
   - Recompute coefficients by solving a least squares problem restricted to S.

- Update the residual.

3. Stop after k iterations or when the residual is sufficiently small.

OMP is intuitive and fast, but it is **not guaranteed** to find the globally optimal sparse solution.

Example: Given:

$$y = [10, 12, 15], \quad X = \begin{bmatrix} 1 & 5 & 8 \\ 1 & 7 & 2 \\ 1 & 6 & 6 \end{bmatrix}$$

Initial residual:

$$r_0 = y$$

Compute correlations:

$$|x_1^\top r_0| = 37, \quad |x_2^\top r_0| = 224, \quad |x_3^\top r_0| = 194$$

Feature x_2 is selected since it has the highest correlation. This illustrates the greedy nature of OMP: **locally optimal choices**, not guaranteed global optimality.

Despite their intuition, greedy algorithms suffer from fundamental limitations:

- Early mistakes cannot be corrected.
- Feature interactions are ignored in early steps.
- No global optimality guarantee.

Most importantly, the underlying ℓ_0 problem remains combinatorial and NP-hard. This motivates the search for a **convex relaxation**.

ℓ_1-Norm and LASSO: The Convex Alternative

The ℓ_1-norm is defined as:

$$||\mathbf{w}||_1 = \sum_j ||w_j||$$

LASSO solves:

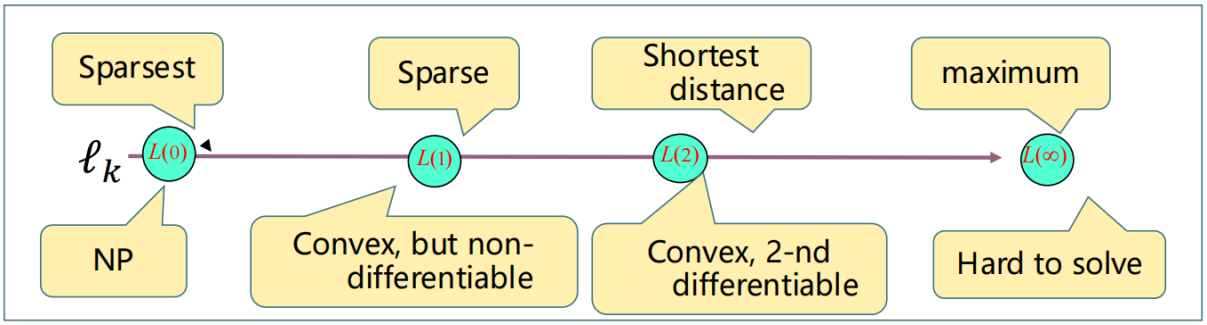
$$\arg\min_{\mathbf{w}} ||y - X\mathbf{w}||_2^2 + \lambda ||\mathbf{w}||_1$$

Key properties:

- Convex optimization problem.

- Computationally efficient.
- Encourages exact sparsity.
- Closest convex approximation to ℓ_0.

LASSO provides the **best trade-off** between tractability and sparsity.



Recall: Ordinary Least Squares (OLS)

We start from the standard linear regression model. The OLS objective is

$$\mathcal{L}(w) = (y - Xw)^\top (y - Xw)$$

This objective is **convex and differentiable**, so we can solve it by setting the gradient to zero. Taking derivative with respect to w:

$$\frac{\partial \mathcal{L}}{\partial w} = -2X^\top (y - Xw)$$

Setting it to zero yields the normal equations:

$$X^\top Xw = X^\top y$$

Assuming $X^\top X$ is invertible, the closed-form solution is:

$$\hat{w}_{\text{OLS}} = (X^\top X)^{-1} X^\top y$$

This solution exists **only if** $X^\top X$ is full-rank.

Solving Ridge Regression (ℓ_2-Regularization)

Ridge regression modifies the OLS objective by adding an ℓ_2 penalty:

$$\mathcal{L}_{\text{Ridge}}(w) = (y - Xw)^\top (y - Xw) + \lambda w^\top w$$

This objective remains **convex and differentiable**. Taking the derivative:

$$\frac{\partial \mathcal{L}}{\partial w} = -2X^\top(y - Xw) + 2\lambda w$$

Setting it to zero gives:

$$(X^\top X + \lambda I)w = X^\top y$$

Thus the ridge solution is:

$$\hat{w}_{\text{Ridge}} = (X^\top X + \lambda I)^{-1} X^\top y$$

Key insight: Adding λI shifts the eigenvalues(特征值) of $X^\top X$ away from zero, making the matrix **always invertible**. This is why ridge regression solves the multicollinearity and ill-conditioning problem.

What Changes with ℓ_1? Enter LASSO

LASSO replaces the ℓ_2 penalty with an ℓ_1 penalty:

$$\mathcal{L}_{\text{LASSO}}(w) = ||y - Xw||_2^2 + \lambda ||w||_1$$

where

$$||w||_1 = \sum_{j=1}^{p} |w_j|$$

Here is the crucial difference:

- The ℓ_1 norm is **not differentiable at** $w_j = 0$.
- Therefore, the gradient does not exist everywhere.
- We can no longer solve LASSO by “take derivative = 0”.

Conclusion: There is **no global closed-form solution** for LASSO like OLS or Ridge.

Shrinkage via Soft-Thresholding

Although LASSO has no closed-form solution in general, the **1D LASSO problem does**. The solution is the **soft-thresholding operator**:

$$S_T(z) = \begin{cases} z - T & z > T \\ 0 & |z| \le T \\ z + T & z < -T \end{cases}$$

where $T = \lambda\alpha$ (depending on step size / normalization).

Interpretation:

- Large values are shrunk toward zero.
- Values inside the “dead zone” are set **exactly to zero**.
- This is the mathematical reason LASSO produces sparsity.

Coordinate Descent for LASSO

Since LASSO is convex but non-differentiable, we use **iterative optimization**. Coordinate descent works as follows:

1. Initialize w (often all zeros).
2. Cycle through coordinates $j = 1, 2, \ldots, p$.
3. Fix all w_k for $k \neq j$.
4. Optimize with respect to w_j only.

Surprisingly, this **one-dimensional subproblem has a closed-form solution** via soft-thresholding.

LASSO Update Rule

For coordinate j, compute:

$$z_j = \sum_{i=1}^{n} x_{ij}\left(y_i - \sum_{k \neq j} x_{ik} w_k\right)$$

Then update:

$$w_j \leftarrow S_{\lambda/2}(z_j)$$

This update:

- Pulls coefficients toward zero,
- Sets small coefficients exactly to zero,
- Performs implicit feature selection.

This is how LASSO achieves sparsity **algorithmically**, not heuristically.

Basis Pursuit: Same Problem, Different Form

In signal processing, LASSO is often called **Basis Pursuit**. Two equivalent formulations:

Penalized (Lagrangian) form:

$$\min_w ||y - Xw||_2^2 + \lambda ||w||_1$$

Constrained form:

$$\min_w ||y - Xw||_2^2 \quad \text{s.t.} \quad ||w||_1 \leq t$$

For every λ, there exists a corresponding t yielding the same solution. This equivalence connects directly to the **diamond-shaped feasible region** interpretation of LASSO.

Coordinate Descent is intuitive but updates **one coordinate at a time**, which can be slow or awkward in some settings. Here introduce two more "global-update" approaches:

1. **ISTA**: a proximal gradient method that updates **all coefficients at once**.
2. **ADMM**: a general splitting framework that turns "hard + non-smooth" into "two easy subproblems".

ISTA: Iterative Shrinkage-Thresholding Algorithm

Each ISTA iteration does two conceptual moves:

1. **Gradient step (smooth part):** move as if you were only minimizing RSS (where OLS would head).
2. **Shrinkage / soft-threshold step (L1 part):** "pull" small coefficients toward zero, and clamp very small ones to exactly zero.

That's the whole personality of Lasso: **follow OLS, then enforce sparsity**.

The two-step update rule

Let $w^{(k)}$ be the current iterate and α be the step size.

(1) Gradient step

$$w_{\text{grad}} = w^{(k)} - \alpha \nabla f\left(w^{(k)}\right), \quad f(w) = ||y - Xw||_2^2$$

For this f,

$$\nabla f(w) = -2X^\top (y - Xw),$$

so equivalently

$$w_{\text{grad}} = w^{(k)} + 2\alpha X^\top (y - Xw^{(k)}).$$

(2) Shrinkage step (prox / soft-threshold)

$$w^{(k+1)} = S_{\lambda\alpha}(w_{\text{grad}})$$

where $S_\tau(\cdot)$ is applied element-wise:

$$S_\tau(a) = \text{sign}(a)\max(|a| - \tau, 0).$$

So coefficients with magnitude $\leq \tau$ become **exactly 0**, and the rest get reduced by τ.

ISTA converges when α is not too large; a standard condition is $\alpha \leq 1/L$, where L is the Lipschitz constant of ∇f. For $f(w) = ||y - Xw||_2^2$, a common bound is $L = 2||X^\top X||_2$. Conceptually: too large α makes the "OLS step" overshoot, and the shrinkage can't reliably fix the chaos.

ADMM: a more general framework (splitting + consensus)

Why ADMM exists

ADMM (Alternating Direction Method of Multipliers(乗子)) is designed for problems that are "easy if split". The core idea is:

1. **Split the problem** by introducing a new variable z so different parts of the objective operate on different variables.
2. **Add a constraint** to tie them together (consensus): $w = z$.
3. **Solve iteratively** by alternating between easy subproblems, gradually enforcing the constraint.

Variable splitting reformulation (for Lasso-style objective)

Start with

$$\min_w ||y - Xw||_2^2 + \lambda||w||_1.$$

Introduce z to "hold" the L1 term:

$$\min_{w,z} ||y - Xw||_2^2 + \lambda||z||_1 \quad \text{s.t. } w - z = 0.$$

Now: the **smooth RSS part** depends only on w; the **non-smooth L1 part** depends only on z. That separation is the whole trick.

Augmented Lagrangian idea (why there's an extra quadratic term)

ADMM uses an augmented Lagrangian (often written with a scaled dual variable u and penalty ρ) to enforce $w \approx z$ during optimization:

$$||y - Xw||_2^2 + \lambda||z||_1 + \frac{\rho}{2}||w - z + u||_2^2$$

(plus a constant term in some conventions).

That quadratic penalty is what makes the alternating steps stable and well-behaved.

ADMM updates for this problem

Each iteration alternates between:

1) w-update = a Ridge-like regression subproblem

Hold z^k, u^k fixed and solve:

$$w^{k+1} = \arg\min_w ||y - Xw||_2^2 + \frac{\rho}{2}||w - (z^k - u^k)||_2^2.$$

This is "ridge regression–shaped": a quadratic loss plus a quadratic regularizer around a target point.

This subproblem has a closed-form linear-system solution (solve a matrix equation), which is why it's considered "easy".

2) z-update = soft-thresholding (prox of L1)

Hold w^{k+1}, u^k fixed and solve:

$$z^{k+1} = \arg\min_z \lambda||z||_1 + \frac{\rho}{2}||z - (w^{k+1} + u^k)||_2^2.$$

Let the target be

$$d = w^{k+1} + u^k.$$

Then the solution is exactly soft-thresholding:

$$z^{k+1} = S_{\lambda/\rho}(d).$$

This mirrors ISTA's "shrink" step, but here it arises as an exact minimizer of the z-subproblem.

3) Dual update (the "enforce consensus" step)

ADMM then updates the scaled dual variable:

$$u^{k+1} = u^k + w^{k+1} - z^{k+1}.$$

The slide's note about "the target d uses the newly updated w" is emphasizing the **alternating** nature: each step immediately benefits from the latest update.

Decomposition
It brilliantly transforms a difficult, non-smooth problem into a sequence of simpler sub-problems that we know how to solve efficiently.

Robustness
The method is known for its robust convergence properties for a wide class of problems, even with non-trivial objective functions.

Parallelization
Its structure is highly amenable to distributed or parallel computing, making it suitable for very large-scale, modern datasets.

How ISTA and ADMM relate

ISTA:

- single variable w, alternating **gradient** then **prox(近似)** on the same vector.
- very simple; per-iteration cost is mostly a gradient (matrix-vector multiplications(矩阵-向量乘法)).

ADMM:

- two variables w, z plus a dual u, alternating between “solve a quadratic problem” and “apply prox”, with an explicit constraint-enforcement mechanism.
- heavier iteration, but often more flexible (works nicely when different pieces have different structures, constraints, or when the w-step can be solved efficiently using factorization).

6 Classification

In machine learning, the type of outcome we want to predict determines whether we are solving a regression or classification problem.

From classification to margin-based thinking

We consider a binary classification problem with labels $y \in \{+1, -1\}$. The model produces a representation $z = f_\theta(x)$, followed by a linear classifier with weight vector w. We define the **margin(间隔)** as

$$m = y \cdot (w^\top z)$$

- If $m > 0$, the prediction has the same sign as the true label → correct classification.
- If $m < 0$, the prediction has the opposite sign → incorrect classification.
- The magnitude $|m|$ measures **confidence**: large positive means confidently correct; large negative means confidently wrong.

Thus, learning reduces to designing a loss function $L(m)$ that penalizes negative (or small) margins appropriately.

Three canonical(经典的) margin-based loss functions

Different classifiers correspond to different choices of $L(m)$.

Hinge Loss (SVM-style):

$$L(m) = \max(0, 1 - m)$$

This loss enforces a *margin requirement(约束)*. Once $m \geq 1$, the loss becomes zero, meaning that confidently correct predictions are not further rewarded.

Logistic Loss (Logistic Regression):

$$L(m) = \log(1 + e^{-m})$$

This loss is smooth and strictly decreasing. It never becomes exactly zero, but approaches zero asymptotically as $m \to +\infty$.

Question 8 Which of the following statements is TRUE regarding the application of logistic regression and linear regression on binary classification tasks?

- [x] Logistic regression models the probability of an instance belonging to a particular class, while linear regression predicts class labels directly.
- [ ] Logistic regression is able to generate non-linear decision boundaries with respect to original input features when the data are not linearly separable.
- [ ] Both logistic regression and linear regression have a closed-form solution for the optimal parameters.
- [ ] For linearly separable data, logistic regression via gradient descent converges in a finite number of steps.

Solution: The correct one is 'Logistic regression models the probability of an instance belonging to a particular class, while linear regression predicts class labels directly'.

1. **逻辑回归建模的是样本属于某一类的概率，而线性回归是直接预测类别标签。**
2. 逻辑回归的"非线性"只存在于输出空间（概率的 S 形曲线），但在**原始特征空间中，决策边界仍然是线性的**（一个超平面）。
3. 线性回归在平方损失下确实有闭式解（正规方程、伪逆）；但逻辑回归用的是对数似然 / 交叉熵损失，这是一个凸优化问题，但**没有解析解**，只能靠数值方法（梯度下降、牛顿法、拟牛顿法）来解。
4. 在线性可分的情况下，逻辑回归的最大似然解是不存在的：权重会不断变大，损失函数会不断下降，但永远到不了有限的最优点。

Exponential Loss (AdaBoost-style):

$$L(m) = e^{-m}$$

This loss grows extremely fast when m is negative, assigning enormous penalties to confident mistakes.

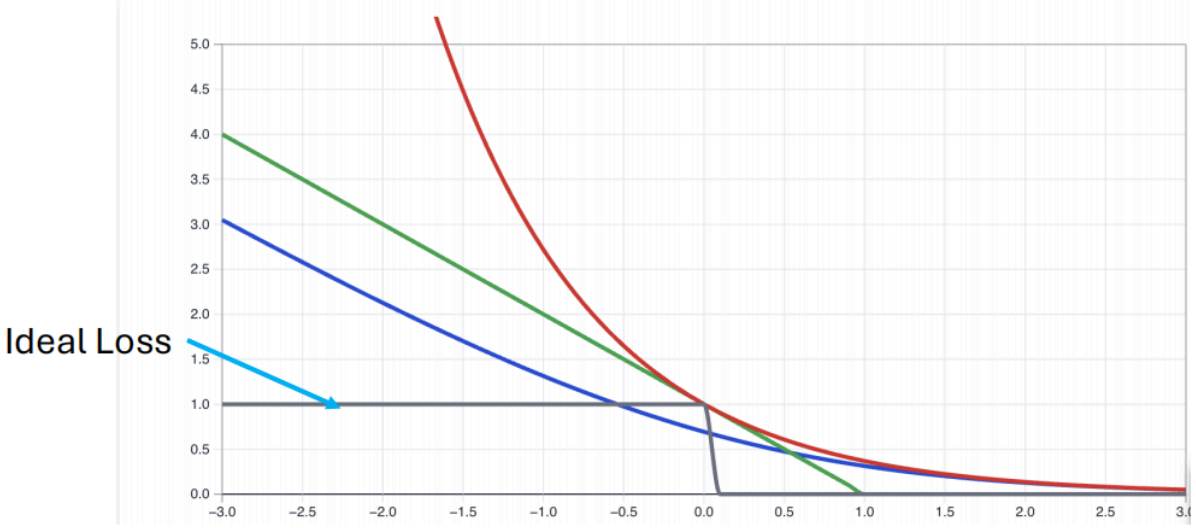
All three losses share a core property: **they strongly penalize very negative margins**.

Visual intuition: how the losses behave

When plotted as functions of the margin:

- Hinge loss is piecewise linear with a sharp "kink" at $m = 1$.
- Logistic loss is smooth, convex, and bowl-shaped.
- Exponential loss drops slowly for positive margins but explodes for negative ones.

The *shape* of the loss directly determines optimization behavior, robustness, and numerical stability.



From per-sample loss to learning an actual classifier

To learn w, we minimize the empirical risk over all training samples:

$$L(w) = \sum_i L(y_i \cdot w^\top x_i)$$

However, something important is missing. Without any constraint on w, these objectives often admit **infinitely many solutions**. If the data are linearly separable, scaling w by a large constant keeps decreasing the loss. To fix this, we introduce **regularization**:

$$L(w) = \sum_i L(y_i \cdot w^\top x_i) + \lambda ||w||_2^2$$

Regularization:

- prevents unbounded solutions,
- controls model complexity,
- and ensures well-posed optimization.

Gradient behavior and robustness

The real difference between these losses shows up in their gradients.

For logistic loss:

$$\nabla L(w) = -\frac{y}{1 + e^{yw^\top x}} x$$

For misclassified points, the gradient magnitude(幅值) is bounded(有界的). This means **no single data point can dominate training**, making logistic loss relatively robust to noise.

For exponential loss:

$$\nabla L(w) = -ye^{-yw^\top x} x$$

Here, the gradient grows exponentially for negative margins. A single outlier with a very negative margin can cause **exploding gradients**, leading to unstable training.

Hinge loss sits in between: linear penalty, sparse gradients, and robustness, but non-differentiability at the margin boundary.

Convexity and optimization guarantees

With ℓ_2-regularization, logistic loss becomes:

- convex,
- smooth (differentiable everywhere),
- and has a unique global minimum.

This guarantees that gradient descent converges to the same solution regardless of initialization. Hinge loss is convex but not smooth. Exponential loss is convex but numerically dangerous due to its gradient behavior. This explains why logistic regression is often the “default safe choice” in practice.

SVM

Linear classifiers and decision geometry

We start with a linear decision function

$$f(x) = w^\top x + b,$$

and predict labels by

$$y = \text{sign}(f(x)), \quad y \in \{+1, -1\}.$$

The equation $w^\top x + b = 0$ defines a hyperplane that separates the input space into two half-spaces. The vector w is normal (perpendicular(垂直的)) to the hyperplane, and its direction determines which side is classified as positive. At this point, infinitely many hyperplanes can separate the same data. The central question is: **Which separating hyperplane should we choose?**

Distance to a hyperplane and the margin

For simplicity, consider $b = 0$. The (signed) distance from a point x_i to the hyperplane $w^\top x = 0$ is $\frac{w^\top x_i}{|w|}$. Including the label y_i, we define the **functional margin**

$$\gamma_i = y_i(w^\top x_i).$$

- $\gamma_i > 0$: correctly classified
- $\gamma_i < 0$: misclassified
- larger γ_i: more confident classification

The **geometric margin** is obtained by normalizing with $|w|$.

The maximum margin principle

The key idea of SVM is extremely geometric:

> The best hyperplane is the one that is as far as possible from the closest data points of both classes.

Formally, we want to maximize the minimum margin:

$$\max_{||w||=1} \min_i y_i(w^\top x_i).$$

This is a max–min problem: we push the hyperplane away from *all* points, but the closest ones (later called **support vectors**) determine the solution.

Reformulating the optimization problem

Instead of fixing $||w|| = 1$, we exploit scale invariance. We can rescale w so that

$$\min_i y_i(w^\top x_i) = 1.$$

Under this normalization, maximizing the margin becomes equivalent to minimizing the norm of w:

$$\min ||w||^2 \quad \text{s.t. } y_i(w^\top x_i) \geq 1.$$

This is the **hard-margin SVM** formulation. It assumes that the data are perfectly linearly separable.

The real world: soft-margin SVM

Real data are noisy. Perfect separation is usually impossible or undesirable. To allow violations, we introduce **slack(松弛) variables** $\xi_i \geq 0$:

$$\min ||w||^2 + \lambda \sum_i \xi_i$$

subject to

$$y_i(w^\top x_i) \geq 1 - \xi_i.$$

Interpretation of ξ_i:

- $\xi_i = 0$: correctly classified with sufficient margin
- $0 < \xi_i < 1$: inside the margin
- $\xi_i > 1$: misclassified

This objective trades off margin maximization and constraint violations. The parameter λ controls how much we tolerate errors. At this point, notice something important:

$$\xi_i = \max(0, 1 - y_i w^\top x_i),$$

which is exactly the **hinge loss**. So soft-margin SVM is regularized empirical risk minimization with hinge loss.

Why introduce the Lagrangian?

The primal problem has constraints. To solve it efficiently, we construct the Lagrangian:

$$L_P = \frac{1}{2}||w||^2 + \lambda \sum_i \xi_i - \sum_i \alpha_i [y_i(w^\top x_i) - 1 + \xi_i] - \sum_i \mu_i \xi_i,$$

with $\alpha_i \geq 0$, $\mu_i \geq 0$.

We minimize L_P with respect to the primal(基本) variables (w, ξ). Taking derivatives gives:

$$\frac{\partial L_P}{\partial w} = 0 \Rightarrow w = \sum_i \alpha_i y_i x_i,$$

$$\frac{\partial L_P}{\partial \xi_i} = 0 \Rightarrow \alpha_i \leq \lambda.$$

Already here, a deep structural fact appears: **the optimal weight vector is a linear combination of training points.**

Our goal is to minimize L with respect to the primal variables $(\boldsymbol{w}, \boldsymbol{\xi})$.

$$1.\ \frac{\partial L_P}{\partial \boldsymbol{w}} = 0 \implies \boldsymbol{w} = \sum_{i=1}^{N} \alpha_i y_i \boldsymbol{x}_i$$

$$2.\ \frac{\partial L_P}{\partial \xi_i} = 0 \implies \lambda - \alpha_i - \mu_i = 0 \qquad \alpha_i \in [0, \lambda]$$

$$L_P = \frac{1}{2}\|\boldsymbol{w}\|^2 + \lambda \sum \xi_i - \sum \alpha_i y_i (\boldsymbol{w}^T \boldsymbol{x}_i) + \sum \alpha_i - \sum \alpha_i \xi_i - \sum \mu_i \xi_i$$

Grouping the ξ_i terms:

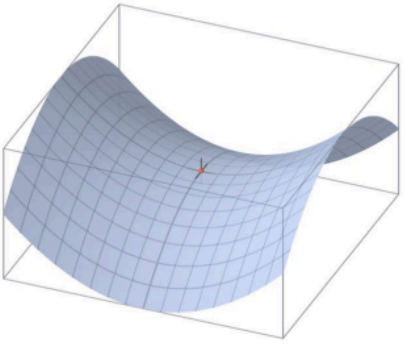
$$L_P = \frac{1}{2}\|\boldsymbol{w}\|^2 - \sum \alpha_i y_i (\boldsymbol{w}^T \boldsymbol{x}_i) + \sum \alpha_i + \sum \xi_i (\lambda - \alpha_i - \mu_i)$$

$$L_P = \frac{1}{2}\|\boldsymbol{w}\|^2 - \sum \alpha_i y_i (\boldsymbol{w}^T \boldsymbol{x}_i) + \sum \alpha_i + \sum \underbrace{\xi_i (\lambda - \alpha_i - \mu_i)}_{\text{This is zero!}}$$

$$G(w, \alpha) = \min_w \max_\alpha \frac{1}{2}||w||^2 - \sum \left(y_i \cdot (\boldsymbol{w}^T \boldsymbol{x}_i) - 1\right)\alpha_i$$

$$\max_\alpha \min_w G(w, \alpha) \leq \min_w \max_\alpha G(w, \alpha)$$

$$\max_\alpha W(\alpha) \leq \min_w L_P(w)$$



Because the function G is convex with respect to w and concave with respect to α

$$L_P = \frac{1}{2}\|\boldsymbol{w}\|^2 - \sum \alpha_i y_i (\boldsymbol{w}^T \boldsymbol{x}_i) + \sum \alpha_i + \sum \underbrace{\xi_i(\lambda - \alpha_i - \mu_i)}_{\text{This is zero!}}$$

$$\sum \alpha_i y_i (\boldsymbol{w}^T \boldsymbol{x}_i) = \boldsymbol{w}^T \left(\sum \alpha_i y_i \boldsymbol{x}_i \right) \qquad \text{Recall:} \quad \boldsymbol{w} = \sum_{i=1}^{N} \alpha_i y_i \boldsymbol{x}_i$$

$$L_P = \frac{1}{2}\|\boldsymbol{w}\|^2 - \|\boldsymbol{w}\|^2 + \sum \alpha_i = \sum \alpha_i - \frac{1}{2}\|\boldsymbol{w}\|^2$$

$$\begin{aligned} \|\boldsymbol{w}\|^2 &= \left(\sum_i \alpha_i y_i \boldsymbol{x}_i \right)^T \left(\sum_j \alpha_j y_j \boldsymbol{x}_j \right) \\ &= \sum_i \sum_j \alpha_i \alpha_j y_i y_j (\boldsymbol{x}_i^T \boldsymbol{x}_j) \end{aligned}$$

From primal to dual: why it works

After eliminating w and ξ, we obtain the dual problem:

$$\max_{\alpha} W(\alpha) = \sum_i \alpha_i - \frac{1}{2} \sum_{i,j} \alpha_i \alpha_j y_i y_j (x_i^\top x_j),$$

subject to

$$0 \le \alpha_i \le \lambda.$$

Key properties:

- The dual objective is **concave** in α.
- Most α_i are zero → **sparsity**.
- Only points with $\alpha_i > 0$ matter → **support vectors**.

Geometrically, only the closest points to the decision boundary shape the classifier.

Why the dual formulation matters

The dual formulation depends on data only through inner products $x_i^\top x_j$. This makes two things possible:

1. Efficient optimization in high dimensions
2. The kernel trick (replacing inner products with kernels)

This is why SVMs scale with the number of samples, not the feature dimension.

Can SVM be solved analytically?

The short answer is: **no closed-form analytical solution exists**, even though the problem is convex. The reasons are structural rather than accidental.

First, the SVM objective is a **constrained optimization problem** with inequality constraints.

Second, the dual variables are subject to **box constraints** $0 \leq \alpha_i \leq \lambda$, which prevents simple linear-algebraic solutions.

Third, in high dimensions, directly manipulating matrices (e.g., inversion) would cost $O(N^3)$, which is computationally infeasible for large datasets.

Finally, although the problem is a quadratic program (二次规划，QP) and well-studied, QP problems generally do not admit closed-form solutions.

So SVM is *theoretically clean but computationally nontrivial*.

The practical solution: numerical optimization

In practice, SVM is solved numerically. Given the data and the regularization parameter λ, we use a **QP solver** (most famously SMO: Sequential Minimal Optimization) to obtain

$$\alpha^* = (\alpha_1^*, \ldots, \alpha_N^*).$$

Once the optimal dual variables are found, the primal solution is recovered as

$$w^* = \sum_{i=1}^{N} \alpha_i^* y_i x_i.$$

A crucial structural insight emerges here: **Only points with $\alpha_i^* > 0$ contribute to w^*.**

These points are exactly the **support vectors**. All other training points are irrelevant at test time.

Re-interpreting SVM through Gradient Descent

Now comes the conceptual twist.

Instead of viewing SVM as a constrained optimization problem, we can rewrite it as an **unconstrained objective**:

$$J(w) = \frac{1}{2}||w||^2 + \lambda \sum_{i=1}^{N} \max(0, 1 - y_i w^\top x_i).$$

This is simply: ℓ_2 regularization; plus hinge loss. In other words, **soft-margin SVM is regularized empirical risk minimization**.

Subgradient structure of the hinge loss

The hinge loss is not differentiable everywhere, but it admits a subgradient. For a single sample x_i:

$$\nabla_w L_i = \begin{cases} 0, & \text{if } y_i w^\top x_i \geq 1 \\ -y_i x_i, & \text{if } y_i w^\top x_i < 1 \end{cases}$$

Therefore, the gradient of the full objective is:

$$\nabla_w J = w + \lambda \sum_{i: y_i w^\top x_i < 1} (-y_i x_i).$$

This equation hides an important fact: **Only margin-violating points influence the update.** Points that are correctly classified *and* far enough from the boundary exert **zero force** on the model. This is the gradient-descent version of the support vector principle.

Connection between GD and support vectors

Although gradient descent does not explicitly solve the dual problem, it implicitly discovers the same structure:

- Points outside the margin disappear from the gradient
- Points on or inside the margin dominate learning
- The final solution depends only on a small subset of samples

So **sparsity is not an artifact(假象) of duality**. It is a geometric consequence of the hinge loss.

Prediction rule and decision boundary

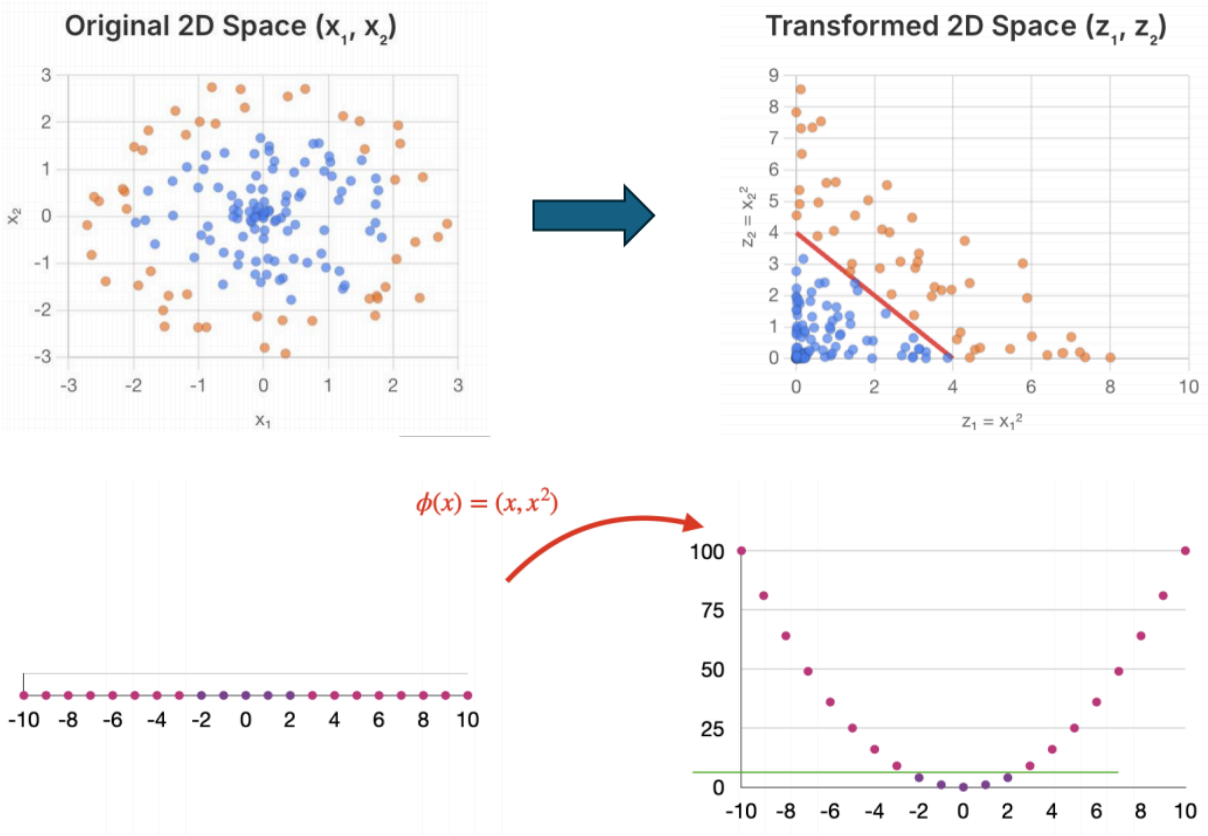
Once w^* is learned, classification is performed by:

$$f(x) = \text{sign}(w^{*\top} x).$$

Even though training involved all data, prediction depends only on support vectors through w^*. This explains why SVMs are both:

- memory efficient at inference time
- robust to irrelevant training points

Kernel Methods



Why kernels appear in SVM at all

Recall the dual form of SVM:

$$W(\alpha) = \sum_i \alpha_i - \frac{1}{2} \sum_{i,j} \alpha_i \alpha_j y_i y_j (x_i^\top x_j).$$

A critical observation is that **data points appear only through inner products** $x_i^\top x_j$. This means that if we replace each input x by a feature map

$$\phi(x) \in \mathbb{R}^m$$

the dual objective depends only on

$$\phi(x_i)^\top \phi(x_j).$$

So SVM never needs the explicit coordinates of $\phi(x)$, only their dot products.

Feature lifting: linear becomes nonlinear

Suppose we define a nonlinear feature map

$$\phi : \mathbb{R}^n \to \mathbb{R}^m, \quad m > n.$$

Example (second-order polynomial features):

$$x = (x_1, x_2), \quad \phi(x) = (x_1^2, \sqrt{2}x_1x_2, x_2^2).$$

In this lifted(升维后的) space:

- nonlinear structures (e.g., circles) become linearly separable,
- but computing $\phi(x)$ explicitly becomes expensive when m is large.

This creates a tension: **geometric power vs. computational feasibility**.

The kernel trick: collapsing the computation

Now comes the key insight. For the polynomial feature map above:

$$\phi(x)^\top \phi(z) = (x_1z_1 + x_2z_2)^2 = (x^\top z)^2.$$

Define

$$K(x, z) = (x^\top z)^2.$$

Then:

$$K(x, z) = \phi(x)^\top \phi(z),$$

without ever computing (\phi(x)).

This is the kernel trick: **Compute inner products in feature space using only operations in input space.** The computational gain is enormous, especially when m is very large or infinite.

Kernel functions and Gram matrices

A kernel is a function

$$k : X \times X \to \mathbb{R}$$

that behaves like an inner product. Given data $x_1, \ldots, x_N$, the **kernel (Gram) matrix** is

$$K_{ij} = k(x_i, x_j).$$

In SVM, the dual problem becomes:

$$\max_\alpha \sum_i \alpha_i - \frac{1}{2} \sum_{i,j} \alpha_i \alpha_j y_i y_j K_{ij}.$$

The dimensionality of the feature space **never appears explicitly**.

Beyond polynomials: infinite-dimensional kernels

Polynomial kernels increase dimensionality finitely. But what if we want *unbounded expressive power*? The most important example is the **Radial Basis Function (RBF) kernel**:

$$k(x, z) = \exp(-\gamma||x - z||^2).$$

This kernel corresponds to a feature map into an **infinite-dimensional space**. Using a Taylor expansion, one can show:

$$k(x, z) = \sum_{\alpha} \phi_\alpha(x)\phi_\alpha(z),$$

where the index α ranges over infinitely many monomials(单项式). Thus, RBF kernels implicitly perform linear classification in an infinite-dimensional Hilbert space.

Euclidean space vs. Hilbert space

This motivates a shift in perspective. A **Hilbert space** is:

- an inner-product space,
- possibly infinite-dimensional,
- complete(完备的) (limits of Cauchy sequences(柯西列的极限) exist).

Kernel methods do not require explicit coordinates. They require only:

- an inner product,
- and the geometry induced by it.

In this sense, kernels generalize Euclidean geometry to functional spaces.

When is a function a valid kernel?

Not every function $k(x, z)$ is allowed. **Mercer's condition** states that a function is a valid kernel if:

1. It is symmetric: $k(x, z) = k(z, x)$.
2. For any finite set $x_1, \ldots, x_N$, the Gram matrix K is **positive semidefinite (PSD)**:

$$\sum_{i,j} c_i c_j K_{ij} \geq 0 \quad \forall c \in \mathbb{R}^N.$$

This guarantees the existence of a feature map ϕ such that

$$k(x, z) = \phi(x)^\top \phi(z).$$

Why symmetry and PSD are necessary

Geometrically, PSD ensures that the kernel behaves like a true inner product. Algebraically, it guarantees convexity of the SVM dual problem. Statistically, it prevents “negative similarity” from breaking consistency. Without PSD, optimization may fail or yield meaningless geometry.

Kernel composition rules

A powerful consequence of Mercer’s theorem is **closure**. If K_1 andK_2 are valid kernels, then so are:

- cK_1 for $c > 0$,
- $K_1 + K_2$,
- $K_1 \cdot K_2$,
- $\exp(K_1)$.

This allows us to *engineer kernels* tailored(**定制**) to specific data structures.

Kernel methods beyond SVM

Kernelization is not unique to SVM. For example, ridge regression:

$$\min_w ||y - Xw||^2 + \lambda||w||^2$$

has the closed-form solution

$$w = (X^\top X + \lambda I)^{-1} X^\top y.$$

Using kernels, this becomes **kernel ridge regression**, where predictions depend only on kernel evaluations. This shows that kernels are a **general computational principle**, not an SVM trick.

7 Deep Learning

From Binary Classification to Multi-class

We start from the simplest setting: **binary classification**, where a classifier answers a yes/no question such as “Is class A preferred to class B?” A natural idea for multi-class problems is to decompose them into multiple binary classifiers, for example:

- one-vs-one (pairwise comparisons),
- or one-vs-rest.

However, binary decisions are **local**, while multi-class prediction requires a **global, consistent decision**. Pairwise classifiers may contradict each other. Even if every binary classifier seems reasonable in isolation, their outputs may:

- violate transitivity,
- form cycles,
- or fail to produce a valid global ordering.

This shows that **multi-class classification is not simply a collection of independent binary problems**. We need a model that reasons about all classes *jointly*.

Multiple classifiers as score functions

To move beyond yes/no decisions, we introduce the idea of **scoring functions**. Assume we have:

- input space $X \subset \mathbb{R}^d$,
- K classes,
- K real-valued functions

$$f_k : X \to \mathbb{R}, \quad k = 1, \ldots, K$$

Each function produces a **score** measuring how compatible(兼容的) the input is with class k. Prediction is done by:

$$\hat{y}(x) = \arg \max_{k=1,\ldots,K} f_k(x)$$

This is a crucial conceptual shift:

- We are no longer asking “Is class A better than class B?”
- We ask “Which class gets the highest score overall?”

Decision boundaries now arise from **score equality**:

$$f_i(x) = f_j(x)$$

These boundaries naturally partition the input space into regions assigned to different classes.

Linear multi-class models and geometric intuition

If each f_k is linear:

$$f_k(x) = w_k^\top x + b_k$$

then:

- decision boundaries are linear,
- regions are convex,
- and all class decisions interact in a single geometric structure.

This avoids the inconsistency problems of pairwise voting: **all classes are compared simultaneously in the same score space**.

Image classification as a multi-class problem

Now we move to real data: **image classification**. Each training example consists of:

$$(x^{(i)}, y^{(i)})$$

where:

- $x^{(i)}$ is an image,
- $y^{(i)} \in 1, \ldots, K$ is its class label (e.g., fish, bear, chameleon).

We want to learn a function:

$$f_\theta : X \to \mathbb{R}^K$$

that maps an image to **K class scores**.

Encoding labels: from class names to one-hot vectors

Class labels are categorical, but optimization needs numbers. We encode labels as **one-hot vectors**:

- class 1 → $[1, 0, 0, \ldots]$
- class 2 → $[0, 1, 0, \ldots]$
- class k → $[0, \ldots, 1, \ldots, 0]$

This representation allows us to:

- compare predictions with ground truth,
- define differentiable loss functions,
- train models using gradient-based optimization.

Predictions as probability distributions

Instead of raw scores, we interpret outputs as **probabilities** over classes:

$$\hat{y} = (\hat{y}_1, \dots, \hat{y}_K), \quad \sum_k \hat{y}_k = 1$$

The model expresses *uncertainty*:

- not just which class is chosen,
- but how confident it is relative to all other classes.

Measuring error with cross-entropy (entropy loss)

To train the model, we need a loss function that compares:

- predicted distribution $\hat{y}$,
- true distribution y (one-hot).

The standard choice is **cross-entropy**:

$$\mathcal{L}(\hat{y}, y) = -\sum_{k=1}^{K} y_k \log \hat{y}_k$$

Interpretation:

- only the true class contributes to the loss,
- confident wrong predictions are penalized heavily,
- correct and confident predictions receive low loss.

This loss measures **"how much better the model could have done"** by shifting probability mass toward the true class.

Final optimization objective

Training becomes a single, clean optimization problem:

$$\min_{\theta} \sum_{i=1}^{N} \mathcal{L}(f_\theta(x^{(i)}), y^{(i)})$$

Key properties:

- all classes trained jointly,
- no pairwise inconsistencies,
- global structure enforced by the loss.

Softmax Regression: a coherent view

From scores to probabilities

In multi-class classification, the model does **not** directly output a label. Instead, it outputs a **vector of scores** (also called *logits*):

$$f_\theta : X \to \mathbb{R}^K, \quad z = f_\theta(x)$$

Each component z_k represents how compatible the input x is with class k. These scores are **unbounded** and **not probabilities**. To convert scores into a valid probability distribution, we apply the **softmax function**:

$$\hat{y} = \mathrm{softmax}(z), \quad \hat{y}j = \frac{e^{z_j}}{\sum_{k=1}^{K} e^{z_k}}$$

Properties:

- $\hat{y}_j \geq 0$
- $\sum_j \hat{y}_j = 1$

Thus, softmax maps arbitrary real-valued scores into a **probability mass function** over classes.

Probabilistic interpretation

After softmax, the output has a clean probabilistic meaning:

$$\hat{y} = \left[P_\theta(Y = 1 \mid X = x), \ldots, P_\theta(Y = K \mid X = x)\right]$$

This is not a metaphor—it is a **conditional categorical distribution**. The model explicitly represents uncertainty across all classes. Prediction is still:

$$\hat{y}_{\text{pred}} = \arg\max_k \hat{y}_k$$

but training uses the **full distribution**, not just the winner.

Cross-entropy loss: learning by likelihood

To train the model, we measure how well the predicted distribution $\hat{y}$ matches the true distribution y. The standard loss is **cross-entropy**:

$$H(y, \hat{y}) = -\sum_{k=1}^{K} y_k \log \hat{y}_k$$

Because y is one-hot, this simplifies to:

$$H(y, \hat{y}) = -\log \hat{y}_{\text{true class}}$$

Interpretation:

- The loss is the **negative log-likelihood** of the correct class.
- Confident wrong predictions are punished heavily.
- Confident correct predictions yield near-zero loss.

This makes training a **maximum likelihood estimation** problem.

The learning objective

Given a dataset $\{(x^{(i)}, y^{(i)})\}_{i=1}^{N}$, training solves:

$$\theta^* = \arg\min_{\theta} \sum_{i=1}^{N} H\left(y^{(i)}, \text{softmax}(f_\theta(x^{(i)}))\right)$$

This model is also known as: Multinomial(多项式) Logistic Regression Logistic regression is no longer binary here—it generalizes naturally through softmax.

Why softmax is better than pairwise classification

Softmax regression:

- compares all classes **simultaneously**,
- enforces a **global normalization constraint**,
- avoids inconsistent pairwise decisions,
- provides calibrated(校准) probabilities.

This directly fixes the issues you saw earlier with binary-to-multi-class constructions.

Numerical instability: the softmax explosion

Softmax involves exponentials. If logits are large, this causes overflow:

$$z = [1000, 1001, 999] \quad \Rightarrow \quad e^z \approx [\infty, \infty, \infty]$$

This is not a theoretical issue—it **will crash real code**.

The Softmax Trick (log-sum-exp trick)

Key identity:

$$\text{softmax}(z) = \text{softmax}(z - c) \quad \text{for any constant } c$$

We choose:

$$c = \max_k z_k$$

Example:

$$z = [1000, 1001, 999] \Rightarrow z' = [-1, 0, -2]$$

Now

$$e^{z'} = [0.3679, 1.0, 0.1353]$$

$$\text{softmax}(z) = [0.245, 0.665, 0.09]$$

Same probabilities, **no overflow**. This trick is mandatory in real implementations.

The History of Artificial Intelligence

1. The Dawn of AI (1940s - 1960)

1943 – Artificial Neuron Model

- Proposed by Warren McCulloch and Walter Pitts.
- Established the first mathematical model of a neuron: it takes binary inputs, sums them, and outputs 1 if the total exceeds a threshold, otherwise 0.
- This laid the foundation for artificial neural networks.

1957 – The Perceptron

- Designed by Frank Rosenblatt as a single-layer neural network that could “learn” by adjusting its weights.
- Considered the beginning of true artificial intelligence.

2. The First AI Winter (1969)

Event: *Perceptrons* by Minsky & Papert

- Demonstrated that single-layer perceptrons cannot solve nonlinear problems (e.g., XOR).
- This finding led to declining enthusiasm and funding, plunging(使……突然陷入) AI research into its first “winter.”

3. The Revival and Backpropagation (1980s - 1990s)

1986 – The Backpropagation Algorithm

- Introduced by Rumelhart, Hinton, and Williams.

- Used the chain rule to compute the gradient of the error for each weight, allowing networks to learn complex nonlinear patterns.
- Solved the perceptron's limitations and sparked(引发) the "Connectionist" revival.
- The mathematical foundation was actually laid earlier by Seppo Linnainmaa in 1970 with the concept of *automatic differentiation*.

1989 – LeNet for Digit Recognition

- Yann LeCun applied backpropagation to convolutional neural networks (CNNs).
- LeNet-5 was used commercially for handwritten digit recognition, proving the real-world utility of neural networks.

1997 – LSTM and the Vanishing Gradient Problem

- Proposed by Hochreiter and Schmidhuber as the Long Short-Term Memory (LSTM) network.
- Solved the vanishing gradient problem in deep networks through gate mechanisms, enabling RNNs to learn long-term dependencies.

4. The Second AI Winter (1990s - 2006)

Rise of SVMs and Random Forests

- In 1995, Vapnik and Cortes introduced the Support Vector Machine (SVM).
- SVMs were more efficient, mathematically grounded, and robust than neural networks of the time.
- Due to computational limitations and training difficulties, neural networks fell out of favor once again.

The "AI Godfathers" Persist

- Geoffrey Hinton, Yoshua Bengio, and Yann LeCun continued deep learning research through the "winter."
- Their persistence was later recognized when they received the 2024 Nobel Prize for foundational contributions to machine learning and neural networks.

5. The "Big Bang" Era (2006 - 2012)

Three Pillars: ImageNet, GPU, and Technical Breakthroughs

1. **Pretraining and Deep Belief Networks (2006, Hinton)**
    - Introduced layer-wise pretraining to stabilize the training of deep networks.
2. **ReLU Activation Function (2010, Nair & Hinton)**
    - Accelerated training and mitigated the vanishing gradient problem.

3. **AlexNet (2012, Krizhevsky, Sutskever, Hinton)**
    - Leveraged GPUs, ReLU, and Dropout to dominate the ImageNet competition.
    - Achieved a top-5 error rate of 16.4%, marking the full revival of deep learning.

6. The Deep Learning Revolution (2014 – Present)

2014 – Generative Adversarial Networks (GANs)

- Introduced by Ian Goodfellow.
- A novel architecture where two networks (generator and discriminator) compete, enabling AI to generate realistic images, audio, and video.

2017 – Transformer ("Attention is All You Need")

- Proposed by Vaswani et al., introducing a model based entirely on self-attention.
- Replaced recurrent architectures, greatly improving parallelism and efficiency.
- Became the foundation for modern large language models (LLMs).

2020 – The GPT and Generative AI Era

- Introduced by Radford et al. with the Generative Pretrained Transformer (GPT).
- LLMs and foundation models became the core of AI research, extending from text generation to images, code, and video synthesis.

7. Trends and the Future (2016 – Present)

- Data, computational power, and algorithms now advance synergistically(协同地), driving exponential AI growth.
- Deep learning adoption within Google and other tech giants has skyrocketed.
- Neural networks now serve as a universal framework for defining functions, evaluating their performance, and optimizing for the best representation.

Training Deep Neural Networks with Fully Connected Layers

From perceptron to modern neurons

A fully connected (FC) layer computes a linear transformation of its input:

$$z_j = \sum_i w_{ij} x_i + b_j$$

wherew_{ij} are weights and b_j is the bias. If we stack only linear layers, the entire network collapses into a single linear transformation. Therefore, **non-linearity is**

essential. The simplest historical non-linearity is the **perceptron activation**:

$$g(z) = \begin{cases} 1, & z > 0 \\ 0, & \text{otherwise} \end{cases}$$

This enables classification but is **non-differentiable**, which prevents gradient-based learning in deep networks.

Smooth activations and backpropagation

To train deep networks with gradient descent, we need **differentiable activation functions**. A classic choice is the **sigmoid**:

$$\sigma(z) = \frac{1}{1 + e^{-z}}, \quad \sigma'(z) = \sigma(z)(1 - \sigma(z))$$

Key observation:

- when $z \approx 0$, gradient is maximal (≈ 0.25);
- when $|z|$ is large, gradients vanish.

Similarly, **tanh** behaves like a centered sigmoid:

- output in $[-1, 1]$,
- same saturation problem for large $|z|$.

Vanishing gradients in deep networks

In a deep network, gradients propagate via the chain rule:

$$\frac{\partial L}{\partial w_1} = \frac{\partial L}{\partial a_N} \cdot \prod_{i=1}^{N} \sigma'(z_i) \cdot \frac{\partial a_1}{\partial w_1}$$

Even in the *best* case for sigmoid ($\sigma'(z) = 0.25$):

$$(0.25)^{10} \approx 10^{-6}$$

This explains why early deep fully connected networks were **extremely hard to train**: gradients decay exponentially with depth.

Early attempts: RBM and unsupervised pretraining

Before ReLU became standard, researchers explored **Restricted Boltzmann Machines (RBMs)** and **Deep Boltzmann Machines (DBMs)**. RBMs define an energy-based probabilistic model:

$$P(v;\theta) = \frac{1}{Z(\theta)} \sum_{h} \exp(-E(v,h;\theta))$$

However:

- computing the partition(分区) function $Z(\theta)$ is intractable(极其困难),
- training relies on approximations (e.g. contrastive divergence),
- scaling to large networks is difficult.

The ReLU revolution

The breakthrough came with the **Rectified(整流) Linear Unit (ReLU)**:

$$g(z) = \max(0, z)$$

Why ReLU works:

- no saturation for positive values,
- gradient is constant 1 when active,
- extremely cheap to compute,
- empirically accelerates convergence.

Drawback:

- neurons can “die” if stuck in the negative region (zero gradient).

Despite this, ReLU became the **default activation** in modern deep networks.

Variants: Leaky ReLU and PReLU

To fix dead neurons, **Leaky ReLU** introduces a small slope:

$$g(z) = \begin{cases} z, & z \geq 0 \\ \alpha z, & z < 0 \end{cases} \quad (\alpha \approx 0.01)$$

Properties:

- non-zero gradient everywhere,
- still computationally efficient,
- empirically more robust in some settings.

If α is learned, the activation is called **PReLU**.

Depth: how deep is “deep”?

Historically:

- AlexNet (2012): ~8 layers,
- VGG (2014): 16–19 layers,
- GoogLeNet (2014): 22+ layers.

The key lesson is **not** the exact number of layers, but that:

- deeper networks learn hierarchical representations,
- training deep models required better activations and optimization.

😈 **训练一个全连接神经网络的计算复杂度，在单次迭代中是关于宽度（每层神经元数）的二次函数。**层数增加，复杂度大致是线性叠加；但一层里神经元一多，权重矩阵是 $n \times n$，计算量自然平方增长。

Expressivity vs trainability

The **Universal Approximation Theorem(通用逼近定理)** states:

> A neural network with a single hidden layer can approximate any continuous function, given enough neurons.

However:

- this says nothing about *efficiency*,
- nothing about *optimization*,
- nothing about *generalization*.

Deep networks trade width for depth, gaining:

- compositional structure,
- parameter efficiency,
- better inductive bias.

Recent theory (e.g. Neural Tangent Kernel) explains why overparameterized deep networks are still trainable.

Many Layers and Representation Learning

Stacking layers: from input to output

A multi-layer perceptron (MLP) consists of successive transformations:

$$h = g(W_1 x + b_1), \quad y = g(W_2 h + b_2)$$

More generally, for L layers:

$$h^{(l)} = g(W_l h^{(l-1)} + b_l)$$

The parameters of the network are:

$$\theta = W_1, \ldots, W_L, b_1, \ldots, b_L$$

Each layer produces a **new representation** of the same input.

What "representation" really means

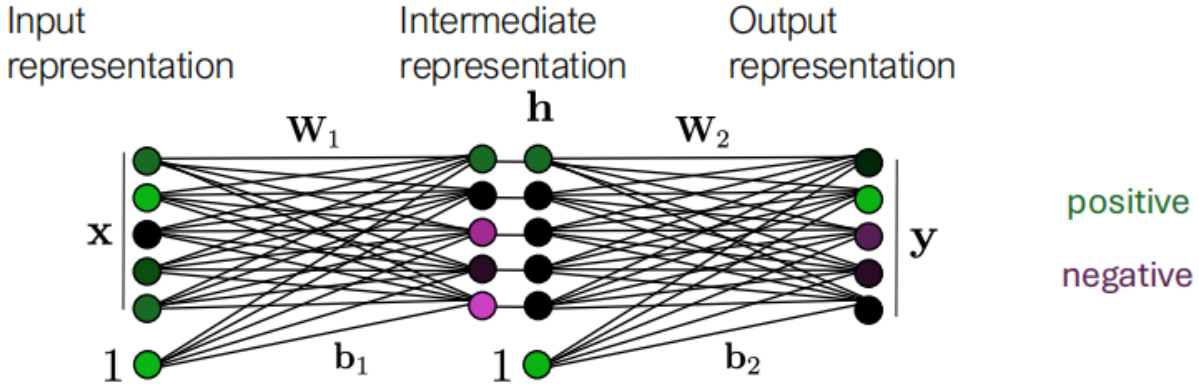
Neurons do not store symbols. They store **patterns of activation**. In the diagrams:

- green nodes indicate positive activation,
- purple nodes indicate negative activation,
- black nodes are near zero.

As information flows forward:

- different hidden units respond to different aspects of the input,
- activations become increasingly selective,
- the representation becomes easier to separate at the output.

This is why we call deep learning **representation learning**.



How non-linearity creates expressive power

If g were linear, multiple layers would collapse into one. Non-linearity allows the network to:

- bend decision boundaries,
- carve input space into regions,
- compose simple functions into complex ones.

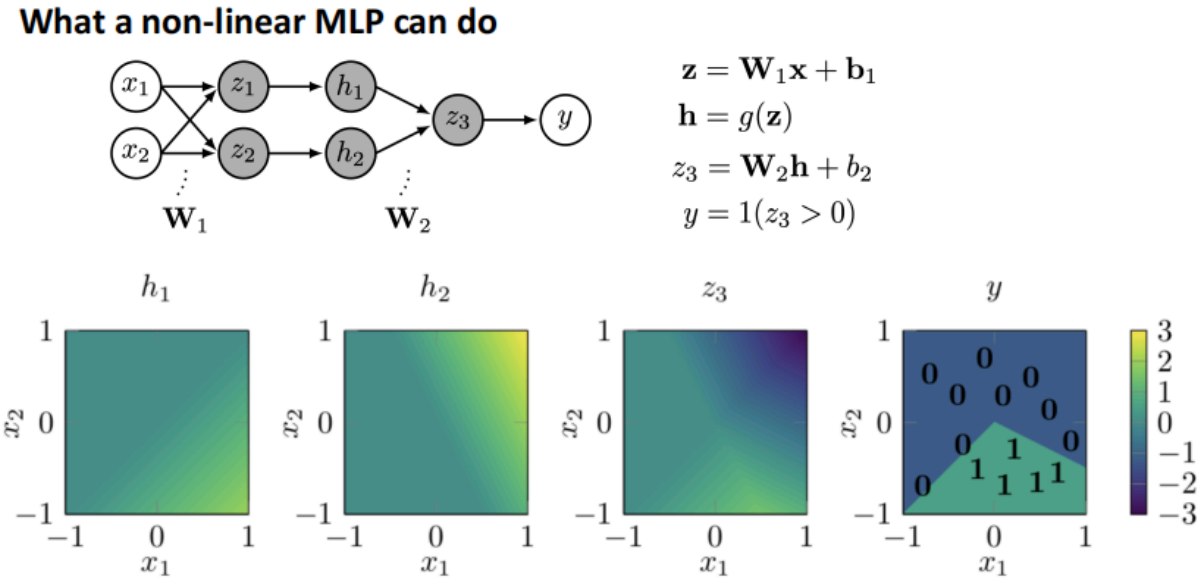
A non-linear MLP can implement decision regions that are **not linearly separable**, even in low dimensions.

Example: what a non-linear MLP can do

Each hidden unit defines a soft partition of the input space. By combining multiple hidden units:

- the network constructs piecewise-linear regions,
- decision boundaries become curved and complex,
- classification accuracy improves dramatically.

Depth enables **compositional structure**: simple features → combined features → task-level decisions.



Connectivity: fully connected vs locally connected

A **fully connected layer** connects every input unit to every output unit:

- maximum flexibility,
- maximum number of parameters,
- ignores spatial structure.

A **locally connected layer** uses sparse connectivity:

- each unit sees only a local neighborhood,
- far fewer parameters,
- strong inductive bias.

Sparse connectivity reflects the assumption that **local interactions matter more than global ones**.

Convolution as structured connectivity

Convolution is a special case of local connectivity with **weight sharing**. In continuous form:

$$(f * g)(n) = \int f(t)g(n - t), dt$$

In discrete form:

$$(f * g)(n) = \sum_{\tau} f(\tau)g(n - \tau)$$

In neural networks:

- f is the input signal (image),
- g is a learnable filter,
- the same filter is applied across all spatial locations.

This introduces:

- translation equivariance,
- parameter efficiency,
- better generalization for images.

Why CNNs replace FC layers for vision

Using FC layers on images:

- destroys spatial locality,
- requires enormous parameter counts,
- overfits easily.

CNNs encode prior knowledge:

- nearby pixels are related,
- the same pattern can appear anywhere.

This architectural bias is as important as depth itself.

8 Deep Learning II

Softmax: From Scores to Probabilities

A deep neural network for classification maps an input x (e.g., an image) to a vector of scores at the last layer. Each dimension corresponds to a class, such as *dolphin*,

cat, or *clown fish*. These scores are often called **logits**. They are not probabilities yet. The predicted class is obtained by applying **argmax** to the logits:

$$\hat{y} = \arg\max_k z_k$$

However, argmax is not differentiable and therefore **cannot be used directly for training**. To train the model, we convert logits into probabilities using the **softmax function**:

$$\hat{y}_k = \frac{e^{z_k}}{\sum_j e^{z_j}}$$

Softmax ensures: **All outputs are in the range $(0, 1)$; the probabilities sum to 1.** Now the network output can be interpreted as a **categorical probability distribution** over classes.

Training Objective Over the Dataset

Given a dataset $\{(x^{(i)}, y^{(i)})\}_{i=1}^N$, the overall objective is:

$$\theta^* = \arg\min \theta \sum_{i=1}^{N} \mathcal{L}(f_\theta(x^{(i)}), y^{(i)})$$

Here:

- θ denotes all network parameters (weights and biases)
- f_θ is the forward computation of the network

This is an optimization problem over millions of parameters.

Gradient Descent

To solve the optimization problem, we use **gradient descent**:

$$\theta^{(t+1)} = \theta^{(t)} - \eta \nabla_\theta \mathcal{L}$$

Where:

- η is the learning rate
- $\nabla_\theta \mathcal{L}$ is the gradient of the loss with respect to parameters

The challenge is **computing these gradients efficiently**.

Backpropagation

Backpropagation is an efficient algorithm to compute gradients for all parameters using the **chain rule**. It consists of two phases:

- **Forward pass**: compute all intermediate activations(中间激活值)
- **Backward pass**: propagate gradients from the loss back to earlier layers

Chain Rule Fundamentals

If variables are composed sequentially:

$$y = g(x), \quad z = h(y)$$

Then:

$$\frac{dz}{dx} = \frac{dz}{dy} \cdot \frac{dy}{dx}$$

If a variable influences the output through multiple paths:

$$\frac{dz}{ds} = \frac{\partial z}{\partial x}\frac{dx}{ds} + \frac{\partial z}{\partial y}\frac{dy}{ds}$$

Backpropagation systematically applies these rules.

Gradients in a Single Neuron

For a neuron:

$$z = x_1 w_1 + x_2 w_2 + b$$

The gradients are:

$$\frac{\partial z}{\partial w_1} = x_1, \quad \frac{\partial z}{\partial w_2} = x_2$$

This shows a key intuition: **The gradient with respect to a weight equals the input connected to that weight(某个权重的梯度等于与该权重相连的输入).**

Gradients Through Activation Functions

If:

$$a = \sigma(z)$$

Then:

$$\frac{\partial \mathcal{L}}{\partial z} = \frac{\partial \mathcal{L}}{\partial a} \cdot \sigma'(z)$$

The derivative(倒数) of the activation function scales the backpropagated gradient.

Hidden Layers: Recursive(递归) Gradient Propagation

For hidden layers, gradients are computed by summing contributions from all downstream paths:

$$\frac{\partial \mathcal{L}}{\partial z} = \sigma'(z) \sum_k w_k \frac{\partial \mathcal{L}}{\partial z_k}$$

This explains:

- Why gradients flow backward
- Why vanishing/exploding gradients can occur
- Why initialization and activation choice matter

Output Layer vs Hidden Layer

- **Output layer**: gradients are computed directly from the loss
- **Hidden layers**: gradients are computed recursively from later layers

Backpropagation continues **until the input layer is reached**.

Batch, Tensor, Computation Graphs, PyTorch

Neural networks are implemented as vectorized computations over batches of data. Instead of processing one sample at a time, modern frameworks operate on tensors that stack multiple samples along the batch dimension.

Each layer transforms a tensor representation of the data. For example, a hidden layer $H \in \mathbb{R}^{N \times C}$ represents N samples described by C features.

Training minimizes the average loss over a batch:

$$L = \frac{1}{N} \sum_{i=1}^{N} \ell(x^{(i)}, y^{(i)})$$

and gradients are averaged accordingly.

Neural networks can be viewed as computation graphs: directed acyclic graphs whose nodes are differentiable operations. Forward propagation evaluates the graph, while backward propagation applies the chain rule in reverse.

Each module implements:

- a forward function
- a backward function that propagates gradients to inputs and parameters

For a linear layer $x_{\text{out}} = W x_{\text{in}}$:

$$\frac{\partial J}{\partial x_{\text{in}}} = g_{\text{out}} W, \quad \frac{\partial J}{\partial W} = x_{\text{in}} g_{\text{out}}$$

When a variable or parameter is used multiple times in the graph, gradients from all paths are summed. This naturally arises from the chain rule.

Automatic differentiation frameworks such as PyTorch build computation graphs dynamically and support higher-order differentiation by constructing graphs of gradients themselves.

PyTorch

PyTorch separates computation from parameter management by design.

A `torch::autograd::Function` defines a differentiable operation. It is **stateless** and contains **no parameters**. The forward method implements the raw computation, while the backward method explicitly defines how gradients are propagated using the chain rule.

During the forward pass, tensors required for gradient computation are stored using `ctx->save_for_backward`. During the backward pass, the function receives `grad_output`, which represents the gradient of the loss with respect to the function's output.

The backward method must return gradients for all forward inputs in the exact same order.

A `torch::nn::Module` is a stateful wrapper that **holds parameters** such as weights and biases. Its forward method typically calls a `Function` to perform the actual computation. Modules manage **parameter registration**, device placement, and serialization, and provide a user-friendly interface.

This separation allows PyTorch to implement flexible automatic differentiation over computation graphs while keeping parameter management clean and modular.

Regularizing Deep Neural Networks

Deep neural networks often contain far more parameters than training data, making overfitting a central concern. Regularization methods aim to reduce the effective capacity of the model.

Weight decay (ridge regression) adds an L2 penalty to the loss function, encouraging solutions with smaller parameter norms. This biases learning toward smoother functions and reduces sensitivity to noise.

Dropout randomly zeroes out hidden units during training. It prevents strong co-adaptation between features and can be interpreted as averaging over an exponential number of subnetworks, effectively acting as an ensemble method.

Normalization methods standardize activations by subtracting the mean and dividing by the standard deviation. This stabilizes training, improves gradient flow, and introduces invariance to scale and shift in intermediate representations.

Different normalization techniques differ in which dimensions statistics are computed over. **Batch normalization** depends on mini-batch statistics and is common in CNNs, while **layer normalization** is batch-independent and widely used in Transformer models.

> 😈 BN 在训练时，确实用当前 batch 的均值和方差；但一旦进入推理阶段，batch 的统计量是**固定的**，通常来自训练过程中累计的滑动平均（EMA）。

Together, these methods constrain parameters, representations, and optimization dynamics, thereby reducing overfitting.

Differentiable Programming

Differentiable programming is a paradigm in which programs are constructed from differentiable components and trained using gradient-based optimization.

Each program defines a mapping from inputs to outputs, and learning corresponds to adjusting parameters to optimize a loss function. Linear transformations and nonlinear activations such as ReLU and sigmoid are basic building blocks of such programs.

A differentiable program can be viewed from multiple perspectives: as a wiring graph, a mathematical equation, or a geometric transformation in representation space. These views are equivalent and interchangeable.

By composing multiple differentiable transformations, neural networks learn intermediate representations that make complex tasks such as classification easier. Training reshapes the geometry of the data so that classes become separable in learned feature spaces.

Backpropagation enables efficient gradient computation through these programs, while regularization and normalization constrain the space of functions that can be learned. Together, these elements form the foundation of modern deep learning systems.