Yale University

Mathematical Language for Machine Learning

S&DS 265 · Introductory Machine Learning · Lecture 2

Zhuoran Yang

Fall 2026

AI usage: figures were generated, and these slides polished, with the help of AI tools.

Outline

1. Vectors, Norms, and Geometry
2. Symmetric Matrices, Eigenvalues, and the SVD
3. Gradients and Curvature
4. Computation Graphs and Automatic Differentiation
5. Probability and the Gaussian

Learning Objectives

In this lecture you will learn:

1. How vectors, norms, and inner products give geometry to a data matrix;
2. How a gradient and a Hessian describe the local shape of a loss surface;
3. How automatic differentiation evaluates those derivatives on a computation graph;
4. What eigenvalues and the SVD reveal about a symmetric matrix and a general one;
5. How the multivariate Gaussian encodes means, variances, and correlations.

Outline

Vectors: Data Points, Points, and Arrows

A vector $x \in \mathbb{R}^d$ is a list of d numbers. Here that list is almost always a **data point**: one case, its d measurements as coordinates — a **feature vector**. For one credit-card customer,

$$x = (\text{balance},\ \text{income}) = (729.53,\ 44{,}361.63) \in \mathbb{R}^2.$$

Geometry gives two more readings of that same list:

A point

a location in $\mathbb{R}^d$, so two customers can be near or far

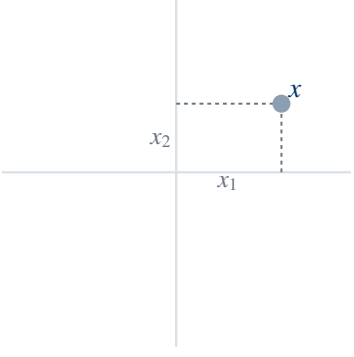
An arrow

from the origin to that point, so vectors can be added and scaled

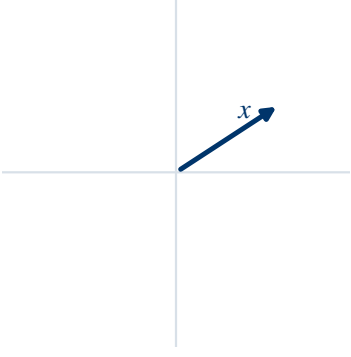
The arrow perspective is what makes $\alpha x + \beta z$ meaningful. A set closed under those operations is a **vector space**, and all such combinations of x and z form their **span**.

Visualizing a Vector as a Point and as an Arrow

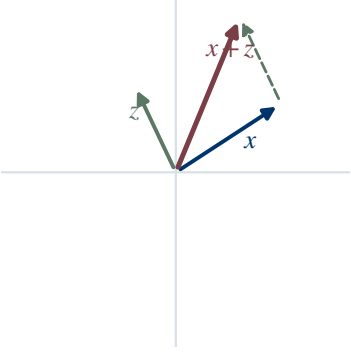
a point: a location



an arrow: a displacement



arrows add



Vectors in NumPy

```
import numpy as np
x = np.array([729.53, 44361.63])   # balance and income, one customer
w = np.array([0.004, -0.00002])    # one weight per feature
w @ x                              # inner product: one risk score
np.linalg.norm(x)                  # Euclidean length
```

`@` is matrix multiplication; on two one-dimensional arrays it is the inner product. Shapes are the first thing to check: `x.shape` is `(2,)`, not `(2,1)`.

Inner Product: Score and Angle

As a computation

$$w^\top x = \sum_{j=1}^{d} w_j \cdot x_j$$

each feature contributes $w_j \cdot x_j$ to the score

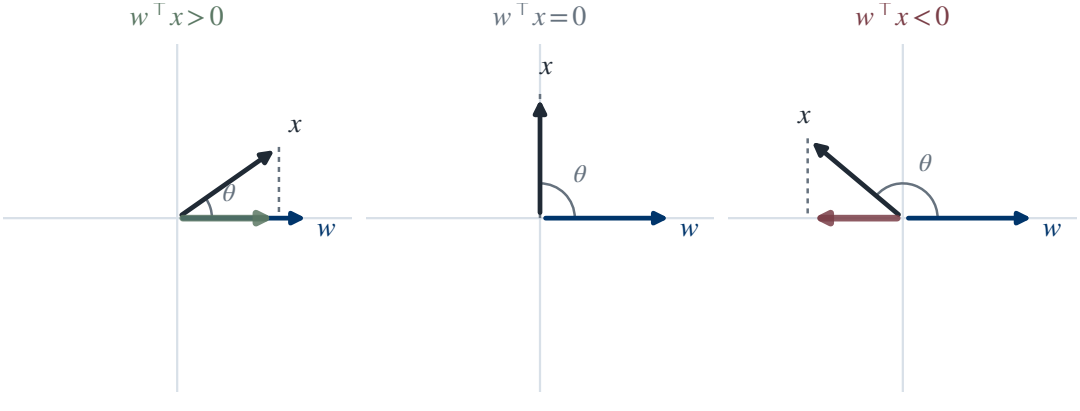
As geometry

$$w^\top x = \|w\|_2 \cdot \|x\|_2 \cdot \cos\theta$$

the score grows when x points along w

Two consequences used constantly: $w^\top x = 0$ means **orthogonal**, and $\|x\|_2^2 = x^\top x$.

Visualizing the Inner Product and the Angle



Only the part of x lying along w contributes: the coloured arrow is that component, of signed length $\|x\|_2 \cdot \cos\theta$. It points with w when $\theta < 90°$, vanishes at $90°$, and reverses beyond it.

Orthogonal Vectors and Orthonormal Bases

The inner product decides orthogonality: u and v are **orthogonal** when $u^\top v = 0$, so neither has any component along the other. A set $u_1, \dots, u_k$ is **orthonormal** when

$$u_i^\top u_j = \begin{cases} 1, & i = j \\ 0, & i \neq j \end{cases}$$

— unit length, mutually orthogonal. If $u_1, \dots, u_d$ is an orthonormal **basis** of $\mathbb{R}^d$, every x is rebuilt from inner products:

$$x = \sum_{j=1}^{d} (u_j^\top x)\, u_j.$$

Coordinates in an orthonormal basis are projections: no linear system has to be solved.

Span and Subspaces

The **span** of $v_1, \dots, v_k$ is the set of all linear combinations

$$\operatorname{span}\{v_1, \dots, v_k\} = \left\{ \sum_j \alpha_j v_j : \ \alpha \in \mathbb{R}^k \right\},$$

and any such set is a **subspace**: closed under addition and scaling, and containing the origin. Its **dimension** is the number of independent directions it contains.

- One vector spans a line; two independent vectors span a plane;
- The columns of A span its **column space** $\operatorname{col}(A) = \{Ax\}$, which is where every output of the map lives.

Vector Norms: the ℓ_p Family

For $p \geq 1$, the ℓ_p **norm** of $x \in \mathbb{R}^d$ is

$$\|x\|_p = \Big(\sum_j |x_j|^p\Big)^{1/p}.$$

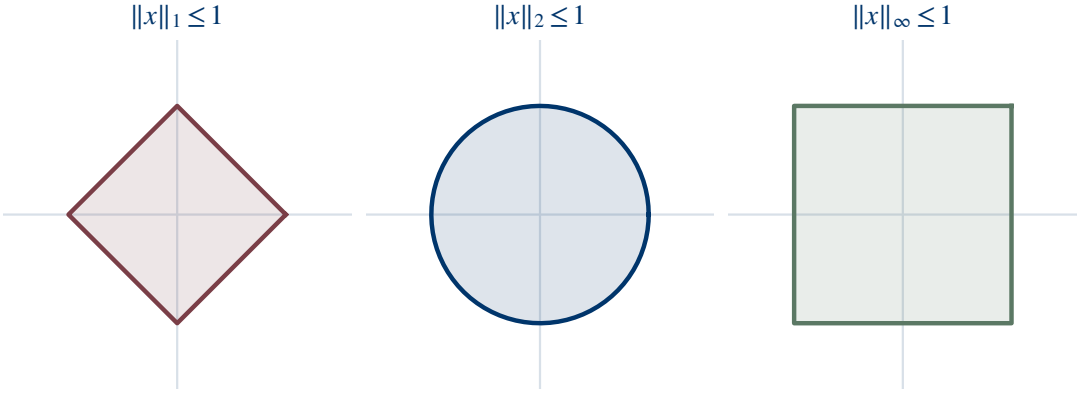
Three cases account for nearly all uses in this course:

$$\|x\|_1 = \sum_j |x_j|, \qquad \|x\|_2 = \sqrt{\sum_j x_j^2}, \qquad \|x\|_\infty = \lim_{p\to\infty} \|x\|_p = \max_j |x_j|.$$

Each gives a distance $\|x - z\|_p$, and they disagree about which vectors count as big.

For $x = (3, -4)$: $\|x\|_1 = 7$, $\|x\|_2 = 5$, $\|x\|_\infty = 4$.

Unit Balls of the ℓ_1, ℓ_2, and ℓ_∞ Norms



Each region is $\{x : \|x\| \le 1\}$. The ℓ_1 ball has corners on the axes, which is why an ℓ_1 penalty produces exact zeros; the ℓ_2 ball is round and has no preferred direction; the ℓ_∞ ball limits each coordinate separately.

Vector Norms in NumPy

```
x = np.array([3.0, -4.0])
np.linalg.norm(x, 1)         # 7.0
np.linalg.norm(x)            # 5.0   (default: 2-norm)
np.linalg.norm(x, np.inf)    # 4.0
np.linalg.norm(x - z)        # distance between two vectors
```

Norms are not scale-free: standardize the columns before treating a distance as meaningful.

Hyperplanes: Level Sets of a Linear Score

Fix $w \neq 0$ and $b \in \mathbb{R}$. The set

$$H = \{x \in \mathbb{R}^d : \; w^\top x = b\}$$

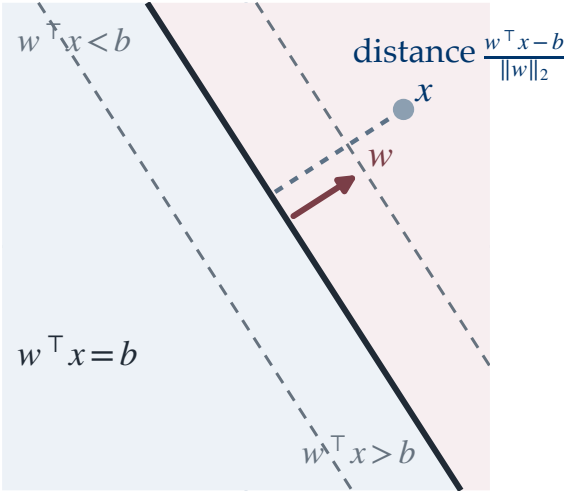
is a **hyperplane**: a line in $\mathbb{R}^2$, a plane in $\mathbb{R}^3$, a flat $(d-1)$-dimensional slice in general.

Two facts make it useful:

- w is orthogonal to H, so it names the direction the score increases;
- The signed distance from x to H is $(w^\top x - b)/\|w\|_2$.

Every linear classifier in this course predicts by asking which side of a hyperplane a customer falls on.

Visualizing a Hyperplane and Its Normal Vector



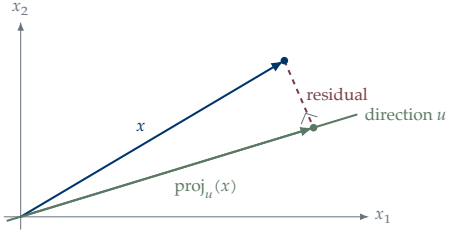
Dashed lines are other level sets of the same score.

Projection Onto a Direction

For a unit vector u, the part of x lying along u is

$$\text{proj}_u(x) = (u^\top x)\, u = uu^\top x,$$

and the rest, $x - uu^\top x$, is orthogonal to u. This decomposition into a component along u and a component orthogonal to it underlies least squares, PCA, and every residual computed in this course.



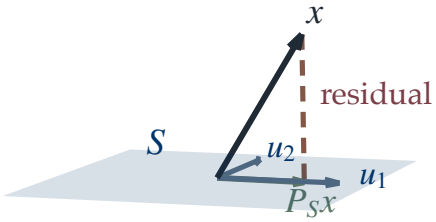
Projection Onto a Subspace

Let S be a subspace with orthonormal basis $u_1, \ldots, u_k$, collected as the columns of $U \in \mathbb{R}^{d \times k}$. Projecting onto S means adding up the projections onto each direction:

$$P_S\, x = \sum_{j=1}^{k} (u_j^\top x)\, u_j = U U^\top x.$$

- $P_S x$ is the closest point of S to x;
- The residual $x - P_S x$ is orthogonal to every vector in S;
- $k = 1$ gives the previous slide, and $P_S^2 = P_S$: projecting twice changes nothing.

Visualizing Projection Onto a Plane



$S = \text{span}\{u_1, u_2\}$ inside $\mathbb{R}^3$.

Matrices: Data Table and Linear Map

The same array plays two roles, and confusing them is a common error.

As a table

$X \in \mathbb{R}^{n \times d}$: one case per row, one feature per column — 10,000 customers by 2 measurements.

As a map

$A \in \mathbb{R}^{m \times d}$ sends $x \mapsto Ax$, turning a vector in $\mathbb{R}^d$ into one in $\mathbb{R}^m$.

$Xw \in \mathbb{R}^n$ is one score per customer, $XW \in \mathbb{R}^{n \times k}$ is k of them.

Inner dimensions must agree; outer dimensions survive.

Outline

Quadratic Forms

A symmetric matrix $A = A^\top$ turns a vector into a number:

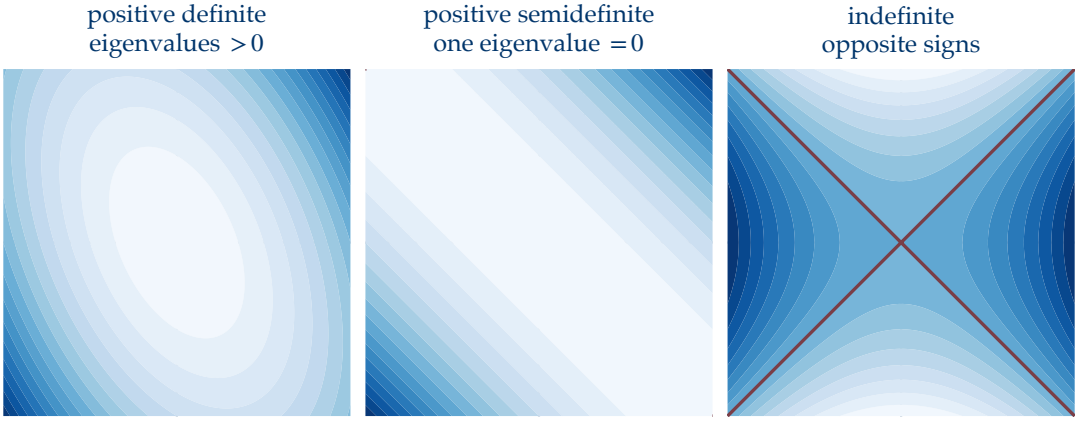
$$q(x) = x^\top A x = \sum_{i,j} A_{ij} x_i x_j.$$

This is the shape of a squared distance, a variance, and a second-order Taylor term, so it appears everywhere in the course.

- A is **positive semidefinite** (PSD) if $x^\top A x \geq 0$ for all x;
- **Positive definite** if $x^\top A x > 0$ for all $x \neq 0$.

Written $A \succeq 0$ and $A \succ 0$.

Visualizing Quadratic Forms



Level sets of $x^\top Ax$. Positive definite gives closed ellipses around one lowest point; semidefinite leaves a flat direction; indefinite gives a saddle, with directions that go up and directions that go down.

Eigenvalues and Eigenvectors

A vector $v \neq 0$ with

$$Av = \lambda v$$

is an **eigenvector**; λ is its **eigenvalue**. The matrix does not rotate v, it only rescales it.

When A is symmetric, the **spectral theorem** states more: its eigenvalues are real, and its eigenvectors may be chosen to form an orthonormal basis of $\mathbb{R}^d$. Collecting them in $V = [v_1, \dots, v_d]$ and writing $\Lambda = \operatorname{diag}(\lambda_1, \dots, \lambda_d)$,

$$A = V \Lambda V^\top = \sum_{j=1}^{d} \lambda_j v_j v_j^\top, \qquad V^\top V = I.$$

Eigenvalues and Eigenvectors

A vector $v \neq 0$ with

$$Av = \lambda v$$

is an **eigenvector**; λ is its **eigenvalue**. The matrix does not rotate v, it only rescales it.

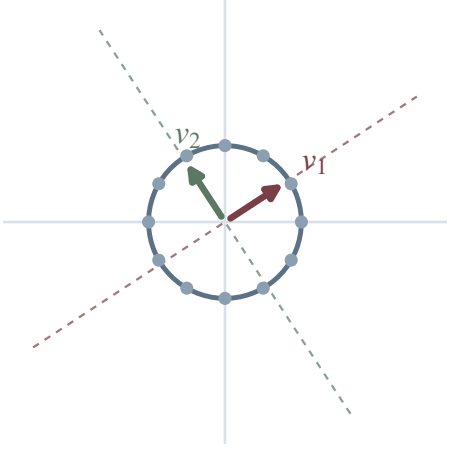
When A is symmetric, the **spectral theorem** states more: its eigenvalues are real, and its eigenvectors may be chosen to form an orthonormal basis of $\mathbb{R}^d$. Collecting them in $V = [v_1, \dots, v_d]$ and writing $\Lambda = \operatorname{diag}(\lambda_1, \dots, \lambda_d)$,

$$A = V\Lambda V^\top = \sum_{j=1}^{d} \lambda_j v_j v_j^\top, \qquad V^\top V = I.$$

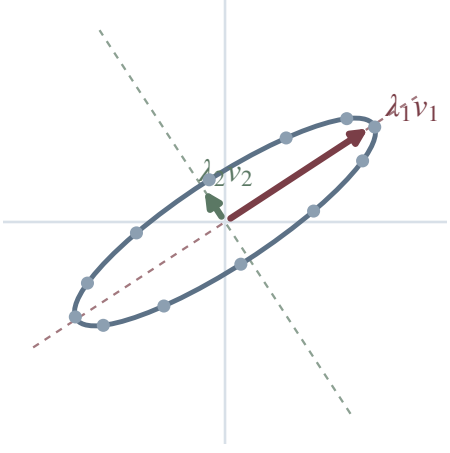
In that basis $x^\top Ax = \sum_j \lambda_j (v_j^\top x)^2$, so $A \succeq 0$ exactly when every $\lambda_j \geq 0$.

Visualizing the Action of a Symmetric Matrix

every unit vector: $\|x\|_2 = 1$



its image: $\{Ax : \|x\|_2 = 1\}$



$\lambda_1 = 2.32$ and $\lambda_2 = 0.58$.

Eigendecomposition in NumPy

```
A = np.array([[1.8, 0.8],
              [0.8, 1.1]])
values, vectors = np.linalg.eigh(A)   # eigh: symmetric matrices
values                                 # array([0.577, 2.323])
vectors.T @ vectors                    # identity: columns are orthonormal
(vectors * values) @ vectors.T         # rebuilds A
values.min() ⩾ 0                       # is A positive semidefinite?
```

Use `eigh` rather than `eig` when the matrix is symmetric: it is faster and returns real, sorted eigenvalues.

The Singular Value Decomposition

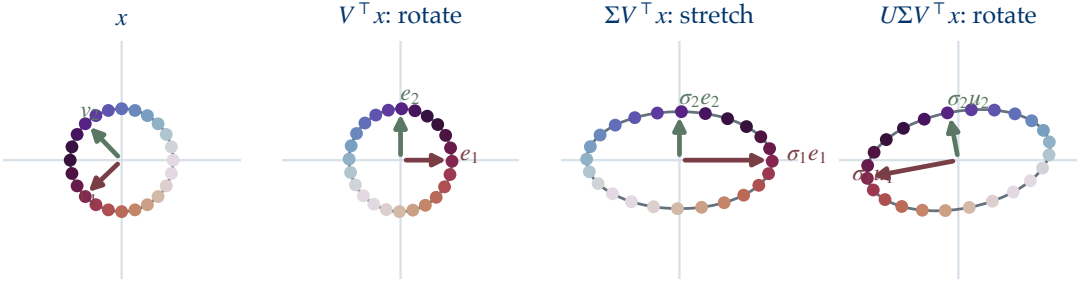
Eigenvectors need a square matrix. Any A, square or not, factors as

$$\underbrace{A}_{m\times d} = \underbrace{U}_{m\times r}\,\underbrace{\Sigma}_{r\times r}\,\underbrace{V^\top}_{r\times d}, \qquad U^\top U = V^\top V = I_r,$$

with $\Sigma = \mathrm{diag}(\sigma_1 \geq \cdots \geq \sigma_r > 0)$ and $r = \mathrm{rank}(A)$, the number of nonzero singular values.

Read right to left, Ax performs three operations: a rotation by $V^\top$, a scaling of the axes by $\sigma_1, \ldots, \sigma_r$, and a second rotation by U. Every linear map admits such a factorization.

Visualizing the SVD: Rotate, Stretch, Rotate



Points are coloured by their starting angle, so the rotations are visible. $V^\top$ turns the right singular vectors v_1, v_2 onto the axes, Σ stretches those axes by $\sigma_1 = 1.80$ and $\sigma_2 = 0.94$, and U rotates the result into place.

The SVD in NumPy and Low-Rank Approximation

```
U, s, Vt = np.linalg.svd(A, full_matrices=False)
s                                   # singular values, descending
(U * s) @ Vt                        # rebuilds A
A_r = (U[:, :r] * s[:r]) @ Vt[:r]   # best rank-r approximation
```

Retaining the largest r singular values yields the closest rank-r matrix in squared error. This result underlies PCA in Lecture 14, the linear autoencoder in Lecture 17, and low-rank compression generally.

Source: The optimality statement is the Eckart–Young theorem.

Matrix Norms: Frobenius and Operator

Matrices carry norms of their own. The two used in this course are both determined by the singular values:

$$\|A\|_F = \sqrt{\sum_{i,j} A_{ij}^2} = \sqrt{\sum_j \sigma_j^2}, \qquad \|A\|_{\text{op}} = \max_{\|x\|_2=1} \|Ax\|_2 = \sigma_1.$$

- The **Frobenius norm** treats the matrix as one long vector: its total size, used to measure approximation error;
- The **operator norm** asks how far the map can stretch any single vector: its worst-case gain.

In NumPy, `np.linalg.norm(A, 'fro')` and `np.linalg.norm(A, 2)`.

Covariance Matrices Are Positive Semidefinite

For centered data $X \in \mathbb{R}^{n \times d}$, the sample covariance is

$$S = \frac{1}{n} X^\top X \in \mathbb{R}^{d \times d}.$$

It is symmetric, and for any w,

$$w^\top S w = \frac{1}{n} \|Xw\|_2^2 \geq 0,$$

the variance of the scores Xw. So a covariance matrix is always PSD — a variance cannot be negative.

Covariance Matrices Are Positive Semidefinite

For centered data $X \in \mathbb{R}^{n \times d}$, the sample covariance is

$$S = \frac{1}{n} X^\top X \in \mathbb{R}^{d \times d}.$$

It is symmetric, and for any w,

$$w^\top S w = \frac{1}{n} \|Xw\|_2^2 \geq 0,$$

the variance of the scores Xw. So a covariance matrix is always PSD — a variance cannot be negative.

Its eigenvectors are the directions of greatest spread, which is exactly what PCA will ask for.

Outline

The Derivative as a Local Linear Approximation

For $L : \mathbb{R} \to \mathbb{R}$,

$$L(\theta + \Delta) \approx L(\theta) + L'(\theta)\,\Delta.$$

The derivative is the slope of the best straight-line approximation to L near θ, and training uses it in precisely that role: it reports how the objective function responds to a small change in the parameter.

What follows extends the same statement to several variables.

The Gradient

For $L : \mathbb{R}^d \to \mathbb{R}$, collect the partial derivatives:

$$\nabla L(\theta) = \begin{bmatrix} \partial L/\partial \theta_1 \\ \vdots \\ \partial L/\partial \theta_d \end{bmatrix} \in \mathbb{R}^d, \qquad L(\theta + \Delta) \approx L(\theta) + \nabla L(\theta)^\top \Delta.$$

The approximation is an inner product, so the vector geometry applies at once: among all steps Δ of a fixed length, the one aligned with ∇L increases the loss most, and $-\nabla L$ decreases it most.

The Gradient

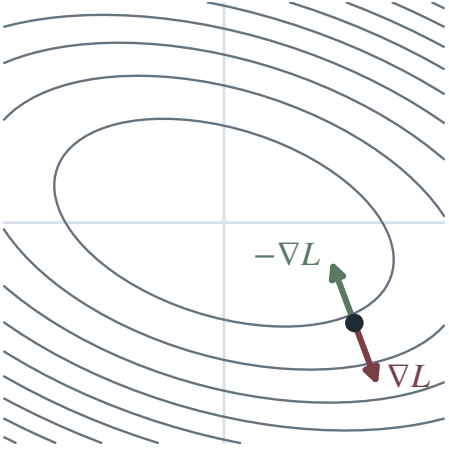
For $L : \mathbb{R}^d \to \mathbb{R}$, collect the partial derivatives:

$$\nabla L(\theta) = \begin{bmatrix} \partial L/\partial \theta_1 \\ \vdots \\ \partial L/\partial \theta_d \end{bmatrix} \in \mathbb{R}^d, \qquad L(\theta + \Delta) \approx L(\theta) + \nabla L(\theta)^\top \Delta.$$

The approximation is an inner product, so the vector geometry applies at once: among all steps Δ of a fixed length, the one aligned with ∇L increases the loss most, and $-\nabla L$ decreases it most.

This is the rule gradient-based optimization relies on.

Visualizing the Gradient and the Contours



Along a contour the loss does not change, so the direction of fastest change must be perpendicular to it. Descent follows $-\nabla L$, which crosses the contours at right angles.

The Hessian and the Second-Order Approximation

The gradient says which way is downhill. It says nothing about how far that stays true. The second derivatives do:

$$\nabla^2 L(\theta) = \begin{bmatrix} \partial^2 L/\partial\theta_1^2 & \cdots & \partial^2 L/\partial\theta_1\partial\theta_d \\ \vdots & & \vdots \\ \partial^2 L/\partial\theta_d\partial\theta_1 & \cdots & \partial^2 L/\partial\theta_d^2 \end{bmatrix} \in \mathbb{R}^{d\times d},$$

a symmetric matrix, giving the second-order approximation

$$L(\theta + \Delta) \approx L(\theta) + \nabla L(\theta)^\top \Delta + \tfrac{1}{2}\Delta^\top \nabla^2 L(\theta)\, \Delta.$$

Eigenvalues of the Hessian

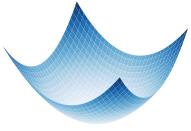
The Hessian is symmetric, all of its eigenvalues are real numbers:

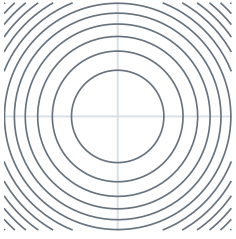
- All eigenvalues > 0: a bowl, and a stationary point is a minimum;
- Mixed signs: a saddle, downhill in some directions;
- Eigenvalues of very different size: a long narrow valley, where a step good for one direction is bad for another.

If $\nabla^2 L(\theta) \succeq 0$ at every θ, the loss is **convex** and has no local minimum that is not global. That is a property of the whole surface, not something one Hessian can certify.

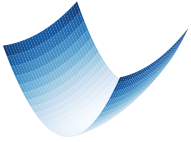
Visualizing Curvature: Bowl, Valley, Saddle

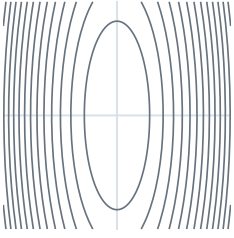
eigenvalues 1, 1: bowl



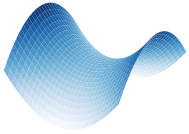


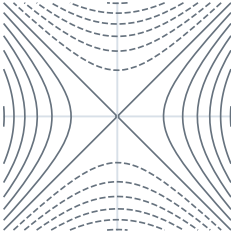
eigenvalues 1, .12: valley





eigenvalues 1, -1: saddle





The surface $\frac{1}{2}x^\top Hx$ above, its contours below.

Finite-Difference Gradient Check

```
loss = lambda th, x=2.0, y=5.0: 0.5 * (y - th * x) ** 2
eps  = 1e-5
numeric = (loss(1.0 + eps) - loss(1.0 - eps)) / (2 * eps)
numeric, (1.0 * 2.0 - 5.0) * 2.0     # (-6.000000, -6.0)
```

A centered difference is the standard check on a hand-derived gradient. **Too large an ε and truncation dominates; too small and floating-point cancellation does.**

Outline

Why Not Just Differentiate by Hand?

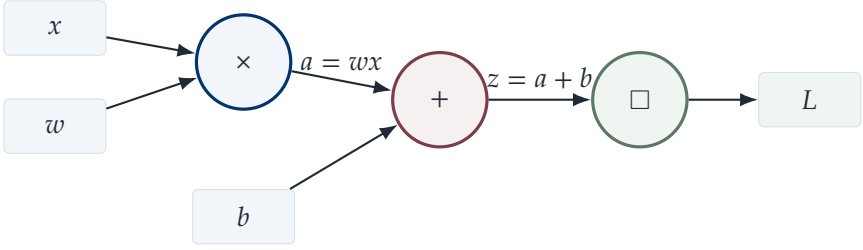
A modern model composes thousands of simple operations. Two familiar options both fail:

- By hand: correct, but impractical at scale, and rederived whenever the architecture changes;
- Finite differences: needs one extra forward pass per parameter — millions of them, each only approximate.

Automatic differentiation gives exact derivatives of the composed function at a cost close to one extra forward pass in total, whatever the number of parameters.

The Computation Graph

Inputs and parameters at the leaves, one elementary operation per node, the loss at the root.



Forward pass: evaluate leaves to root, keeping each intermediate value. Here $a = wx$, then $z = a + b$, then $L = z^2$.

Reverse-Mode Differentiation

Each node knows only how its own output responds to its own inputs. The chain rule multiplies those local factors along the path:

$$\frac{\partial L}{\partial w} = \frac{\partial L}{\underbrace{\partial z}_{2z}} \underbrace{\frac{\partial z}{\partial a}}_{1} \underbrace{\frac{\partial a}{\partial w}}_{x}.$$

Reverse-Mode Differentiation

Each node knows only how its own output responds to its own inputs. The chain rule multiplies those local factors along the path:

$$\frac{\partial L}{\partial w} = \frac{\partial L}{\underbrace{\partial z}_{2z}} \frac{\partial z}{\underbrace{\partial a}_{1}} \frac{\partial a}{\underbrace{\partial w}_{x}}.$$

Reverse mode walks root to leaves, carrying the accumulated $\partial L/\partial(\cdot)$ and multiplying by one local derivative at each step. One sweep reaches every parameter.

Worked Example: Forward and Backward Values

Take $x = 2$, $w = 3$, $b = -1$.

$$\text{forward:} \quad a = wx = 6, \qquad z = a + b = 5, \qquad L = z^2 = 25;$$

$$\text{backward:} \quad \frac{\partial L}{\partial z} = 2z = 10, \quad \frac{\partial L}{\partial a} = 10 \cdot 1 = 10, \quad \frac{\partial L}{\partial w} = 10 \cdot x = 20.$$

Worked Example: Forward and Backward Values

Take $x = 2$, $w = 3$, $b = -1$.

$$\text{forward:} \quad a = wx = 6, \qquad z = a + b = 5, \qquad L = z^2 = 25;$$

$$\text{backward:} \quad \frac{\partial L}{\partial z} = 2z = 10, \quad \frac{\partial L}{\partial a} = 10 \cdot 1 = 10, \quad \frac{\partial L}{\partial w} = 10 \cdot x = 20.$$

Also $\partial L/\partial b = 10$ and $\partial L/\partial x = 10 \cdot w = 30$: the same sweep produced all three derivatives, having stored a, z on the way in.

Automatic Differentiation in PyTorch

```
import torch
x = torch.tensor(2.0)
w = torch.tensor(3.0, requires_grad=True)   # track this one
b = torch.tensor(-1.0, requires_grad=True)

a = w * x        # the graph is built as the code runs
z = a + b
loss = z ** 2
loss.backward()  # one reverse sweep
w.grad, b.grad   # tensor(20.), tensor(10.)
```

`requires_grad=True` marks the leaves to differentiate; `backward()` fills their `.grad`.

Three Rules for Using Autograd

```
w.grad.zero_()              # .grad accumulates; clear it each step
with torch.no_grad():       # evaluation: build no graph, save memory
    predictions = model(x_test)
value = loss.detach()       # take the number, cut the graph
```

- Gradients add into `.grad`, they do not overwrite;
- The graph is rebuilt on every forward pass, so ordinary Python control flow works;
- `backward()` computes derivatives and does not change any parameter.

Checking Autograd Against a Hand Derivation

```
theta = torch.tensor(1.0, requires_grad=True)
x, y = torch.tensor(2.0), torch.tensor(5.0)

loss = 0.5 * (y - theta * x) ** 2
loss.backward()

theta.grad                      # tensor(-6.)
(theta * x - y) * x             # tensor(-6., grad_fn=...)
```

Autodiff is exact: it applies the chain rule to the operations that actually ran. Disagreement with a hand derivation means one of the two is wrong.

Outline

Joint and Marginal Distributions

Two measurements on the same case are described together by a **joint** distribution $p(x, y)$: how often each pair of values occurs.

Ignoring one of them recovers a **marginal**,

$$p(x) = \int p(x, y)\, dy, \qquad p(y) = \int p(x, y)\, dx,$$

(sums, for discrete variables). Marginalizing discards information: two very different joints can share the same pair of marginals.

Conditional Distributions

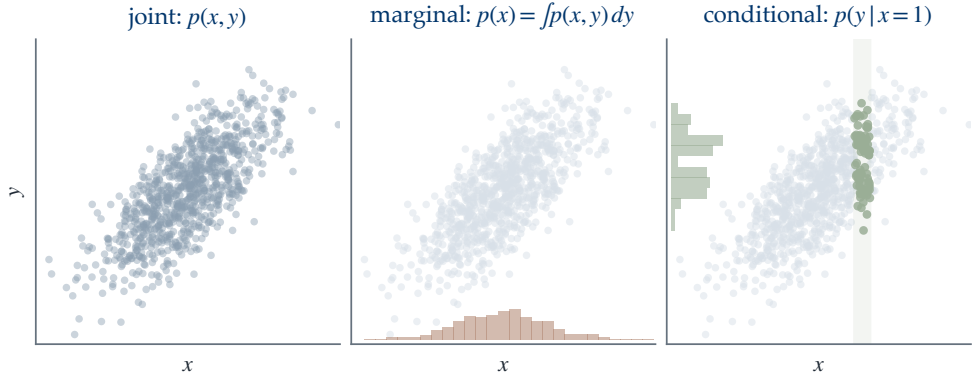
Fixing one measurement and asking about the other gives a **conditional** distribution,

$$p(y \mid x) = \frac{p(x, y)}{p(x)}, \qquad p(x) > 0.$$

Two moves, and the picture on the next slide shows both: select the cases with that value of x, then renormalize so what remains is a distribution in its own right.

This is the object every supervised model estimates: $p(y \mid x)$ is exactly "what should I predict, given these features?"

Visualizing Joint, Marginal, and Conditional



The same 900 pairs each time. Joint: the cloud itself. Marginal: collapse it onto one axis and the other coordinate is forgotten. Conditional: keep only the narrow strip at $x = 1$ and look at how y is distributed inside it.

Original course simulation, seed 26502.

Bayes' Rule

Writing the joint two ways, $p(x, y) = p(y \mid x)\, p(x) = p(x \mid y)\, p(y)$, and rearranging gives Bayes' rule — one named factor at a time:

$$\underbrace{p(y \mid x)}_{\text{posterior}} = \frac{\overbrace{p(x \mid y)}^{\text{likelihood}}\ \overbrace{p(y)}^{\text{prior}}}{\underbrace{p(x)}_{\text{evidence}}}.$$

The **prior** is what we believed about y before seeing anything. The **likelihood** says how probable the observed x would be under each value of y. The **evidence** $p(x) = \int p(x \mid y) p(y)\, dy$ normalizes. The **posterior** is the belief after the data.

Expectation and Variance

Centre

$$\mathbb{E}[X] = \int x\, p(x)\, dx$$

the balance point of the distribution

Spread

$$\mathrm{Var}(X) = \mathbb{E}[(X - \mathbb{E}X)^2]$$

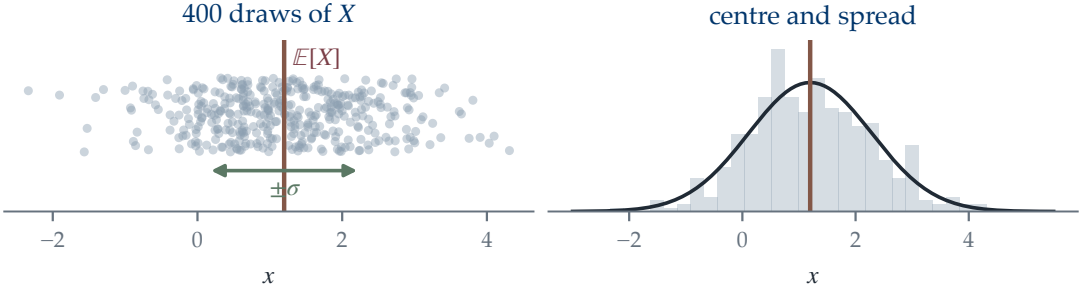
the mean squared distance from it

In $\mathbb{R}^d$ the spread becomes a matrix, the **covariance**

$$\mathrm{Cov}(X) = \mathbb{E}[(X - \mathbb{E}X)(X - \mathbb{E}X)^\top] \in \mathbb{R}^{d \times d},$$

symmetric and positive semidefinite, so the eigenvalue results apply.

Visualizing Expectation and Variance



400 draws with $\mathbb{E}[X] = 1.2$ and $\sigma = 1.1$. The mean is where the cloud balances; the standard deviation is a typical distance from it, not a bound — points fall outside $\pm\sigma$ routinely.

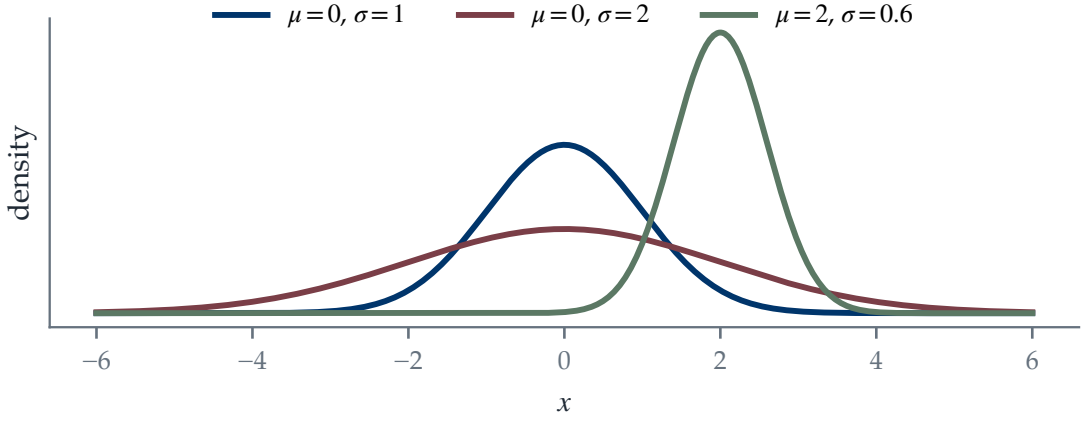
Original course simulation, seed 26505.

The Gaussian in One Dimension

$$p(x) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{(x-\mu)^2}{2\sigma^2}\right), \qquad x \sim \mathcal{N}(\mu, \sigma^2).$$

Two parameters, and each does one job: μ moves the density, σ sets its width. The exponent is a squared distance from the centre, in units of σ — which is why a Gaussian log likelihood is a squared error.

Visualizing the Gaussian Density



The Multivariate Gaussian

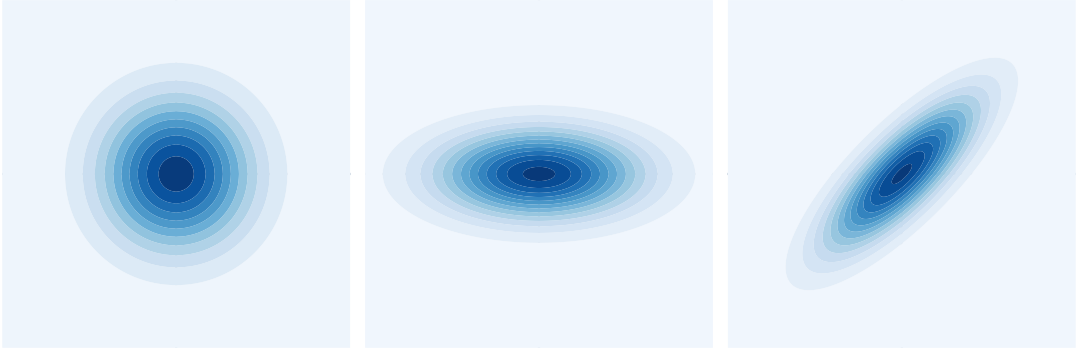
For $x \in \mathbb{R}^d$, mean $\mu \in \mathbb{R}^d$, covariance $\Sigma \succ 0$:

$$p(x) = \frac{1}{(2\pi)^{d/2}|\Sigma|^{1/2}} \exp\Big(-\tfrac{1}{2}(x-\mu)^\top \Sigma^{-1}(x-\mu)\Big).$$

The exponent is a quadratic form, so everything said about them applies: the level sets $\{x : (x-\mu)^\top \Sigma^{-1}(x-\mu) = c\}$ are ellipsoids, centred at μ, with axes along the eigenvectors of Σ and half-lengths proportional to $\sqrt{\lambda_j}$.

Visualizing Multivariate Gaussian Contours

$\Sigma = I$ $\Sigma = \text{diag}(1.8, .35)$ $\Sigma = \begin{pmatrix} 1 & .8 \\ .8 & 1 \end{pmatrix}$



$\Sigma = I$ gives circles. A diagonal Σ stretches the axes independently. An off-diagonal entry tilts the ellipse: at right both variances are 1 and $\Sigma_{12} = 0.8$, so knowing one coordinate tells you something about the other.

A Linear Map of a Gaussian Is Gaussian

If $z \sim \mathcal{N}(0, I_d)$ and $x = Az + \mu$ for $A \in \mathbb{R}^{d \times d}$, then

$$\mathbb{E}[x] = A\,\mathbb{E}[z] + \mu = \mu, \qquad \mathrm{Cov}(x) = A\,\mathrm{Cov}(z)A^\top = AA^\top,$$

A Linear Map of a Gaussian Is Gaussian

If $z \sim \mathcal{N}(0, I_d)$ and $x = Az + \mu$ for $A \in \mathbb{R}^{d \times d}$, then

$$\mathbb{E}[x] = A\,\mathbb{E}[z] + \mu = \mu, \qquad \mathrm{Cov}(x) = A\,\mathrm{Cov}(z)A^\top = AA^\top,$$

and the distribution stays in the family:

$$\boxed{x \sim \mathcal{N}(\mu, AA^\top)}$$

A Linear Map of a Gaussian Is Gaussian

If $z \sim \mathcal{N}(0, I_d)$ and $x = Az + \mu$ for $A \in \mathbb{R}^{d \times d}$, then

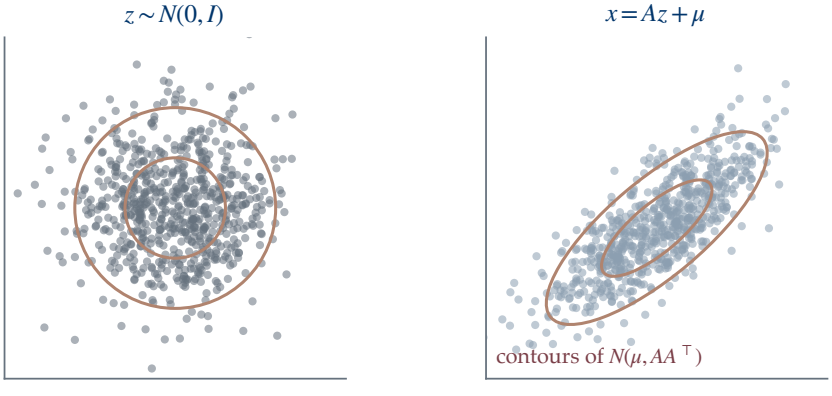
$$\mathbb{E}[x] = A\,\mathbb{E}[z] + \mu = \mu, \qquad \mathrm{Cov}(x) = A\,\mathrm{Cov}(z)A^\top = AA^\top,$$

and the distribution stays in the family:

$$\boxed{x \sim \mathcal{N}(\mu, AA^\top)}$$

This is the property that makes Gaussians so convenient. Sums, marginals, and conditionals of jointly Gaussian variables are all Gaussian — so a linear model with Gaussian noise can be pushed through the algebra and stay solvable.

Sampling a Gaussian by Transforming Standard Normals



700 draws from $\mathcal{N}(0, I)$, each mapped by $x = Az + \mu$. The red ellipses are the one- and two-standard-deviation contours of $\mathcal{N}(\mu, AA^\top)$, computed from A rather than from the sample: the empirical covariance $\left(\begin{smallmatrix} 1.18 & .81 \\ .81 & .90 \end{smallmatrix}\right)$ matches $AA^\top = \left(\begin{smallmatrix} 1.21 & .83 \\ .83 & .92 \end{smallmatrix}\right)$.

Original course simulation, seed 26502.

Gaussians in NumPy and PyTorch

```
mu = np.array([1.2, -0.6])
A  = np.array([[1.1, 0.0], [0.75, 0.6]])
z = np.random.default_rng(0).normal(size=(700, 2))
x = z @ A.T + mu              # x ~ N(mu, A A^T)
np.cov(x.T)                   # close to A @ A.T

import torch
normal = torch.distributions.MultivariateNormal(
    torch.tensor(mu), covariance_matrix=torch.tensor(A @ A.T))
normal.sample((700,))         # the same distribution, drawn directly
```

Drawing standard normals and applying A is exactly how a VAE samples its codes in Lecture 17.

Summary: Linear Algebra

$$w^\top x = \|w\|_2 \|x\|_2 \cos\theta, \qquad A = V\Lambda V^\top, \qquad A = U\Sigma V^\top.$$

1. A score is an inner product; a decision boundary is a hyperplane; a residual is a projection.
2. Symmetric PSD matrices are the shape of variances and of curvature, and their eigenvalues say which directions matter.
3. Every linear map rotates, stretches, and rotates; truncating the stretch gives the best low-rank approximation.

Summary: Derivatives and Probability

$$L(\theta + \Delta) \approx L(\theta) + \nabla L(\theta)^\top \Delta + \tfrac{1}{2}\Delta^\top \nabla^2 L(\theta)\Delta, \qquad x = Az + \mu \sim \mathcal{N}(\mu, AA^\top).$$

1. The gradient points along the steepest increase; the Hessian says how far that remains true.
2. Reverse-mode autodiff gets every parameter's derivative in one sweep, exactly, and PyTorch builds the graph as the code runs.
3. Gaussian densities are quadratic forms in the exponent, and linear maps keep them Gaussian.

Next Lecture

These objects describe shapes, scores, curvature, and uncertainty. They do not yet answer a question about data.

Lecture 3 — Linear Regression puts them to work: a design matrix, a squared-error loss, a normal-equation solution, and the first honest assessment of what the fit does and does not establish.

Reading for this lecture: DL Chapters 2–4 (linear algebra, probability, numerical computation); D2L Chapter 2.