Derivatives, Gradients, Jacobians & Hessians

The calculus behind optimization and backpropagation

Prof. Nipun Batra

ES 667 · Deep Learning · IIT Gandhinagar

Why a calculus lecture?

Every step of training is a gradient step:

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

So we must be fluent with

$$\frac{\mathrm{d}}{\mathrm{d}x}, \quad \frac{\partial}{\partial x_i}, \quad \nabla f, \quad J, \quad H.$$

These five objects are the entire vocabulary of optimization and backprop.

Outline

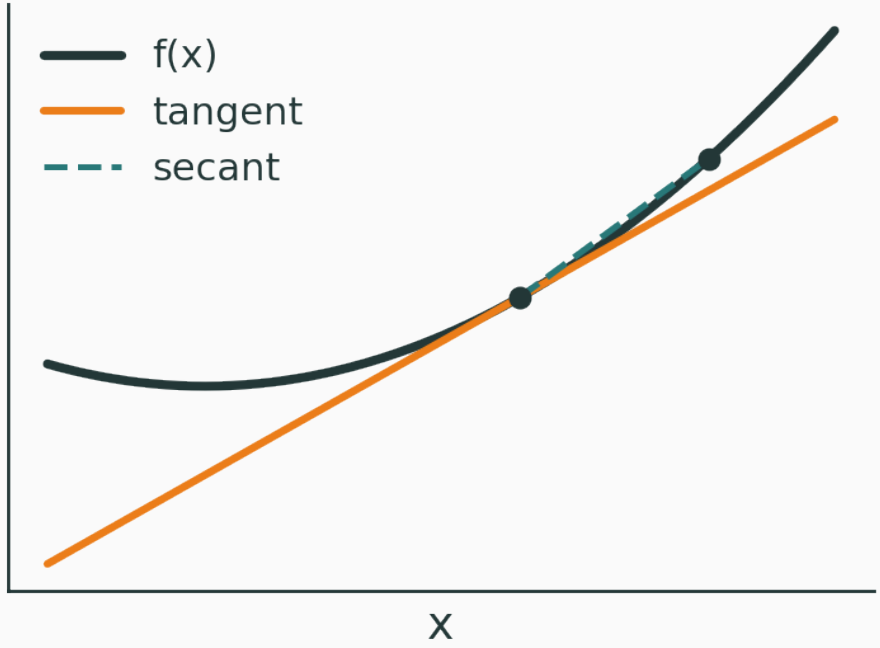
1. **Derivative** — the local linear model of a scalar function
2. **Gradient** — combine partials into a direction for optimization
3. **Hessian** — describe curvature and explain difficult descent paths
4. **Jacobian** — linearize a vector-valued transformation
5. **Chain rule and VJPs** — compose local derivatives into backpropagation

The scalar derivative

Derivative as slope

V VISUAL

$$f'(x) = \frac{\mathrm{d}f}{\mathrm{d}x} = \lim_{\varepsilon \to 0} \frac{f(x+\varepsilon) - f(x)}{\varepsilon}$$



the secant slope → the tangent slope as $\varepsilon \to 0$

The derivative is a local linear model

Near a point x:

$$f(x + \Delta x) \approx f(x) + f'(x)\,\Delta x$$

derivative = best local linear approximation

Example: $f(x) = x^2$ at $x = 3$, so $f(3) = 9$ and $f'(3) = 6$:

$$f(3 + \Delta x) \approx 9 + 6\,\Delta x.$$

Local linear model: numeric check (cont.)

D DERIVATION

Test the line $f(3+\Delta x) \approx 9 + 6\,\Delta x$ at a **small** step $\Delta x = 0.1$: Linear prediction: $9 + 6 \cdot 0.1 = 9.6$. True value: $f(3.1) = 3.1^2 = 9.61$. Error: $9.61 - 9.6 = 0.01$ — tiny.

Now a **large** step $\Delta x = 1$: the line gives $9 + 6 = 15$, but $f(4) = 16$ — error 1. The linear model degrades as Δx grows.

Interactive: tangent line

I INTERACTIVE

INTERACTIVE

↗ nipunbatra.github.io/interactive-articles/derivative-tangent

drag the point and Δx; watch the tangent, the secant, and the approximation error. Functions: x^2, $\sin x$, e^x, $\log(1 + e^x)$.

Explore how the approximation error grows as the step leaves the local tangent regime.

Checkpoint: local linearization

Q QUESTION

When is a first-order linear approximation most reliable?

A. For a large step on a sharply curved function

B. For a small step where curvature is modest

C. Only at a global minimum

D. Whenever the gradient is zero

Answer: local linearization

A ANSWER

Correct: B — For a small step where curvature is modest

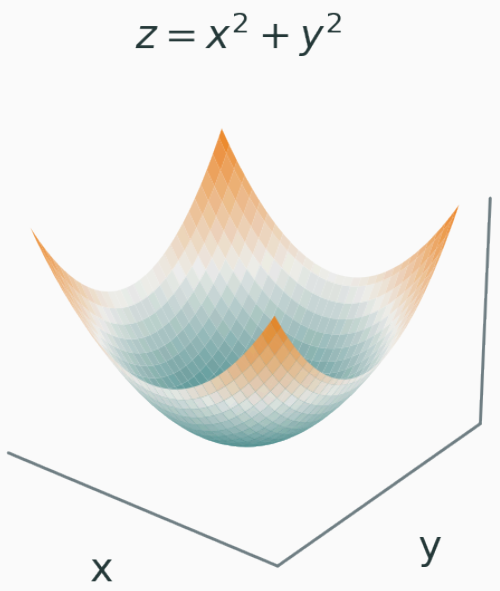
Why: The derivative captures local slope; omitted higher-order terms become important over large steps or sharp bends.

Partial derivatives and surfaces

From curves to surfaces

V VISUAL

A function of two inputs is a **surface**: $f(x, y) = x^2 + y^2$.



input (x, y) → height $z = f(x, y)$

Partial derivatives

$\frac{\partial f}{\partial x}$ — change x, **hold y fixed**. $\frac{\partial f}{\partial y}$ — change y, hold x fixed. For $f(x, y) = x^2 + y^2$:

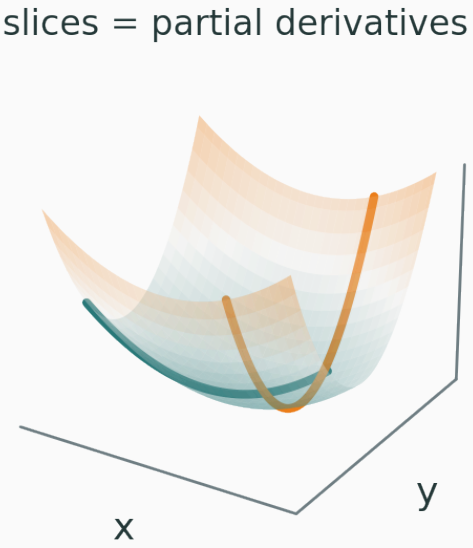
$$\frac{\partial f}{\partial x} = 2x, \qquad \frac{\partial f}{\partial y} = 2y$$

At $(x, y) = (3, 1)$: $\frac{\partial f}{\partial x} = 6$, $\frac{\partial f}{\partial y} = 2$.

A partial is the slope of a slice

V VISUAL

Fix one input and you are left with a 1-D curve — the partial is **its** slope.



For $f = x^2 + 3y^2$ at $(1, 2)$: $\frac{\partial f}{\partial x} = 2$, $\frac{\partial f}{\partial y} = 12$ — the function climbs **faster** along y.

Interactive: surface slices

I INTERACTIVE

INTERACTIVE

↗ nipunbatra.github.io/interactive-articles/surface-partials

move a point on a 3-D surface and read off the two slice slopes. Functions: $x^2 + y^2$, $x^2 + 3y^2$, $\sin x \cos y$.

Use the two slice slopes to compare the axes at a chosen point.

The gradient

Gradient: stack the partials

$$\nabla f(x) = \begin{pmatrix} \frac{\partial f}{\partial x_1} \\ \frac{\partial f}{\partial x_2} \\ \vdots \\ \frac{\partial f}{\partial x_d} \end{pmatrix}$$

For $f(x, y) = x^2 + y^2$: $\nabla f = \begin{pmatrix} 2x \\ 2y \end{pmatrix}$. At $(3, 1)$: $\nabla f = \begin{pmatrix} 6 \\ 2 \end{pmatrix}$ — pointing **outward**, uphill from the bowl's centre.

Gradient = local linear model

D DERIVATION

Same first-order Taylor idea, now with a **dot product**:

$$f(x + \Delta x) \approx f(x) + \nabla f(x)^\top \Delta x$$

the vector version of $f(x + \Delta x) \approx f(x) + f'(x)\Delta x$

Directional derivative

For a **unit** direction u:

$$D_u f(x) = \nabla f(x)^\top u$$

The rate of change along u is just the **projection** of the gradient onto u.

Example: $f = x^2 + y^2$ at $(1, 2)$, so $\nabla f = \begin{pmatrix} 2 \\ 4 \end{pmatrix}$. Take $u = (3, 4)/5 = (0.6, 0.8)$:

$$D_u f = 2 \cdot 0.6 + 4 \cdot 0.8 = 1.2 + 3.2 = 4.4$$

compare $\|\nabla f\| = \sqrt{20} \approx 4.47$ – the **gradient** direction is a touch steeper.

The gradient points uphill

D DERIVATION

By Cauchy–Schwarz, over all unit u:

$$\nabla f(x)^\top u \le \|\nabla f(x)\|$$

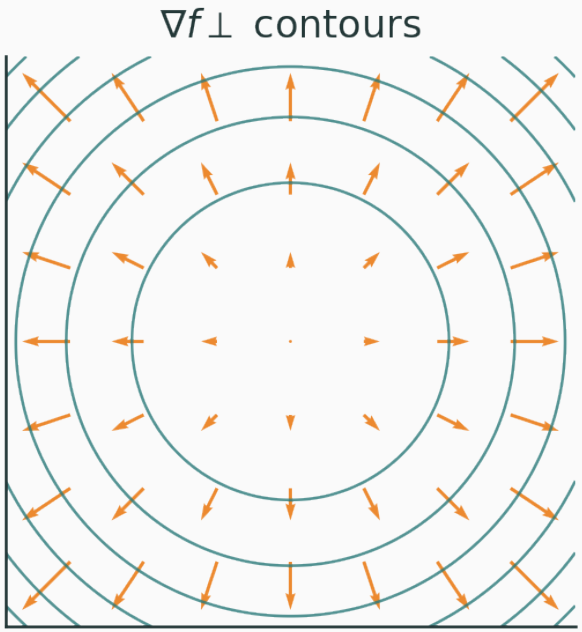
with equality at $u = \nabla f(x)/\|\nabla f(x)\|$.

∇f = steepest ascent · $-\nabla f$ = steepest descent

The gradient is perpendicular to contours

V VISUAL

A contour is the set $f(x, y) = c$. Along it, f does not change, so for any tangent v: $\nabla f^\top v = 0$.



so ∇f is **orthogonal** to the contour — it points straight across the level lines.

Gradient descent on the contour map

$$x_{t+1} = x_t - \eta \nabla f(x_t)$$

Example: at $x_t = (3, 1)$ with $\nabla f = (6, 2)$ and $\eta = 0.1$:

$$x_{t+1} = (3, 1) - 0.1(6, 2) = (2.4, 0.8)$$

Each step moves **perpendicular** to the current contour, downhill. The learning rate η sets the step length.

Interactive: gradient field & descent

I INTERACTIVE

INTERACTIVE

↗ nipunbatra.github.io/interactive-articles/optimizer-race

Watch gradient arrows on a contour map and a descent trajectory as you change η and the start point. Surfaces: $x^2 + y^2$, $x^2 + 10y^2$, $\sin x + \cos y$.

Move the start point to the bottom of the surface and inspect the update.

Checkpoint: a stationary point

Q QUESTION

At a differentiable local minimum, what does a plain gradient-descent step do?

A. Moves uphill

B. Leaves the point unchanged because $\nabla f = 0$

C. Doubles the learning rate

D. Must move along a contour

Answer: a stationary point

A ANSWER

Correct: B — Leaves the point unchanged

Why: At a differentiable local minimum the gradient is zero, so $x_{\{t+1\}} = x_t - \eta \nabla f(x_t) = x_t$.

The Hessian and curvature

Second derivative in 1-D

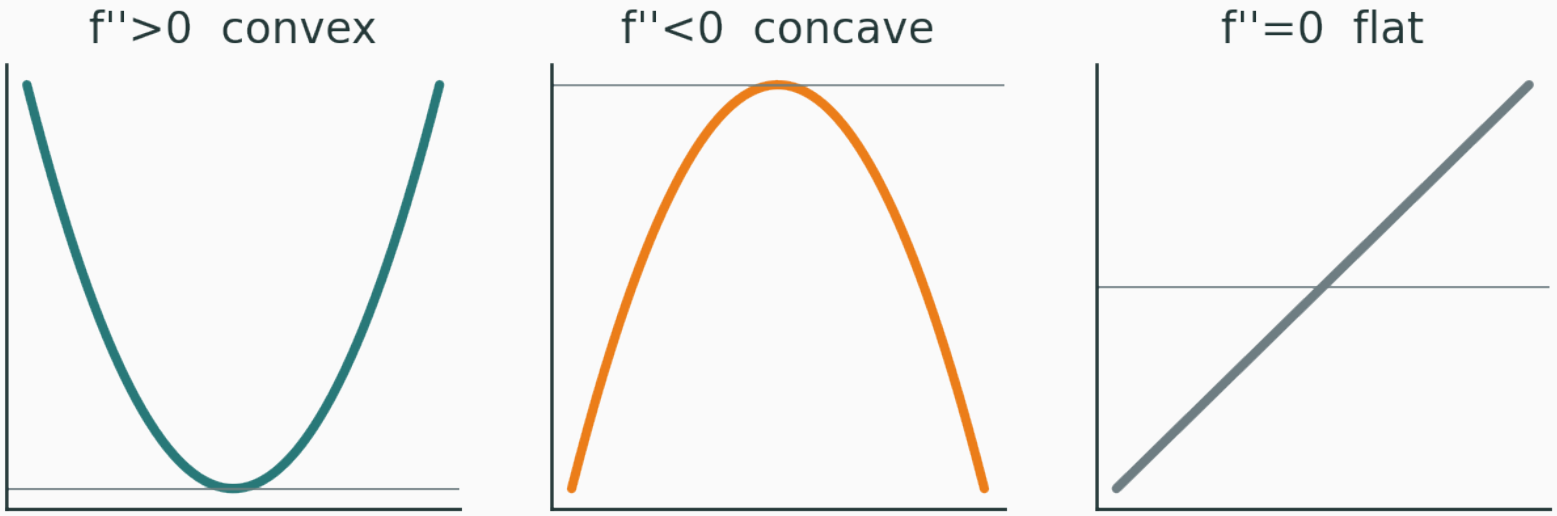
$$f''(x) = \frac{\mathrm{d}^2 f}{\mathrm{d}x^2} \quad = \quad \text{how fast the slope changes}$$

$f(x) = x^2 \Rightarrow f'' = 2$ (bowl, curves **up**); $f(x) = -x^2 \Rightarrow f'' = -2$ (hill, curves **down**).

Convex, concave, flat

V VISUAL

The **sign** of f'' names the shape:



$f'' > 0$ convex (bowl) · $f'' < 0$ concave (hill) · $f'' = 0$ flat (inflection)

The Hessian: matrix of second partials

$$H = \nabla^2 f, \quad H_{ij} = \frac{\partial^2 f}{\partial x_i \partial x_j}$$

For two variables:

$$H = \begin{pmatrix} \frac{\partial^2 f}{\partial x^2} & \frac{\partial^2 f}{\partial x \partial y} \\ \frac{\partial^2 f}{\partial y \partial x} & \frac{\partial^2 f}{\partial y^2} \end{pmatrix}$$

Hessian example

D DERIVATION

$$f(x,y) = x^2 + 3y^2 \quad \Longrightarrow \quad \nabla f = \begin{pmatrix} 2x \\ 6y \end{pmatrix}$$

Differentiate each partial again, one entry at a time:

$$\partial_{xx} = \frac{\partial}{\partial x}(2x) = 2$$

$$\partial_{yy} = \frac{\partial}{\partial y}(6y) = 6, \qquad \partial_{xy} = \frac{\partial}{\partial y}(2x) = 0$$

$$H = \begin{pmatrix} 2 & 0 \\ 0 & 6 \end{pmatrix}$$

Curvature is **larger** in the y direction — the bowl is steeper that way.

Second-order local approximation

D DERIVATION

The full second-order Taylor expansion:

$$f(x+\Delta x) \approx f(x) + \underbrace{\nabla f(x)^\top \Delta x}_{\text{tilt (gradient)}} + \underbrace{\frac{1}{2}\Delta x^\top H(x)\Delta x}_{\text{curvature (Hessian)}}$$

gradient tilts · Hessian bends

Hessian geometry

For a quadratic $f(x) = \frac{1}{2} x^\top A x$: $\nabla f = Ax$, $H = A$.

- **eigenvectors** of H → the curvature **directions**;
- **eigenvalues** of H → the curvature **magnitudes**.

Classifying critical points

At a point where $\nabla f = 0$, the Hessian eigenvalues decide the shape:

Hessian eigenvalues	**Local shape**
all positive	local minimum
all negative	local maximum
mixed signs	saddle point
some zero	inconclusive / flat

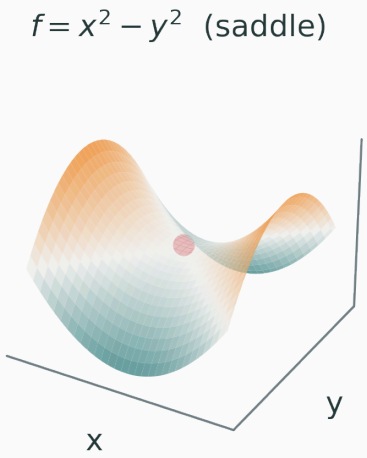
the eigenvalue **signs** alone classify the point.

A saddle point

V VISUAL

$$f(x,y) = x^2 - y^2, \quad \nabla f = \begin{pmatrix} 2x \\ -2y \end{pmatrix}, \quad H = \begin{pmatrix} 2 & 0 \\ 0 & -2 \end{pmatrix}$$

Eigenvalues $+2$ and -2 have **mixed signs** → a saddle.

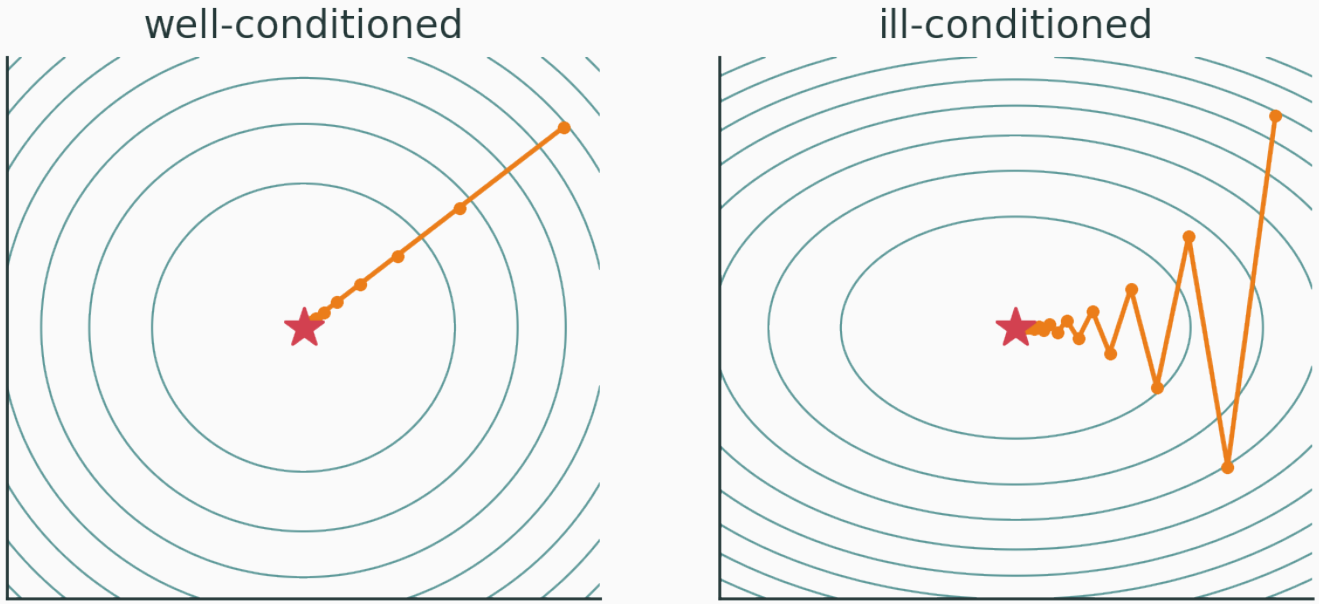


$\nabla f = 0$ at the origin, but it is **not** a minimum

Why curvature matters for optimization

V VISUAL

The gradient picks a **direction**, but curvature decides a good **step size**.



Elongated contours (a large **condition number**) make first-order gradient descent **zigzag** — much curvature one way, little the other.

Interactive: curvature & descent

I INTERACTIVE

INTERACTIVE

↗ nipunbatra.github.io/interactive-articles/optimizer-race

Reshape a quadratic $ax^2 + by^2 + cxy$, see its Hessian eigenvectors, and watch the descent path zigzag.

Observe how one shared step size alternately overshoots the steep direction and creeps along the flat one.

Follow-along: exact quadratic GD Colab — predict both trajectories before running.

The Jacobian

When the gradient is not enough

The gradient is for **scalar** outputs, $f : \mathbb{R}^d \to \mathbb{R}$. But network layers map **vectors to vectors**, $f : \mathbb{R}^n \to \mathbb{R}^m$ — e.g. $z = Wx + b$. We need the **Jacobian**.

Jacobian: matrix of all partials

For $f : \mathbb{R}^n \to \mathbb{R}^m$:

$$J = \frac{\partial f}{\partial x} \in \mathbb{R}^{m \times n}, \qquad J_{ij} = \frac{\partial f_i}{\partial x_j}$$

$f(x + \Delta x) \approx f(x) + J(x)\,\Delta x$ **— a local linear map**

Two Jacobians you already know

D DERIVATION

Two building blocks whose Jacobians you can write down instantly:

Linear layer $f(x) = Wx + b$:

$J = W$

since $\frac{\partial z_i}{\partial x_j} = W_{ij}$.

Elementwise $h = \varphi(z)$:

$J = \mathrm{diag}(\varphi'(z_1), \ldots, \varphi'(z_m))$

(for ReLU, entries are 0 or 1).

Worked example: a $2 \to 2$ Jacobian

D DERIVATION

$f(x_1, x_2) = \begin{pmatrix} x_1^2 + x_2 \\ \sin(x_1 x_2) \end{pmatrix}$ — the map from the PyTorch demo later. Row 1, partials of $f_1 = x_1^2 + x_2$: $\frac{\partial f_1}{\partial x_1} = 2x_1$, $\frac{\partial f_1}{\partial x_2} = 1$. Row 2, partials of $f_2 = \sin(x_1 x_2)$: $\frac{\partial f_2}{\partial x_1} = x_2 \cos(x_1 x_2)$, $\frac{\partial f_2}{\partial x_2} = x_1 \cos(x_1 x_2)$.

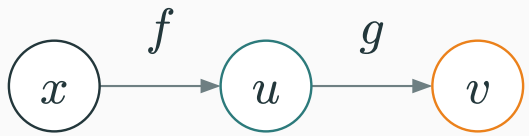
$$J = \begin{pmatrix} 2x_1 & 1 \\ x_2 \cos(x_1 x_2) & x_1 \cos(x_1 x_2) \end{pmatrix}$$

At $(x_1, x_2) = (1, 2)$: $J = \begin{pmatrix} 2 & 1 \\ 2\cos 2 & \cos 2 \end{pmatrix}$.

Each entry is one partial — a 2×2 local linear map, built without ever storing a big matrix.

The chain rule, with Jacobians

D DERIVATION



$$\frac{\partial v}{\partial x} = \frac{\partial v}{\partial u}\frac{\partial u}{\partial x} \quad \Longrightarrow \quad J_{v,x} = J_{v,u}\, J_{u,x}$$

Shapes: with $x \in \mathbb{R}^2, u \in \mathbb{R}^3, v \in \mathbb{R}^2$, then $J_{u,x} \in \mathbb{R}^{3\times 2}$ and $J_{v,u} \in \mathbb{R}^{2\times 3}$:

$$J_{v,x} = J_{v,u}\, J_{u,x} \in \mathbb{R}^{2\times 2}$$

The inner dimension (3) cancels — exactly like ordinary matrix multiplication.

Why we avoid full Jacobians

A single layer with $x, h \in \mathbb{R}^{4096}$ has a Jacobian of $4096 \times 4096 \approx 1.7 \times 10^7$ entries. That is one dense matrix **per layer, per example** — far too big to form and multiply.

Frameworks never materialize J. They compute **vector–Jacobian products** instead.

Vector–Jacobian product = backprop

D DERIVATION

If $y = f(x)$ and $\mathcal{L} = \mathcal{L}(y)$:

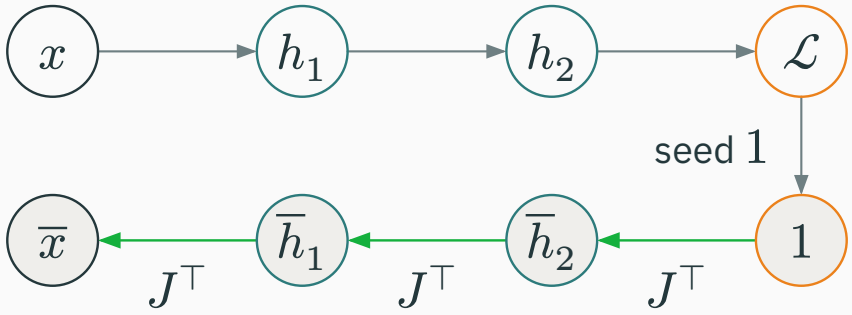
$$\underbrace{\overline{x}}_{\frac{\partial \mathcal{L}}{\partial x}} = J^\top \underbrace{\overline{y}}_{\frac{\partial \mathcal{L}}{\partial y}}$$

one VJP per layer — no matrix ever formed

Backprop is a chain of VJPs

For $x \to h_1 \to h_2 \to \mathcal{L}$:

$$\nabla_x \mathcal{L} = J_{h_1,x}^\top \, J_{h_2,h_1}^\top \, \nabla_{h_2} \mathcal{L}$$



Evaluate **right to left**: $\overline{h}_2 \to \overline{h}_1 \to \overline{x}$ — vectors only.

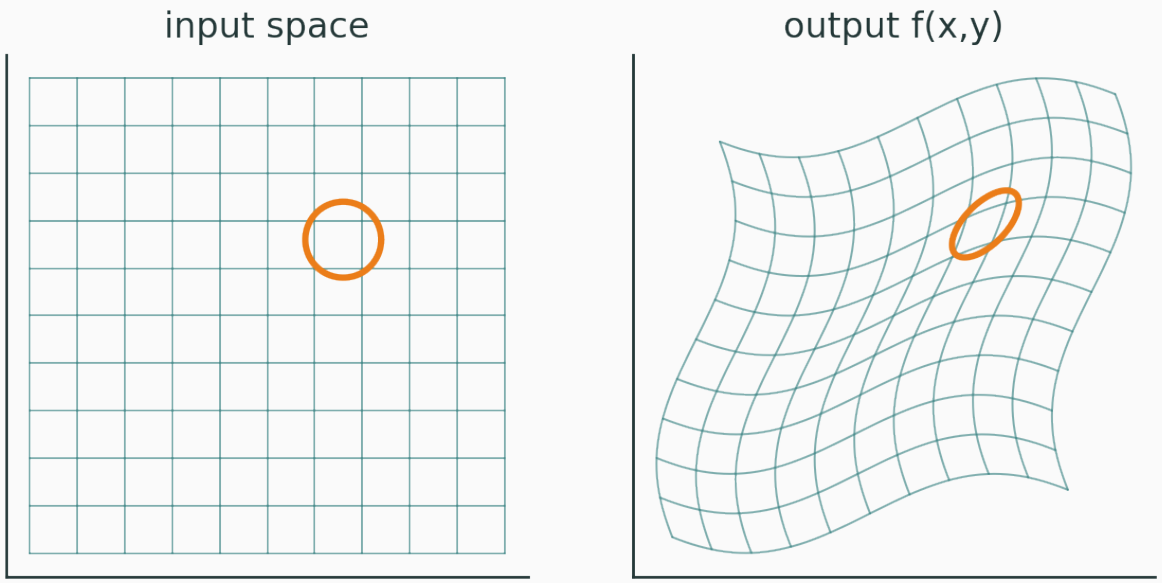
Interactive: the Jacobian as a local warp

I INTERACTIVE

INTERACTIVE

↗ nipunbatra.github.io/interactive-articles/jacobian-local-map

a 2-D map warps a grid; the local Jacobian sends a small circle to an ellipse.



Where each object shows up in deep learning

Derivative → activations

Scalar activation derivatives drive backprop through nonlinearities:

$$\frac{\mathrm{d}}{\mathrm{d}z}\sigma(z) = \sigma(z)(1-\sigma(z))$$

$$\mathrm{ReLU}'(z) = \mathbb{1}[z > 0]$$

Gradient → learning & attribution

- parameter updates: $\theta \leftarrow \theta - \eta \nabla_\theta \mathcal{L}$;
- saliency maps: $\nabla_x f_y(x)$;
- adversarial examples: $x_{\text{adv}} = x + \varepsilon \,\text{sign}(\nabla_x \mathcal{L})$.

Jacobian → sensitivity & flows

Appears in backprop, sensitivity analysis, normalizing flows ($\log|\det J|$), neural ODEs, implicit layers.

But almost always accessed through a product Jv or $J^\top v$ — never the full matrix.

Hessian → curvature & second-order

Newton's method $x_{t+1} = x_t - H^{-1}\nabla f$, Laplace approximation, sharpness / flatness, influence functions.

But the full H is usually too large — we use Hessian–vector products Hv instead.

Why DL stays first-order

For P parameters, $\nabla_\theta \mathcal{L} \in \mathbb{R}^P$ but $H \in \mathbb{R}^{P \times P}$.

$$P = 10^8 \Longrightarrow H \textbf{ has } 10^{16} \textbf{ entries}$$

So practice uses SGD, momentum, Adam, and diagonal / Hessian-vector approximations.

The autodiff connection

Autodiff is none of the naive options

not symbolic differentiation

not finite differences

Automatic differentiation applies the chain rule to the **executed program** — exact, and cheap.

Scalar loss → reverse mode

Training maps $\theta \in \mathbb{R}^P \to \mathcal{L} \in \mathbb{R}$ (many inputs, one output).

reverse-mode autodiff computes $\nabla_\theta \mathcal{L}$ in one forward-pass cost

Non-scalar output needs a seed vector

D DERIVATION

If $y = f(x) \in \mathbb{R}^m$, `backward()` must know **which** scalar to differentiate — you supply a vector v and it returns (same vector as the $J^\top \overline{y}$ above, shaped like x)

$J^\top v$.

This is why PyTorch errors on `.backward()` of a non-scalar tensor without a `gradient=` argument.

PyTorch: Jacobian and VJP

I INTERACTIVE

Compute the very Jacobian we found by hand — then a VJP:

```
import torch
x = torch.tensor([1.0, 2.0], requires_grad=True)
def f(x):
    return torch.stack([x[0]**2 + x[1], torch.sin(x[0]*x[1])])

J = torch.autograd.functional.jacobian(f, x)   # full 2x2 Jacobian
v = torch.tensor([1.0, 3.0])
f(x).backward(v)                                # x.grad == J^T v  (a VJP)
```

`jacobian` forms the whole matrix; `backward(v)` returns only $v^\top J$.

PyTorch: the Hessian

I INTERACTIVE

```
def g(x):
    return x[0]**2 + 3*x[1]**2 + x[0]*x[1]

H = torch.autograd.functional.hessian(g, x)     # -> [[2, 1], [1, 6]]
```

$H = \begin{pmatrix} 2 & 1 \\ 1 & 6 \end{pmatrix}$ — matches the second partials by hand.

Follow-along: exact Jacobian / VJP / Hessian Colab — calculate first, then verify.

Summary

One table to remember

V VISUAL

Six objects, six shapes, six jobs:

Object	Shape	Main DL use
$f'(x)$	scalar	activation derivative
∇f	d	parameter update
J	$m \times n$	vector-map sensitivity
$J^\top v$	n	backpropagation
H	$d \times d$	curvature
Hv	d	second-order methods

Final mental model — one idea

derivative = slope

gradient = direction of steepest ascent

Jacobian = local linear map

Hessian = curvature

backprop = an efficient chain of vector–Jacobian products

We can now read every derivative object in deep learning.

Next: **computation graphs and backpropagation** — the chain rule, run backward through the program.