LECTURE 8 CHEATSHEET

# Shape-safe vector autograd

Backprop is a right-to-left chain of vector–Jacobian products.

**Every tensor receives a gradient with the same shape as that tensor.**

## 1 · Name the local derivative by shape

| Map | Local object | Shape |
|---|---|---|
| $u \in \mathbb{R} \to v \in \mathbb{R}$ | $\partial \frac{v}{\partial} u$ | scalar |
| $\boldsymbol{x} \in \mathbb{R}^d \to z \in \mathbb{R}$ | $\nabla_{\boldsymbol{x}} z$ | $d$ |
| $\boldsymbol{x} \in \mathbb{R}^d \to \boldsymbol{z} \in \mathbb{R}^m$ | $J_{ij} = \partial \frac{z_i}{\partial} x_j$ | $m \times d$ |

Rows of $J$ correspond to outputs; columns correspond to inputs.

## 2 · The vector chain rule

For $\boldsymbol{x} \in \mathbb{R}^d \to \boldsymbol{z} \in \mathbb{R}^m \to \mathcal{L} \in \mathbb{R}$:

$$\underbrace{\nabla_{\boldsymbol{x}} \mathcal{L}}_{d\times 1} = \underbrace{J^T}_{d\times m} \underbrace{\nabla_{\boldsymbol{z}} \mathcal{L}}_{m\times 1}.$$

The arriving vector $\boldsymbol{g}_z = \nabla_{\boldsymbol{z}} \mathcal{L}$ is the upstream gradient. The product $J^T \boldsymbol{g}_z$ is a **vector–Jacobian product** (VJP).

Frameworks compute VJPs directly; they usually never build the full Jacobian.

## 3 · Why the transpose appears

If $\boldsymbol{z} = (x_1 + x_2, 2x_1 - x_2)$, then

$$J = \begin{pmatrix} 1 & 1 \\ 2 & -1 \end{pmatrix}.$$

With upstream $\boldsymbol{g}_z = \begin{pmatrix} 3 \\ 4 \end{pmatrix}$,

$$\boldsymbol{g}_x = J^T \boldsymbol{g}_z = \begin{pmatrix} 1 & 2 \\ 1 & -1 \end{pmatrix} \begin{pmatrix} 3 \\ 4 \end{pmatrix} = \begin{pmatrix} 11 \\ -1 \end{pmatrix}.$$

Each input adds the contribution arriving through every output path.

### Seed rule

A scalar loss starts reverse mode with $\partial \frac{\mathcal{L}}{\partial} \mathcal{L} = 1$. A non-scalar output needs an explicit seed vector $\boldsymbol{v}$; backward returns $J^T \boldsymbol{v}$.

## 4 · Scalar affine node

$$z = \boldsymbol{w}^T \boldsymbol{x} + b.$$

Local gradients:

$$\frac{\partial z}{\partial \boldsymbol{w}} = \boldsymbol{x}, \quad \frac{\partial z}{\partial \boldsymbol{x}} = \boldsymbol{w}, \quad \frac{\partial z}{\partial b} = 1.$$

If upstream $g_z$ arrives, multiply every local gradient by it:

$$\boldsymbol{g}_w = g_z \boldsymbol{x}, \quad \boldsymbol{g}_x = g_z \boldsymbol{w}, \quad g_b = g_z.$$

## 5 · Dense layer, one example

Forward:

$$\boldsymbol{z} = W\boldsymbol{x} + \boldsymbol{b}$$

with $W \in \mathbb{R}^{m\times d}$, $\boldsymbol{x} \in \mathbb{R}^d$, $\boldsymbol{z}, \boldsymbol{b} \in \mathbb{R}^m$.

Backward for upstream $\boldsymbol{g}_z \in \mathbb{R}^m$:

$$\boldsymbol{g}_x = W^T \boldsymbol{g}_z, \quad \boldsymbol{g}_b = \boldsymbol{g}_z, \quad \boldsymbol{g}_W = \boldsymbol{g}_z \boldsymbol{x}^T.$$

Outer product gives the parameter-gradient matrix. Check: $(m \times 1)(1 \times d) = m \times d$.

## 6 · Elementwise nonlinearities

If $\boldsymbol{h} = \varphi(\boldsymbol{z})$ elementwise, then

$$\boldsymbol{g}_z = \boldsymbol{g}_h \cdot \varphi'(\boldsymbol{z}).$$

The local Jacobian is diagonal, so the VJP becomes an elementwise product. Common cases:

$$\mathrm{ReLU}'(z) = \begin{cases} 1 z > 0 \\ 0 z < 0 \end{cases}, \quad \sigma'(z) = \sigma(z)(1 - \sigma(z)).$$

**Predict every gradient's shape before calculating its value.**

LECTURE 8 CHEATSHEET

# Matrices, batches, and autograd traps

Use shape contracts to make vectorized backward passes auditable.

## 1 · Dense layer, row-batch convention

Let $X \in \mathbb{R}^{N \times d}$ store examples as rows:

$$Z = XW^T + \mathbf{1}\boldsymbol{b}^T \in \mathbb{R}^{N \times m}.$$

For upstream $G_Z \in \mathbb{R}^{N \times m}$:

$$G_X = G_Z W, \quad G_W = G_Z^T X, \quad \boldsymbol{g}_b = \sum_{n=1}^{N} G_{Z,n:}.$$

If the loss is a mean over examples, its upstream signal already carries the factor $\frac{1}{N}$.

## 2 · Broadcasting means reduction in backward

Forward broadcasts $\boldsymbol{b}$ across $N$ rows. Backward must undo that duplication by summing:

$$\boldsymbol{g}_b = G_Z^T \mathbf{1}.$$

General rule: if an axis was inserted or expanded during forward broadcasting, sum the gradient across that axis in backward.

Silent broadcasting can create plausible but wrong objectives—for example `[N] - [N,1]` becoming `[N,N]`. Assert shapes before loss computation.

## 3 · Branches accumulate

If $x$ feeds two paths, both paths contribute:

$$\frac{\partial \mathcal{L}}{\partial x} = \frac{\partial \mathcal{L}}{\partial a}\frac{\partial a}{\partial x} + \frac{\partial \mathcal{L}}{\partial c}\frac{\partial c}{\partial x}.$$

Reverse mode adds all arriving gradients at the shared node. Reusing a tensor is not “last path wins.”

## 4 · Matrix multiplication mnemonic

For $C = AB$ and upstream $G_C$:

$$G_A = G_C B^T, \quad G_B = A^T G_C.$$

Keep the untouched neighbour and transpose it so dimensions fit. Then verify that $G_A$ matches $A$ and $G_B$ matches $B$.

## 5 · PyTorch reverse-mode checklist

- ✓ Leaf tensors that need gradients use `requires_grad=True`.
- ✓ The forward graph is built by tracked tensor operations.
- ✓ Call `loss.backward()` on a scalar loss.
- ✓ For vector outputs, supply the VJP seed explicitly.
- ✓ Read leaf gradients from `.grad`; clear them before reuse.
- ✓ Do parameter updates under `no_grad()` or through an optimizer.

## 6 · Minimal VJP check

```
x = torch.tensor([1., 2.], requires_grad=True)
y = f(x)                  # shape [m]
v = torch.randn_like(y)
y.backward(v)             # x.grad == J.T @ v
```

For tiny functions, compare against a full Jacobian or finite differences. The agreement should hold in both **value and shape**.

## 7 · Failure modes

| Symptom | Likely cause |
|---|---|
| wrong orientation | mixed row/column conventions; missing transpose |
| gradient too large | forgot mean factor or summed a branch twice |
| bias gradient wrong | forgot reduction over broadcast axes |
| `.grad` doubles | did not clear accumulated gradients |
| `None` gradient | detached graph, non-leaf tensor, or unused path |
| shape looks valid, loss wrong | unintended broadcasting |

## 8 · Final shape audit

**Forward:** write the shape after every operation.
**Backward:** start from the loss and move right-to-left.
**At each node:** multiply by the local derivative, then sum contributions.
**Invariant:** returned gradient shape equals input shape.

**Backprop is scalar chain-rule reasoning, stacked and vectorized without changing the logic.**