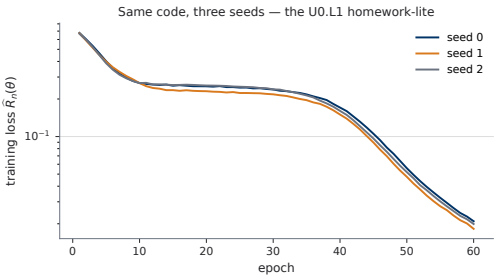
Uo.T2 — Making Training Work

Flow-Based Generative Models · UFRJ · 2026.2

01 Same code, three seeds

The U0.L1 homework-lite, run properly



Same code, same data; only the seed changed. Today the variance is merely *annoying*. In a badly tuned run it decides whether training converges at all.

Question of the day: *which settings (optimizer, schedule, initialization, normalization) make training reliably work, and what are our defaults?*

02

Optimizers: SGD → momentum → Adam → AdamW

Gradient descent meets the minibatch

Full-batch GD on the empirical risk $\widehat{R}_n(\theta)$ costs one pass over the dataset *per step*. Instead: sample a minibatch $B \subset \{1, \dots, n\}$, step on its gradient.

DEFINITION (EMPIRICAL MEASURE, MINIBATCH GRADIENT)

The **empirical measure** of the minibatch, $\widehat{q}_B = \frac{1}{|B|} \sum_{i \in B} \delta_{(x_i, y_i)}$, puts mass $1/|B|$ on each sampled pair, so an expectation under it is simply the batch average:

$$\mathcal{L}_B(\theta) = \mathbb{E}_{(x,y)\sim\widehat{q}_B}\big[\ell(f_\theta(x), y)\big] = \frac{1}{|B|} \sum_{i \in B} \ell(f_\theta(x_i), y_i),$$

and **SGD** iterates $\theta_{k+1} = \theta_k - \eta_k\, g_k$, $g = \nabla_\theta\, \mathcal{L}_B(\theta)$, with learning rate η_k.

With B sampled uniformly, $\mathbb{E}_B[g] = \nabla_\theta \widehat{R}_n(\theta)$: an **unbiased estimator**. Noisy, but pointing the right way on average.

Noise scale vs. batch size

How noisy is g? Across random draws of B it scatters around its mean, and $\mathrm{Cov}_B[g]$ measures that scatter. Averaging $|B|$ independent per-sample gradients divides the covariance by $|B|$:

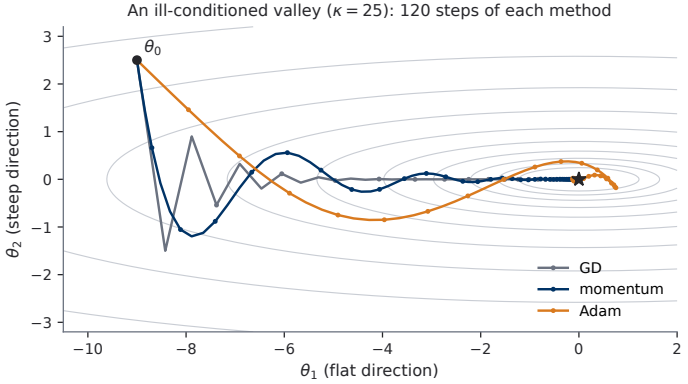
$$\mathrm{Cov}_B\left[g\right] \;\approx\; \frac{1}{|B|}\,\Sigma(\theta), \qquad \Sigma(\theta) = \mathrm{Cov}_{(x,y)\sim\widehat{q}_n}\left[\nabla_\theta\,\ell\big(f_\theta(x),y\big)\right],$$

with $\Sigma(\theta)$ the covariance of a *single* sample's gradient; the noise level shrinks like $1/\sqrt{|B|}$.

- Small batch: noisy, cheap steps (some regularizing jitter). Large batch: cleaner steps, better hardware utilization, but *not* proportionally fewer steps.
- The noise never vanishes: **every curve you saw on the first slide wiggles because of this term.**

Practical rule (recipe card): batch size is chosen by *memory*, then reported.

The real enemy: ill-conditioning



A valley, steep in θ_2 and flat in θ_1: GD's **single** step size must be safe across the valley, hence agonizing along it. Real networks are far worse than κ = 25.

Momentum: an EMA of gradients

Average the gradients instead of trusting the last one:

$$m_k = \beta\, m_{k-1} + (1-\beta)\, g_k, \qquad \theta_{k+1} = \theta_k - \eta\, m_k, \qquad \beta \approx 0.9.$$

- m_k is an **exponential moving average**: oscillating components (across the valley) cancel; consistent components (along it) accumulate. That is the heavy-ball intuition: a ball with inertia rolling down the valley floor.
- PyTorch's `SGD(momentum=0.9)` uses the un-normalized recursion $m_k = \beta m_{k-1} + g_k$: same trajectory family, learning rate rescaled.
- **Nesterov momentum**: evaluate the gradient at the look-ahead point. Named so you recognize it; not derived.

Adam: two EMAs and a per-coordinate step

ADAM (KINGMA & BA, 2015) — PER COORDINATE

$$m_k = \beta_1 m_{k-1} + (1-\beta_1) g_k \qquad \text{(first moment: direction)}$$

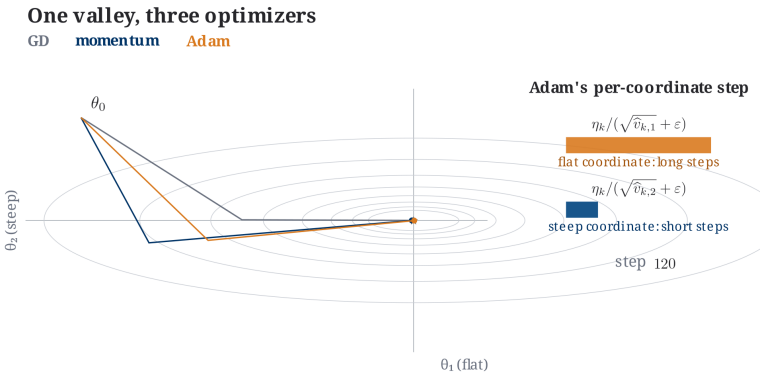
$$v_k = \beta_2 v_{k-1} + (1-\beta_2) g_k^2 \qquad \text{(second moment: scale)}$$

$$\hat{m}_k = \frac{m_k}{1-\beta_1^k}, \quad \hat{v}_k = \frac{v_k}{1-\beta_2^k} \qquad \text{(bias correction: next slide)}$$

$$\theta_{k+1} = \theta_k - \eta_k \frac{\hat{m}_k}{\sqrt{\hat{v}_k} + \varepsilon} \qquad \text{(defaults: } \beta_1 = 0.9,\ \beta_2 = 0.999,\ \varepsilon = 10^{-8}\text{)}$$

Each coordinate gets its own effective step $\eta_k/(\sqrt{\hat{v}_k} + \varepsilon)$: coordinates with historically large gradients step small, and vice versa. That is the ill-conditioning fix, per coordinate and for free.

The race, animated



one step size cannot serve two curvatures, unless it is per-coordinate

Animated in the web deck — final frame shown.

Live derivation: the bias-correction factor

EXAMPLE (BIAS CORRECTION, 3 LINES)

The EMAs start at $m_0 = 0$, so early estimates are *biased toward zero*. Unroll, then take expectations, assuming the gradient distribution is roughly stationary over the window ($\mathbb{E}_B[g_i] \approx g$):

$$m_k = (1-\beta)\sum_{i=1}^{k} \beta^{k-i} g_i \quad \Longrightarrow \quad \mathbb{E}_B[m_k] \;\approx\; g\,(1-\beta)\sum_{i=1}^{k} \beta^{k-i} = g\,\big(1-\beta^{k}\big).$$

Dividing by $1-\beta^k$ removes the startup bias; as k grows the correction retires. □

AdamW: weight decay is not L2 when steps are adaptive

L2 regularization adds $\frac{\lambda}{2}\|\theta\|_2^2$ to the loss; its gradient $\lambda\theta$ shrinks every weight each step ("weight decay"). Under SGD the two views coincide. **Under Adam they do not:**

- **L2 in the loss:** $\lambda\theta$ joins g_k and is divided by $\sqrt{\hat{v}_k}$: large-gradient coordinates are barely decayed, an accident of gradient history.
- **Decoupled decay (AdamW):** shrink the weights outside the adaptive machinery,

$$\theta_{k+1} = \theta_k - \eta_k \left(\frac{\hat{m}_k}{\sqrt{\hat{v}_k} + \varepsilon} + \lambda\, \theta_k \right),$$

  so every weight decays at the same rate $\eta_k\lambda$, as intended.

COURSE DEFAULT (LAW)

The course optimizer is AdamW, with today's recipe-card defaults, from U0.L2 through U3. Deviations must be justified in writing.

Honest empirics: what AdamW does *not* win

- On vision **classification**, well-tuned SGD + momentum often generalizes slightly better than Adam-family optimizers. That literature is real.
- Nobody trains large **generative** models that way: time-conditioned objectives with heterogeneous gradient scales across t are exactly the regime adaptive methods handle well.
- Our default is chosen for the models *we* build, not as a universal claim.

WHY THIS MATTERS LATER

Every flow / CFM / DDPM lab in this course (U1.L1 … U3.L3) trains with AdamW under today's defaults, and PS0 requires them. When a later loss $\mathbb{E}_{t,\, x_1 \sim q,\, x \sim p_t(\cdot|x_1)}[\cdots]$ mixes easy and hard time steps in one batch, the per-coordinate scaling is doing quiet work.

03 Schedules: warmup + cosine

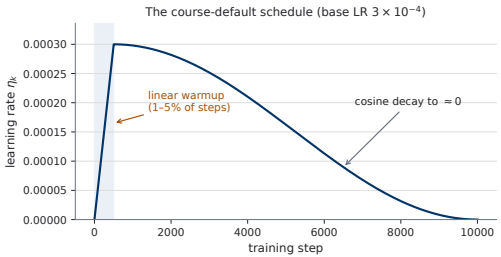
Why warmup exists

The bias-correction slide had a hidden warning: for small k, $\widehat{v}_k$ is an estimate built from a handful of samples.

- At step 10, the second-moment EMA has seen ~ 10 gradients: $\widehat{v}_k$ is **garbage**, and dividing by $\sqrt{\text{garbage}}$ makes some coordinates take enormous steps.
- Early training is also where the loss surface is least friendly: the network is at a random init, gradients are large and unrepresentative.
- Large adaptive steps at random init destabilize runs that would otherwise be fine. The classic symptom is a loss spike (or NaN) in the first few hundred steps.

Fix: start the learning rate near zero and ramp it up *linearly* while the moment estimates fill up.

The course schedule, in one picture



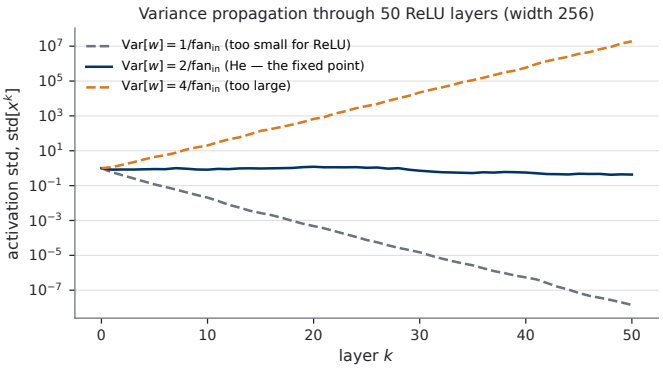
Terminology: a **step** is one optimizer update on one minibatch; an **epoch** is a full pass over the data, $\approx n/|B|$ steps. Course schedules are defined *per step*.

COURSE DEFAULT (LAW)

Linear warmup over 1–5% of total steps, then cosine decay to $\approx$ 0. Anything else in a later lab must be justified in writing.

04 Initialization

The failure mode: exponential in depth



Depth multiplies scales: a per-layer factor $c \neq 1$ becomes c^L, geometric growth or death, inherited by the gradients through the backward pass. At depth 50, the wrong constant costs 10^7.

Live derivation: variance propagation

Setup: $y = Wx$, entries of W i.i.d. with mean 0 and variance σ_w^2, independent of x, whose coordinates are i.i.d. with mean 0.

RESULT (VARIANCE THROUGH A LINEAR LAYER + RELU)

$$\mathrm{Var}[y_i] = \sum_{j=1}^{d} \mathrm{Var}[w_{ij}x_j] = d\,\sigma_w^2\,\mathrm{Var}[x] \qquad (d = \mathrm{fan}_{\mathrm{in}}).$$

A ReLU halves the second moment of a symmetric input, $\mathbb{E}\big[\,\mathrm{ReLU}(z)^2\big] = \frac{1}{2}\mathbb{E}[z^2]$, so one linear + ReLU layer maps $\mathrm{Var}[x] \mapsto \frac{1}{2}\,d\,\sigma_w^2\,\mathrm{Var}[x]$; the scale is preserved iff

$$\sigma_w^2 = 2/\mathrm{fan}_{\mathrm{in}} \qquad \textbf{(He initialization).}$$

Xavier/Glorot ($\sigma_w^2 = 2/(\mathrm{fan}_{\mathrm{in}} + \mathrm{fan}_{\mathrm{out}})$) is the tanh/linear analogue (no ReLU halving to undo). Stated, not derived.

Initialization in practice

- **Trust nothing by default.** PyTorch's `nn.Linear` init is *not* He: check what your framework does before assuming what a paper assumed. (You will verify this in U0.L2.)
- **Zero-init the last layer** when "output ≈ 0 at start" is sensible: the network begins as (near) the zero function, and training grows it from there.
  - $\rightarrow$ For our velocity fields u_t^θ, small outputs at init mean early training is not fighting a random vector field. Standard practice from U3.L1 on.
  - $\rightarrow$ The same idea returns, dressed up, as **adaLN-zero** in the DiT preview (U0.T4).

WHY THIS MATTERS LATER

U0.L2's sabotaged runs include a **bad-init** variant: you will be handed loss curves and asked to name the disease. Today's figure (flat vs. dying vs. exploding scale) is the diagnostic chart.

05 Normalization and residual connections

What normalization actually buys

Folklore first: BatchNorm was introduced (2015) to fix “internal covariate shift”, the story that layer-input distributions drifting during training is the core problem. The explanation **did not survive**; the method did.

What normalization *demonstrably* does:

- allows **higher learning rates** without divergence;
- reduces **sensitivity to initialization** (Board 2’s disease, treated at run time instead of init time);
- smooths the optimization landscape (empirically well documented; mechanism still debated).

Mechanism on the next slide; then the knife.

BatchNorm: mechanism and the train/eval split

BATCHNORM (PER CHANNEL C, OVER MINIBATCH B)

$$\mu_c = \mathbb{E}_{(x,y)\sim\widehat{q}_B}\left[x_c\right], \quad \sigma_c^2 = \mathrm{Var}_{(x,y)\sim\widehat{q}_B}\left[x_c\right], \quad \mathrm{BN}(x_c) = \gamma_c \frac{x_c - \mu_c}{\sqrt{\sigma_c^2 + \varepsilon}} + \beta_c,$$

with learned γ_c, β_c; statistics taken over the batch **and** spatial positions.

- **Train mode:** normalize with the *current batch's* statistics; keep running averages on the side.
- **Eval mode:** normalize with the *running* statistics. The model must work on a single sample.
- Two behaviors, one module: `model.train()` vs `model.eval()`. Forgetting the switch is a classic silent bug (U0.L2 will make you feel it).

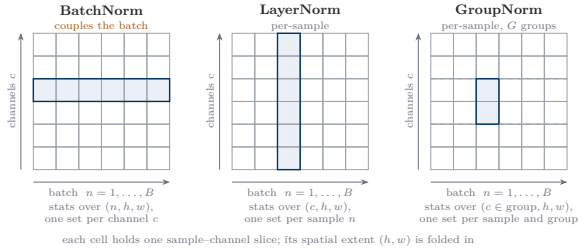
The knife: batch statistics are a liability

Every sample's output now depends on *who else is in the batch*:

- **Small batches** → noisy statistics → normalization itself injects noise (train/eval gap grows).
- **Pointwise evaluation** → there is no batch; running averages must faithfully stand in.
- **Generative models condition on a per-sample time** t**:** a batch mixes samples at $t \approx 0$ (pure noise) and $t \approx 1$ (nearly data). Their activation statistics *should* differ; batch-normalizing across them couples what the model is trying to keep apart.

So the models we build need per-sample normalization. Two candidates, one picture →

LayerNorm and GroupNorm



LN: per sample, all channels (transformers / DiT). **GN:** per sample, channel *groups* (U-Nets). Neither couples the batch; no train/eval split.

COURSE DEFAULT (LAW)

> Generative models here use **GroupNorm (U-Nets) or LayerNorm (transformers)**; BatchNorm only in this bootcamp's CIFAR-10 classifier.

Residual connections

Deep stacks fail even when each layer is healthy: by depth 50, plain stacks are barely trainable. The fix is one plus sign:

$$x_{k+1} = x_k + f_k(x_k).$$

The gradient highway, in one line: the Jacobian $\partial x_{k+1}/\partial x_k = I + \partial f_k/\partial x_k$ has an identity term, so the backward cotangent always has a direct path. Products of Jacobians no longer die like Board 2's geometric chain, and depth stops being the enemy: this is what makes 50-block U-Nets and 30-block DiTs trainable.

Keep this innocuous plus sign in mind. In U1.T2 it becomes a theorem, and in U2 it becomes the whole course.

06 CNNs and inductive bias

Convolution: a constrained linear map

A convolution layer *is* a linear map, with two constraints baked in:

1. **Locality:** each output looks at a $k \times k$ window, not the whole image.
2. **Weight sharing:** the same window weights are used at every position, so shifting the input shifts the output: **translation equivariance**.

The striking number, CIFAR-10 sized ($32 \times 32 \times 3 = 3072$ values):

Layer	Parameters
Fully connected, $3072 \to 3072$	≈ 9.4M
Conv 3×3, $3 \to 64$ channels	**1,792**

Three orders of magnitude fewer parameters, not by shrinking the model but by *encoding an assumption about images*. (Where 1,792 comes from: next slide.)

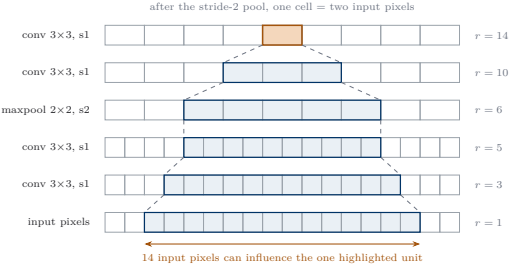
Anatomy of a convolution layer

- **Kernel (filter):** a $k \times k \times C_{\text{in}}$ block of weights. Sliding it over the input and taking a dot product at each position produces one **feature map**; a layer with C_{out} kernels outputs C_{out} channels.
- **Stride s:** the window moves s pixels at a time; $s > 1$ downsamples the output by a factor of s.
- **Padding:** zeros added around the border so windows fit at the edges; "same" padding keeps the spatial size unchanged.
- **Pooling:** parameter-free downsampling: take the max (or mean) over each window, typically 2×2 with stride 2, halving the resolution.

Parameter count: $k \cdot k \cdot C_{\text{in}} \cdot C_{\text{out}} + C_{\text{out}}$ (one bias per output channel). The table's conv layer: $3 \cdot 3 \cdot 3 \cdot 64 + 64 = 1{,}792$.

Receptive fields

The **receptive field** of a unit: the input region that can influence it. *Strides multiply how fast it widens.*



$$r_L = 1 + \sum_{l=1}^{L} (k_l - 1) \prod_{i=1}^{l-1} s_i \qquad (k_l, s_l = \text{kernel size, stride of layer } l).$$

Receptive fields: the board arithmetic

EXAMPLE (THE DIAGRAM'S STACK, WORKED)

Stack: conv 3×3 → conv 3×3 → maxpool 2×2 stride 2 → conv 3×3 → conv 3×3. Stride products before each layer: $1, 1, 1, 2, 2$, so

$$r = 1 + 2 + 2 + 1 \cdot 1 + 2 \cdot 2 + 2 \cdot 2 = 14.$$

Two cheap 3×3 convs *after* a downsample buy what one expensive 9×9 would before it: **striding is how deep layers get to see the whole image.**

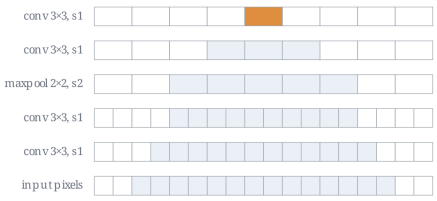
Note the pool's own term: kernel 2, stride product 1 before it, so it contributes $(2 - 1) \cdot 1 = 1$.

The cone, animated

The receptive field, term by term

$$r \;=\; 1 + 2 + 2 + 1 + 4 + 4$$

$$r = 14$$



the same two convs placed before the pool would add 2 pixels each; after it, 4 each

Animated in the web deck — final frame shown.

Feature hierarchies, and architecture as prior

- Stride/pooling builds a **pyramid**: early layers see edges (small RF, high resolution); deep layers see objects (large RF, low resolution). This hierarchy is the empirical signature of CNN features.
- The frame to keep: **architecture = prior about data.** Convolution asserts "statistics are translation-invariant and local"; when the assumption fits, you pay thousands of parameters instead of millions.
- The general concept is **equivariance**: build the symmetry into the map rather than learning it. One sentence today; it returns in U6 (Track C).

WHY THIS MATTERS LATER

The U-Net (U0.T4 / U0.L4) is exactly *CNN inductive bias + multiscale processing*: today's RF arithmetic is why its deep bottleneck sees the whole image, which a velocity field at low t genuinely needs.

07 The recipe card

The course training defaults

RECIPE CARD — OPTIMIZER-SCHEDULE-DEFAULTS (CITE AS: THE U0.T2 DEFAULTS)

1. **Optimizer:** AdamW, base LR 3×10^{-4}, β = (0.9, 0.999), weight decay 0.01, **excluding** normalization parameters and embeddings from decay.
2. **Schedule:** linear warmup over 1–5% of total steps, then cosine decay to ≈ 0.
3. **Init:** He for ReLU-family activations; **zero-init the last layer** where "output ≈ 0 at start" is sensible (velocity fields u_t^θ: yes).
4. **Normalization:** GroupNorm (U-Nets) / LayerNorm (transformers) for generative nets; BatchNorm only in bootcamp classifiers.
5. **Batch size:** the largest that fits in memory, and *always reported*.

Close: next session (U0.L2)

- **Next session (U0.L2):** this recipe meets CIFAR-10 for real. Schedules, logging (MLflow), checkpointing, seeding.
- One of the runs you will be given **has been sabotaged**. You will diagnose it from the curves alone; today's pictures are the diagnostic charts.

See you next session.