

U2.L1 — Solvers and Adjoint in Practice

Flow-Based Generative Models · UFRJ · 2026.2

01 | Orientation and the entry quiz

Where this session sits

Session	Role
U2.T1	the adjoint theorem, and a table of two gradient strategies whose cells were arguments
U2.L1 — today	every cell of that table measured , on a classifier where nothing hurts
U2.L2	the same instruments, pointed at a generative model

You learn the instruments on a problem where nothing is at stake, so that the next reading means something.

Six instruments you leave with

1. An **NFE counter**: three lines.
2. **NFE-against-quality axes**, on two metrics that disagree.
3. **NFE drift** across training.
4. A **peak-memory protocol** for one forward-and-backward pass.
5. A **gradient-fidelity table**: two routes against one witness.
6. **Honest timing**: compile apart from steady state.

Nothing today is about generative models. The instruments come first.

Given code, and a loop written out

Module	Built in	Used today
common/toy2d.py	U1.L1, frozen	two_moons(n, seed)
common/odesolve.py	U0.L5, frozen	odeint, with .nfe
common/train.py	U0.L1, U0.L2	set_seed only

The labels come from the frozen contract (first `n // 2` rows = upper moon):

```
x = two_moons(n, seed)
y = (torch.arange(n) >= n // 2).long()
```

The training loop is written out: the NFE counter must be read **between** the forward solve and the backward pass of every step.

Where the numbers go

MLflow, deployed locally: `sqlite:///mlflow.db` beside the notebook.

- Experiment U2; one run per block.
- Tags `session=U2.L1` and `block=<n>`; the resolved configuration as parameters.
- Curves logged **after** the loop, in one batch. The loop times every step (median 53 ms); a SQLite write inside it would be part of what it times.

The entry quiz: six questions

1. What does NFE count, and why is it reported for every solve?
2. Where does `rtol` dominate, and where `atol`?
3. Why is a step rejected, and what does a rejection cost?
4. What is `dopri5`?
5. What decides the cost of a fixed-step solve? Of an adaptive one?
6. What does *stiff* mean, loosely?

Four or more right: go on. Fewer: read UO.T5, *Adaptive stepping* and *Stability*, before the sweep block.

02 | A Neural ODE classifier on two moons

The model

$$\dot{x}(t) = f_{\theta}(t, x(t)), \quad x(0) = x_0, \quad t \in [0, 1],$$
$$\text{logits} = W \psi_1(x_0) + b$$

- $f_{\theta}(t, x)$: an MLP on $[x; t]$, two hidden layers of 32, tanh. With the head, 1,256 parameters.
- **The state stays in the plane**, $\mathbb{R}^2$: a picture of the trajectories is a picture of the state, not a projection.
- **The activation is smooth** (tanh): f_{θ} is C^1 in (t, x, θ) and globally Lipschitz in x , uniformly in t : both tiers of U2.T1's contract. Tier 2 is what the adjoint assumes.

The loss, and what the flow must do

$$\ell(\theta) = \mathbb{E}_{(x_0, y) \sim \mathcal{D}_{\text{train}}} \left[-\log \text{softmax} (W \psi_1(x_0) + b)_y \right]$$

$\mathcal{D}_{\text{train}}$: the empirical (uniform) distribution of the 512 labelled training points. Each term is U2.T1's terminal loss $L_y(x(1))$, with $L_y = \text{head} + \text{cross-entropy}$ against y .

The head alone, on the raw inputs: 89.4% test accuracy. No line separates the moons; the rest is the flow's work.

TODO 1.1 — the NFE counter

THREE LINES ARE YOURS

```
class NFECOUNTER(nn.Module):
    def __init__(self, f):
        super().__init__()
        self.f = f
        self.nfe = 0 # 1
    def forward(self, t, x):
        self.nfe += 1 # 2
        return self.f(t, x) # 3
```

A module holding the field, so its parameters are the field's. It counts **every** call: stages, start-up calls, and the adjoint's backward calls. Checkpoint 1.1: Euler with N steps must report N ; torchdiffeq's rk4 must report $4N$.

TODO 1.2 — train, and read the gauges

ONE STEP, WITH THE GAUGES READ

```
for step in range(300):  
    model.field.nfe = 0  
    logits = model(x, method="dopri5")  
    nfe_fwd = model.field.nfe  
    loss = F.cross_entropy(logits, y)  
    opt.zero_grad(set_to_none=True)  
    loss.backward()  
    opt.step(); sched.step()  
    nfe_bwd = model.field.nfe - nfe_fwd
```

dopri5 at its defaults: $\text{rtol} = 10^{-7}$, $\text{atol} = 10^{-9}$. Full batch, 512 points. Record the loss, the accuracy, the forward NFE, the backward NFE and the seconds of every step.

Settings, and one deviation in writing

The Uo.T2 recipe card: AdamW, weight decay 0.01, 3% warmup, cosine decay, per step. **Base rate:** 3×10^{-3} , not the card's 3×10^{-4} .

Base rate	Steps	CPU time	Loss	Accuracy
3×10^{-4} (card)	1500	77 s	0.0275	99.6%
3×10^{-3}	300	15.1 s	0.0032	100.0%

Loss and accuracy on the training set; CPU time on one thread. At step 300 of the card-rate run the loss is still 0.287. The card rate, with five times the steps, ends 8.5 times higher. Test accuracy of the 3×10^{-3} run: 100.0%.

Test NFE at the defaults: 116. Backward NFE: **zero**, at every step.

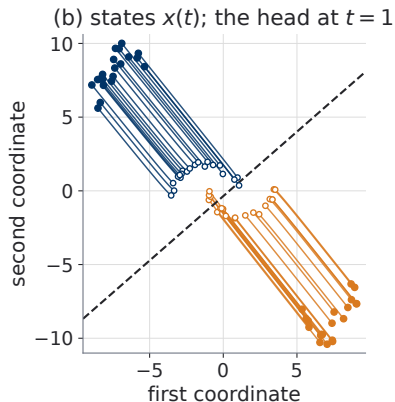
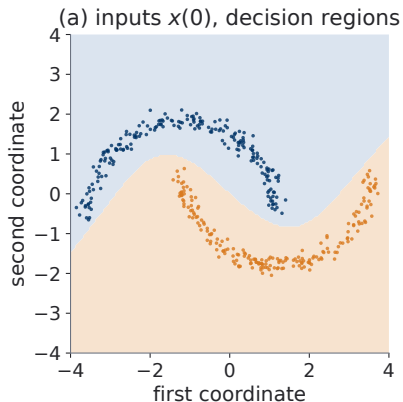
Why the head starts at zero

Card item 3: zero the last layer where “output ≈ 0 at start” is sensible. Two last layers qualify.

- The field's: the flow starts at the **identity**.
- The head's: the first prediction is **uniform**, loss $\log 2$.

Measured: with PyTorch's default head, seeds 2 and 3 of five **stalled at every rate tried** (final training loss 0.23 or above; every other run 0.048 or below). With the head at zero, 10 seeds of 10 reached 100%.

Checkpoint 1 — the state, in the plane



Left: every fifth test input; regions from one solve per grid point. Right: hollow $t = 0$, filled $t = 1$, dashed the head's line.

Uniqueness, as a check

Under U2.T1's Tier 1 (Picard–Lindelöf), two distinct trajectories never occupy the same state at the same time. Read over all 2,096,128 test pairs and 41 times:

$$0.41 \leq \frac{|x^{(i)}(t) - x^{(j)}(t)|}{|x^{(i)}(0) - x^{(j)}(0)|} \leq 21.4.$$

Paths drawn in the plane may still cross: the field depends on t , and a crossing of paths happens at **different** times. None of the 40 drawn paths do; yours may, and that is not a contradiction.

Optional — your RK4 against the library's

N	odesolve RK4	torchdiffeq rk4	the two RK4s
8	5.4×10^{-3}	5.0×10^{-3}	3.4×10^{-4}
16	4.3×10^{-4}	4.1×10^{-4}	2.5×10^{-5}
32	3.1×10^{-5}	2.9×10^{-5}	1.6×10^{-6}
64	2.1×10^{-6}	2.0×10^{-6}	9.9×10^{-8}

Largest coordinate error of the endpoints against a tight `dopri5` reference; the last column is the difference between the two RK4s. `torchdiffeq`'s `rk4` is the **3/8 rule**, not UoT5's classical tableau: both order 4, both four calls per step, two different computations.

03 | Solver and tolerance sweep

TODO 2.1 — fifteen settings, one checkpoint

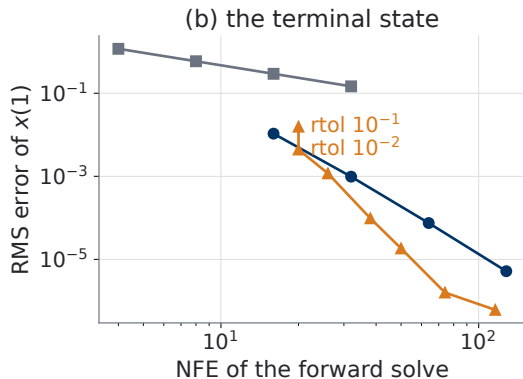
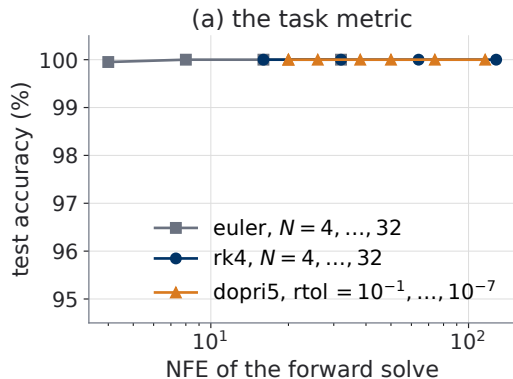
- euler, rk4 with $N \in \{4, 8, 16, 32\}$ fixed steps.
- dopri5 with $\text{rtol} \in \{10^{-1}, \dots, 10^{-7}\}$, $\text{atol} = \text{rtol} \times 10^{-2}$.

Evaluation only, in **float64**, against a reference at $\text{rtol} = 10^{-12}$.

A HYPOTHESIS TO TEST, NOT A LAW

The task metric may saturate before the trajectory error does. Then tighter tolerances spend evaluations without changing the answer.

TODO 2.2 — two axes that disagree

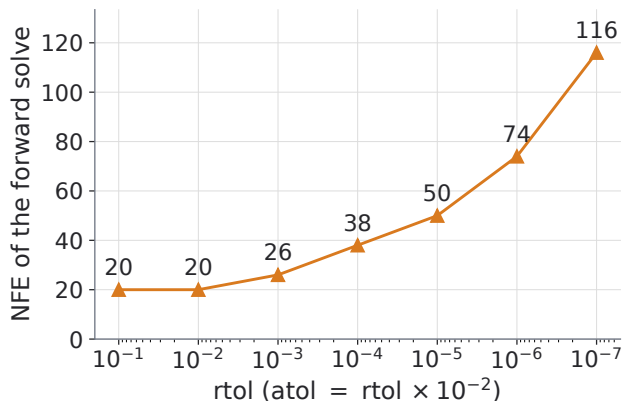


What the two panels say

- **Accuracy is 100% in 14 of 15 settings.** The exception: Euler, 4 steps, 99.95%.
- **The terminal error spans more than 6 decades**, from 1.18 down to 6.1×10^{-7} .
- At similar cost: rk4, 16 steps, 64 calls, error 7.5×10^{-5} ; dopri5 at $\text{rtol} = 10^{-6}$, 74 calls, 1.6×10^{-6} . Fifth-order adaptive against fixed-step fourth-order: the order alone is not isolated.

On this task, accuracy is **nearly blind** to solver error: Euler with 4 steps is off by 1.18 in RMS (3.34 at worst) and misses 1 point.

NFE against tolerance



A log-log fit over $\text{rtol} \leq 10^{-3}$ gives slope -0.16 . It describes **this** model at **this** checkpoint, and predicts nothing about the next.

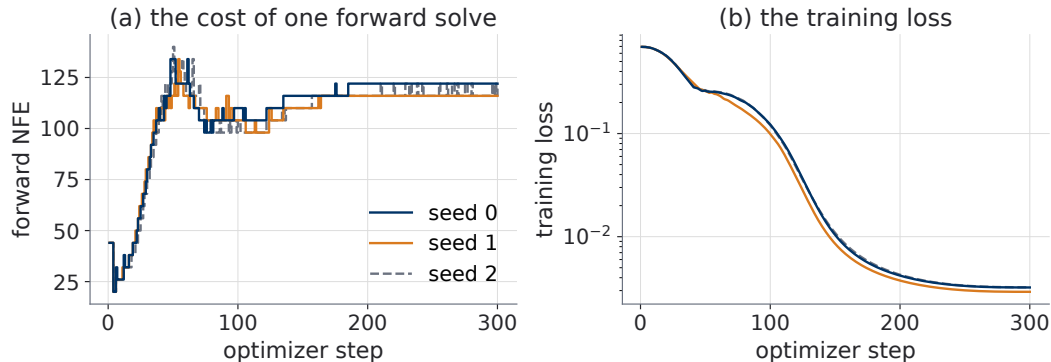
One step size for the whole batch

torchdiffeq's controller reduces the error estimate to one number: an RMS over **every** element of the batched state.

Solve	NFE at the defaults
the 2048 test points, one batch	116
64 of them, both moons, one at a time	50 to 134 (median 74)

60 of 64 points pay more in the batch than alone, and the hardest gets **fewer** evaluations than it asks for alone. The batch NFE is a property of the batch.

TODO 2.3 — NFE across training



A solver setting fixes a tolerance, not a cost; the cost follows the field.

What happened in three runs

- **Steps 1–3 cost 44, by arithmetic.** The field is still exactly zero (learning rate 0 at step 1; at step 2 the field's gradient is still zero, so its last layer stays zero), so torchdiffeq's initial-step rule falls back to $h = 10^{-6}$, every error estimate is zero, and the step grows by the maximum factor, 10. Seven steps, two start-up calls and six per step: $2 + 6 \times 7 = 44$.
- **Then it falls, and rises.** Down to 20 at step 4, the first nonzero field (seed 0); peaks of 134–140 at steps 48–54, while the loss drops from $\log 2$ to about 0.26; down to 104–110 at step 100, then 116–122 to the end.
- **The seeds agree closely.** Only the field's hidden layers are random; both last layers start at zero.

A description of three runs, not a law about Neural ODEs.

04 | Memory and gradient fidelity

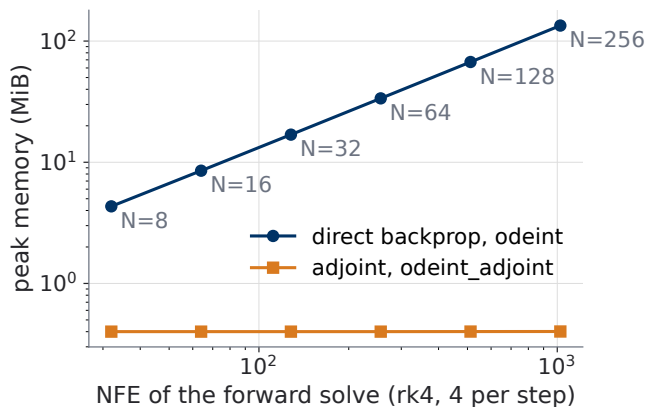
The protocol

PEAK MEMORY, GPU (CANONICAL)

```
torch.cuda.synchronize(); torch.cuda.empty_cache()
torch.cuda.reset_peak_memory_stats()
base = torch.cuda.memory_allocated()
loss = F.cross_entropy(model(x, adjoint=adj,
    method="rk4", options={"step_size": 1 / N}), y)
loss.backward()
torch.cuda.synchronize()
peak = torch.cuda.max_memory_allocated() - base
```

tracemalloc is **not** a tensor meter: it sees Python's allocator, not the native one PyTorch uses. Without a GPU: the process's resident set size (RSS), sampled, **coarse**.

TODO 3.1 — memory against steps



U2.T1's schematic, with numbers: peak over one forward-and-backward pass. RTX 4070 Laptop GPU, batch 512, float32.

The reading, and the bill

- **Direct** backprop: 4.33 MiB at $N = 8$ to 134.26 MiB at $N = 256$, about 134 KiB per call.
- **Adjoint**: 0.40 MiB at every N : constant in the number of steps, for a fixed set of output times. U2.T1's claim, at its strength.
- **The bill**: on this fixed grid the adjoint's backward makes as many field calls as the forward (1024 at $N = 256$); direct backprop makes zero. Time: 1.9 times as long at $N = 8$, 1.6 times at $N = 256$, here.
- **CPU fallback** (direct backprop, sampled RSS): tracks the GPU from $N = 32$ up (127.8 against 134.26 MiB at $N = 256$); at $N = 16$ it read 0.0 MiB against 8.52.

TODO 3.2 — the fidelity ritual

ℓ_B : the loss of the **exact** flow on a fixed batch B of 256 test points, the loss of the slide *The loss, and what the flow must do* with $\mathcal{D}_{\text{train}}$ replaced by B (uniform over B). ℓ_N : the same through N steps of rk4, the loss as the computer evaluates it. Float64; one unit direction v over the field's parameters θ_f ; $\varepsilon = 10^{-6}$.

$$\begin{aligned} d_{\text{FD}} &= \frac{\ell_N(\theta + \varepsilon v) - \ell_N(\theta - \varepsilon v)}{2\varepsilon}, & d_{\text{AD}} &= v^\top \nabla_{\theta_f} \ell_N, \\ d_{\text{ADJ}} &= v^\top \hat{g}_N \text{ (adjoint's estimate of } \nabla_{\theta_f} \ell_B), & d_{\text{cont}} &= v^\top \nabla_{\theta_f} \ell_B \end{aligned}$$

d_{cont} is computed two ways that agree to 1.5×10^{-15} .

The fidelity table

N	AD vs FD (rel.)	ADJ vs FD (rel.)	AD vs cont (abs.)	ADJ vs cont (abs.)
8	6×10^{-9}	2×10^{-4}	1.3×10^{-7}	8.9×10^{-8}
32	5×10^{-9}	3×10^{-6}	7.8×10^{-10}	9.6×10^{-11}

Relative gaps divide by $|d_{\text{FD}}| \approx 2.34 \times 10^{-4}$. Absolute gaps to FD: AD 1.4×10^{-12} and 1.2×10^{-12} ; ADJ 4.5×10^{-8} and 6.8×10^{-10} .

- Direct autodiff matches the witness to the order a central difference allows: the exact gradient of the program that ran, as far as the witness can see.
- The adjoint answers a different question, the exact flow's, discretized.
- Against the exact flow, neither is exact; here the adjoint is closer at both N .

A caution about the ritual's tolerance

The ritual accepts d from either route when $|d - d_{\text{FD}}| \leq 10^{-8} + 10^{-6} \max(|d|, |d_{\text{FD}}|)$.

On this trained model the directional derivative is small and the 10^{-8} floor dominates: the adjoint **fails** at $N = 8$ and **passes** at $N = 32$.

Neither verdict says it is wrong: the ritual catches a **wrong** gradient, and here its verdict on a legitimate route turns on N . The relative columns separate the routes, and they are well conditioned here: $|d_{\text{FD}}|$ is eight orders of magnitude above the witness's own error.

Margin note — the backward solve can drift

U2.T1: where the forward flow contracts, the backward reconstruction **amplifies** error. Nothing in the table isolates that effect; on a harder field it is one way the adjoint's column grows.

Mitigations, named and not oversold:

- tighter tolerances on the backward solve, paid in NFE;
- direct backprop, when the problem is small enough to store;
- checkpointing: store a few states, reconstruct only between them.

05 | The Diffrax mirror

Same classifier, in JAX

The same model in JAX and Diffrax, about thirty lines, loaded with today's weights.
Two checks first, both measured:

- **The port agrees with torch:** terminal states within 2.7×10^{-13} at a tight tolerance.
- **Diffrax reports steps, not calls;** a `jax.debug.callback` counts 49 calls for 8 steps of `Dopri5`: six per step, one at the start.

Two of the routes behind one argument, `adjoint=`:
`RecursiveCheckpointAdjoint` (the default) and `BacksolveAdjoint`.

TODO 4.1 — honest timing

Adjoint	N	trace + lower + compile	steady, per call
RecursiveCheckpointAdjoint	8	1.43 s	9.3 ms
BacksolveAdjoint	8	1.68 s	8.0 ms
RecursiveCheckpointAdjoint	32	1.40 s	42.1 ms
BacksolveAdjoint	32	1.55 s	31.9 ms

Value and gradient, 256 points, float64, Intel Core Ultra 9 185H CPU, one run. The up-front price is the whole first-call overhead; at $N = 8$ it equals about 153 steady calls of the checkpointed route. Do not rank frameworks with this table.

TODO 4.2 — two philosophies, one metric

N	RecursiveCheckpointAdjoint vs FD	BacksolveAdjoint vs FD
8	2.1×10^{-8}	3.4×10^{-5}
32	3.8×10^{-10}	2.7×10^{-8}

Relative gaps, same direction v , same batch, fixed-step `Dopr15`.

- **Checkpointed**: discretize, then optimize. Agrees with its own program.
- **Backsolve**: optimize, then discretize; the continuous adjoint of U2.T1. DiffraX's documentation marks it “not recommended” (Kidger 2021).
- The ritual's verdict passes **both** routes at both N ; only the size of the gaps tells them apart.

Two libraries, two emphases

- **Diffrax** defaults to checkpointed backpropagation.
- **torchdiffeq** still exposes the classical continuous adjoint as its prominent memory-saving route; there the route is the function you call, `odeint` or `odeint_adjoint`.

Neither is wrong; they weigh one trade differently. Know which route you run before you read “we trained with the adjoint”.

The numerics behind Diffrax’s defaults: Kidger (2022, ch. 5), optional.

06 | The instrument panel

The panel

THE INSTRUMENT PANEL (U2.L1)

Instrument	What it read today
NFE counter	every call, forward and backward
NFE-vs-quality axes	accuracy flat; error across 6 decades
NFE drift in training	20 at step 4, 122 at step 300
Peak-memory protocol	direct grows; the adjoint stays flat
Gradient-fidelity table	two routes, two different numbers
Honest timing	compile apart from steady state

Every gauge on this panel gets pointed at a generative model in the FFJORD lab — and there, the needles move. Watch them.

What comes next

- **Next session (U2.T2):** back to the question U2.T1 left on the board.
- **The lab after it (U2.L2):** today's instruments, on the model that question leads to.
- **The solutions notebook** is released today.

Sources for today: torchdiffeq (Chen 2018), Diffrax (Kidger 2021), the Neural ODE paper (Chen et al. 2018, sec. 2), and Massaroli et al. (2020, App. A.1).

References

- Chen, Ricky T. Q. 2018. “torchdiffeq: ODE Solvers and Adjoint Sensitivity Analysis in PyTorch.” <https://github.com/rtqichen/torchdiffeq>.
- Chen, Ricky T. Q., Yulia Rubanova, Jesse Bettencourt, and David Duvenaud. 2018. “Neural Ordinary Differential Equations.” In *Advances in Neural Information Processing Systems*. <https://arxiv.org/abs/1806.07366>.
- Kidger, Patrick. 2021. “DiffraX: Numerical Differential Equation Solvers in JAX.” <https://github.com/patrick-kidger/diffrax>.
- . 2022. “On Neural Differential Equations.” PhD thesis, University of Oxford. <https://arxiv.org/abs/2202.02435>.
- Massaroli, Stefano, Michael Poli, Jinkyoo Park, Atsushi Yamashita, and Hajime Asama. 2020. “Dissecting Neural ODEs.” In *Advances in Neural Information Processing Systems*. <https://arxiv.org/abs/2002.08071>.