tramdag documentation

tramdag — Interpretable Neural Causal Models (TRAM-DAGs) in PyTorch

> **Status: beta (0.x), under active development.** The API may change between releases until 1.0; pin a version (`tramdag==0.2.*`) for reproducibility.

TRAM-DAGs model each variable of a structural causal model with a (transformation-model) flow: one triangular normalizing flow from iid standard-logistic latents to the observed variables. The structure is of the triangular Adjacency Matrix is exactly your causal DAG. Fit it **once** on observational data and answer all three rungs of Pearl's causal hierarchy — observational (L1), interventional (L2, the do-operator), and counterfactual (L3, Pearl abduction) — while keeping **interpretable effects**: every linear-shift coefficient is a log-odds ratio, exactly as in classical proportional-odds models.

> Beate Sick & Oliver Dürr, *Interpretable Neural Causal Models with TRAM-DAGs*, CLeaR 2025 (arXiv:2503.16206). This repo is the reference implementation (PyTorch, built on zuko); all of the paper's experiments are replicated here with pinned tests.

5-minute showcase: the Colab badge above fits the paper's bimodal benchmark live (GPU-ready) and walks L1 → L2 → L3, every answer checked against analytic ground truth. Further notebooks are available at `notebooks/` like the didactic walkthrough of the model: `notebooks/intro_tram_dag.py`.

Install

```
pip install tramdag            # latest release (PyPI)
pip install "git+https://github.com/tensorchiefs/
```

```
tramdag.git@main"   # dev version (track main)
        uv sync                         # or: dev setup from a clone
(tests, experiments)
```

Pin the dev install to a commit for reproducibility, e.g. `...tramdag.git@<sha>`.

30 seconds of API

```
        import tramdag as td
        from tramdag
import CausalFlowDAG, ContinuousNode, OrdinalNode, I, LS, CS

        spec = {  # the spec IS the labelled DAG
            "Age": ContinuousNode(),
            "mRS_pre": OrdinalNode(levels=6, terms=[I("Age")]),
            "NIHSSa": ContinuousNode(terms=[I("Age"), LS("mRS_pre")]),
            "T": OrdinalNode(levels=2, terms=[I("Age"), LS("mRS_pre"),
CS("NIHSSa")]),
            "mRS_3m": OrdinalNode(
                levels=7, terms=[I("Age"), LS("mRS_pre"), CS("NIHSSa"),
LS("T")]
            ),
        }
        flow = CausalFlowDAG(spec)  # validates acyclicity, builds the
flow

        # self-stopping training: per-node plateau lr decay + freezing of
converged
        # nodes (exact, since the per-node NLLs have independent
gradients);
        # see docs/training-speed.md for benchmarks and the classic two-
phase recipe
        flow.fit(
            train_df,
            val_df,
            epochs=4000,
            learning_rate=1e-2,
            schedule="plateau",
            plateau_patience=30,
            freeze_patience=120,
        )

        # all-`ls` model? fit it classically instead: deterministic
float64 L-BFGS,
        # exact MLE matching statsmodels/R (see notebooks/
classical_fit_tram_dag.py)
        flow.fit_classical(train_df)  # raises on cs/ci specs

        flow.log_prob(df)  # L1: joint log-likelihood per row
```

```
        flow.sample(1000)  # L1: observational sampling
        flow.sample(1000, do={"T": 1})  # L2: interventional (graph
mutilation)
        flow.pmf(df, node="mRS_3m", do={"T": 1})  # L2: analytic
interventional PMF

        u = flow.abduct(df)  # L3 step 1: latents from observations
        cf = flow.sample(do={"T": 1}, u=u)  # L3 steps 2+3:
counterfactuals

        flow.ls_coefficients()  # interpret: per-edge log-odds-ratios
        flow.intercept_contributions("NIHSSa", df)  # interpret: per-
parent partial effects
        # of an additive complex intercept (centered)

        # heterogeneous treatment effects: a small, penalized effect head
beta(x)*T
        # (VC term) with a first-class read-out — see docs/varying-
coefficients.md
        # e.g. terms=[CS("Age", "NIHSSa"), VC("T", "Age")] ->
        # flow.varying_coef("mRS_3m", df)          # beta(x):
deterministic, y-free

        flow.scores(df, node="mRS_3m")  # per-observation scores dl_i/
dtheta
        flow.effect_modifier_scan(df, "mRS_3m", on="T")  # which VC
modifiers? (CUSUM
        # scan from a cheap all-ls fit) — docs/scores.md

        flow.save("flow.pt")
        flow = CausalFlowDAG.load("flow.pt")

        td.simulations.REGISTRY  # synthetic DGPs with known ground truth
```

The model in one table

Per node, the transformation is additive on the latent (log-odds) scale — `u = h(x; θ) + Σ β·x_pa + Σ g(x_pa)` — and each parent edge declares how it enters:

term	meaning	interpretability
`LS(pa)`	linear shift `β·x_pa`	`exp(β)` is an odds ratio — one number per edge
`CS(pa)`	complex shift `g(x_pa)` (MLP), still additive	plot `g`
`I(pa)`	complex intercept: the transform's parameters depend on the parents	maximal flexibility, interactions not interpretable

term	meaning	interpretability
	(several parents in one `I(...)` feed one joint network)	
`VC(on, *mods)`	varying-coefficient shift `β(mods)·x_on`	read out with `flow.varying_coef`

Continuous nodes carry a monotone 1-D transform (`bernstein` — TRAM-faithful default, `spline`, `affine`; `ContinuousNode(transform=..., transform_kwargs=...)`); ordinal nodes an ordered-logit head `P(x ≤ k) = σ(θ_k − shift)`. Abduction is exact for continuous nodes and truncated-logistic for ordinal ones, so `flow.sample(u=flow.abduct(df))` reproduces `df` exactly / level-exactly.

There are two ways to fit the model: a stochastic deep learning optimizer (`fit`) and a 2nd order optimization like in the classical statistical models (`fit_classical`). The latter is more efficient for all-`ls` models (each node-conditional is then a classical transformation model). For more details, see the `docs/fitting.md` file.

Validation (all pinned by tests)

- **Paper replication** — each of the paper's four DGP families has a generator in the registry (numpy-only SCM + frozen CSVs + replication script) and is pinned by tests:

family	paper	demonstrates
`triangle` (`linear`,`atan`,`sin`)	§6.1	LS coefficient recovery (β = 2, −0.2, +0.3), CS curve ≡ −f(x_2), non-monotone f
`triangle-mixed` (`linear`,`exp`)	§6.2	mixed data L1/L2 + the C.4 odds-ratio check (OR ≈ 7.4)
`vaca`	§5.1–5.2	the bimodal L1 case a default CNF misses; L2 `p(x₃ | do(x₂))`
`carefl`	§5.3	L3 counterfactual curves vs **analytic** truth

  ```
  cd experiments && uv run python paper_triangle.py atan cs   # etc., see paper_*.py
  ```

  Sign note: ordinal shifts are *subtracted* here but *added* in the paper, so fitted ordinal weights are the paper's with flipped sign (`truth.json` records both conventions per family).

- **Exact classical equivalence** — an all-`ls` flow trained to convergence *is* the proportional-odds MLE: coefficients match `statsmodels` **and** R `MASS::polr` to ~4 decimals (`experiments/validate_ls.py`, R reference committed under `data/magic-mrclean/*/ref_ls/`).
- **Training speed** — schedules, per-node freezing, LBFGS and device benchmarks: `docs/training-speed.md`.

What the tests actually guarantee — the principles behind them (known identities, the datasets and software they compare against, and how the ground truth was obtained) — is documented in `tests/README.md`.

Full storyline, clinical-data context, R cross-check and reading notes: `docs/stroke-case-study.md`.

Testing policy

See the `tests/README.md` file for more details.

Layout

```
src/tramdag/            spec.py transforms.py conditioners.py
flow.py
                        simulations/   (magic_mrclean, triangle,
vaca, carefl,
                                        vc_shift + CLIs)
data/                   frozen synthetic CSVs + truth.json — a
test contract
experiments/            stroke pipeline, paper replications,
training benchmark
notebooks/              intro (didactic) + Colab demo
(jupytext .py — see README there)
tests/                  unit, known-truth recovery, R regression
docs/                   training-speed.md, stroke-case-study.md,
                        varying-coefficients.md, scores.md
```

Implementation conventions (latent-scale signs, raw/one-hot parent encoding, log-space ordinal likelihood, seeding) are documented in `CLAUDE.md` and pinned by tests.

Citation

If you use `tramdag`, please cite the method paper:

```bibtex
@inproceedings{sick2025tramdag,
  title     = {Interpretable Neural Causal Models with TRAM-DAGs},
```

```
          author    = {Sick, Beate and D{\"u}rr, Oliver},
          booktitle = {Proceedings of the 4th Conference on Causal
Learning and Reasoning (CLeaR)},
          series    = {Proceedings of Machine Learning Research},
          volume    = {275},
          year      = {2025},
        }
```

Fitting a TRAM-DAG: how training works

Technical reference for `CausalFlowDAG`: how the flow is built, how the likelihood is computed, and the two fitting paths — the stochastic optimizer (`fit`) and the classical one (`fit_classical`) — plus per-node freezing, the memory model, and where optimizer choice could go next.

How the flow is built: one module, one sub-model per node

A `CausalFlowDAG` is a single `torch.nn.Module`, but it is **not one monolithic network**. At construction (`CausalFlowDAG.__init__`) it topologically sorts the DAG and builds an `nn.ModuleDict` with **one `_Node` sub-module per variable**. The nodes share *no parameters*; each owns the pieces for its own conditional `p(x_i | pa(x_i))`:

- an **intercept** producing the transform parameters `θ`: `SimpleIntercept` (a free parameter vector, data-independent) — or, if the node has `ci` parents, `ComplexIntercept` (a small MLP whose output `θ` depends on those parents);
- a **monotone 1-D transform** `h` (`BernsteinUT / SplineUT / AffineUT`) — note the transform itself carries **no learnable weights**, only the fitted range buffers `xmin`/`xmax`; its shape is set entirely by `θ` from the intercept;
- a `ModuleDict` of **shift modules**, one per parent edge: `LinearShift` (`ls`, a single weight) or `ComplexShift` (`cs`, an MLP).

So "one network or several?" — **one module, several independent per-node sub-models** (each itself a small intercept + shifts assembly). They are bundled into one module and trained by one optimizer, but their parameters are disjoint. The DAG structure lives entirely in *which parents each node reads*; there is no edge weight matrix or shared trunk.

`_Node.theta_shift` assembles a node's `θ` (from the intercept) and total `shift` (sum of its shift modules over the parent features) for a batch; parent features are encoded by `_encode_parent` — continuous parents raw, ordinal parents one-hot.

How the likelihood is computed

A TRAM-DAG maps iid standard-logistic latents `U` to the observed `X` in causal order; because node `i` reads only its parents (earlier variables, as *data*), the Jacobian of `U → X` is **triangular**, so its log-determinant is the sum of the per-node 1-D terms. The joint log-likelihood therefore **decomposes per node**:

```
log p(x) = Σ_i log p(x_i | pa(x_i))
```

`CausalFlowDAG.node_log_prob` computes one term per node and `log_prob` sums them. For a node, given `θ, shift` from `theta_shift`:

- **continuous** — change of variables through the monotone transform:

  ```
  z = h(x; θ) + shift
  log p(x | pa) = log f_logistic(z) + log |dz/dx|
  ```

  i.e. the standard-logistic density at the latent `z` (`StandardLogistic.log_prob`) plus the transform's log-derivative (`ladj`, returned by `ut.forward` — the 1-D Jacobian term that makes it a proper density, not just a score).

- **ordinal** — an ordered-logit / proportional-odds head, `P(x ≤ k) = σ(θ_k − shift)`, evaluated as the log of the cutpoint-interval probability by `ordinal_log_prob` (done in log-space via `logsigmoid`/`log1mexp`, because the naive sigmoid difference underflows to exactly-zero gradients in float32).

The training loss is the summed per-node **mean** NLL over the batch (`Σ_i mean_rows(−log p(x_i | pa))`); `log_prob` instead returns the per-row joint for scoring whole observations.

Consequence used by both optimizers: because parents enter as data, the per-node gradients are independent — optimizing the summed loss jointly is identical to optimizing each node separately. This is what licenses per-node learning rates, per-node freezing (below), and the all-`ls` classical fit.

Path A — stochastic optimization (`fit`)

`CausalFlowDAG.fit` is the general-purpose trainer (required for any `cs`/`ci` edge). Mechanics:

- **Adam** with **one parameter group per node** — built from `self.nodes[name].parameters()`, so each node can carry its own learning rate.
- **Minibatches**: a fresh `torch.randperm` shuffle each epoch; the loss is the summed per-node mean NLL on the batch.
- **Transform ranges** are set once at the start from the train 5%/95% quantiles (`_set_ranges`), defining each Bernstein/spline domain.
- **Learning-rate schedules** (`schedule=`): `None` (constant), `"onecycle"`, `"cosine"`, or `"plateau"` — the last decays each node's lr off *its own* validation NLL.
- **Per-node freezing** (`freeze_patience=`, see below).
- **`restore_best`**: optionally snapshots each node's best-validation weights and restores them at the end (early-stopping regularization). Default `False` so the fit sits at the training-data MLE.

How per-node freezing works

Set `freeze_patience=N` to let converged nodes drop out of training. Each epoch, `fit` tracks every node's own validation NLL and counts epochs since its last improvement (by more than `min_delta`). A node is **frozen** once that counter reaches `N` (under `schedule="plateau"`, only after its learning rate has already been decayed, so a node isn't frozen while a smaller step would still help).

Freezing is not just "stop updating" — the frozen node is **excluded from the computation entirely**: `node_log_prob(values, nodes=active)` is called with only the still-active nodes, so a frozen node runs no forward and no backward pass. That is a real FLOP saving, and it is *correct* precisely because the per-node gradients are independent (above): removing a converged node's term from the loss does not change any other node's gradient. The slowest node sets the total training time; the rest stop costing anything once they have converged.

When **every** node is frozen the fit returns early (the self-stopping behavior). Freeze epochs are recorded in `flow.history["frozen"]`; the freeze state is local to a single `fit` call (a later `fit` call trains all nodes again). This composes with `restore_best`, which still restores each node's best-validation snapshot at the end.

Freezing vs. parallelism (why one helps and the other usually doesn't)

It is tempting to reason "freezing a node speeds things up, so node computation costs time, so computing nodes *in parallel* should also speed things up." The premise is right but the conclusion doesn't follow, because the two act through different mechanisms:

- **Freezing removes work** — the frozen node's forward/backward simply never runs again. Removing work is an *unconditional* win, and most of its payoff is structural: once *all* nodes freeze the fit stops, deleting whole epochs — something parallelism can never do (it can shrink time-per-epoch, not the number of epochs).
- **Parallelism only overlaps work** — it runs the same computations concurrently, which helps *only if the hardware is sitting idle* during the sequential node loop. It usually isn't: parallelism is already extracted one level down, where each node's batched tensor ops run across all rows and the BLAS/GPU backend already uses all cores. While node A's matmul runs the cores are busy, so doing node B "at the same time" just time-slices the same cores — contention, often slightly slower.

So freezing speeding things up does **not** imply spare capacity to parallelize across nodes. The exception is the regime where per-node ops *under-utilize* the hardware — small batches, tiny models, or many small nodes on a big GPU where each kernel leaves it mostly idle. There, overlapping nodes can help, but the right tool is **fusion** (batch same-shaped nodes into one larger tensor op, or CUDA streams), not threading the Python loop (which the GIL fights). That is the "vectorize/fuse the per-node loop" direction in Optimizer choice below — a conditional win in that regime, distinct from freezing's unconditional one.

Benchmarks, schedule trade-offs, and the recommended self-stopping recipe are in training-speed.md; the worked walkthrough is `notebooks/intro_tram_dag.py`.

Path B — classical optimization (`fit_classical`)

`CausalFlowDAG.fit_classical` is the dedicated optimizer for **all-`ls`** models, where every node-conditional is a classical transformation model (ordered-logit / Colr). It raises on any `cs`/`ci` edge.

- **Full-batch, float64, L-BFGS** (strong-Wolfe line search) — no minibatches, no schedule, no early stopping ⇒ **deterministic**

(same init → bit-identical) and on the **exact MLE** (matches `statsmodels`/R to ~1e-3 on well-identified coefficients).
- **float64 is a transient compute mode**: `self.double()` upcasts parameters *and* the transforms' range buffers in one call; the fit runs in double; a `finally: self.float()` restores float32. The stored model — and `save`/`load` — stay float32. The data path (`_tensorize`, `sample`, `pmf`) reads the model dtype so it follows along.
- **Convergence** is judged by NLL flatness and is *advisory*: a Bernstein intercept and weakly-identified directions (rare one-hot levels, a flat treatment-effect ridge) keep drifting along zero-curvature valleys after the likelihood is at the optimum. Correctness is verified against classical software (`experiments/validate_ls.py --classical`), not the flag.
- Read the fitted coefficients with `ls_coefficients()`.

Walkthrough: `notebooks/classical_fit_tram_dag.py`.

Memory and disk during fitting

Neither fitting path writes to disk. Everything during `fit`/`fit_classical` lives in RAM:

- model parameters and (for `fit`) the per-node Adam optimizer state;
- the `history` dict (per-epoch train/val NLL, lr, wall-clock) — an in-memory attribute, not a file;
- `restore_best` snapshots — `copy.deepcopy` of node `state_dicts` held in memory (`flow.py`), never serialized.

The **only** disk I/O in the module is the explicit, user-called `save` (`torch.save` of spec + state_dict + history) and its counterpart `load`. So a fit produces no temp files, no checkpoints, no scratch directory; persistence is opt-in via `flow.save(path)`. (The `results/` and `docs/perf/` artifacts in this repo are written by the *experiment scripts*, not by the library's fitting code.)

Optimizer choice — current and future

Today: **Adam** for flexible models, **L-BFGS** (float64) for all-`ls`. The per-node-param-group structure already in `fit` makes several extensions cheap, and worth considering as the package matures:

- **IRLS / Fisher scoring for the all-`ls` path.** The classical fitting algorithm for proportional-odds / GLM-type models is iteratively reweighted least squares (Newton with the expected information). For the `ls` case it would likely reach the MLE in *fewer, more stable* steps than L-BFGS — and the Fisher information it computes is exactly the ingredient for the

planned **standard-error table** (currently flagged as next in the CHANGELOG). Strong candidate to back `fit_classical` in a future version.
- **Second-order / curvature-aware methods for flexible nodes.** K-FAC or a Gauss-Newton approximation could help the `ci`/`cs` MLPs, though full-batch quasi-Newton is a poor fit for them (minibatch noise also regularizes — see why `fit_classical` refuses them).
- **Modern first-order variants.** AdamW (decoupled weight decay), RAdam (warmup- free), or Lion/Sophia are drop-in alternatives to Adam; the benchmark harness (`experiments/bench_training.py`) is built to evaluate exactly such swaps on time-to-target.
- **Per-node optimizer selection.** Because the loss decomposes and the optimizer already holds one group per node, different nodes could in principle use different optimizers (e.g. L-BFGS for the `ls` nodes, Adam for an MLP node in a mixed model) — an interesting direction for mixed-flexibility DAGs.
- **Warm-start handoffs.** `fit_classical → fit` already works (the classical MLE as a fast, principled initialization); the reverse (Adam to escape, L-BFGS to polish) is a natural pattern to formalize.

These are *directions*, not commitments. The autoresearch loop (`docs/research/MISSION_autoresearch.md`) is a good venue to test the speed-oriented ones empirically before adopting any.

Per-observation scores & the effect-modifier scan

`flow.scores(df, node)` returns each observation's score $\psi_i = \partial \ell_i / \partial \theta$ for the node's interpretable shift coefficients — every `LS` weight (one column per continuous parent, one per one-hot level for an ordinal parent) and every `VC` term's `beta0` (column named after the treatment). The computation is **analytic and exact** (no autograd): shifts enter the latent additively, so $\partial \ell_i / \partial \beta = (\partial \ell_i / \partial s_i) \cdot x_i$ with $\partial \ell_i / \partial s$ in closed form (`src/tramdag/scores.py`). Pure read-out — no fitting or sampling code path is touched. At a fitted MLE the per-column sums are ≈ 0 (pinned by tests, incl. a float64 finite-difference check).

Worked example: where does β(x) need to bend?

The score is the cheapest effect-modifier detector there is (model-based recursive partitioning / structural-change logic — Zeileis & Hornik; Dandl et al. 2024). Before fitting anything expensive: fit the **cheap all-`LS` model** (seconds, deterministic), and scan the treatment-coefficient scores —

```
flow = td.CausalFlowDAG(all_ls_spec, seed=0)
flow.fit_classical(df)  # seconds, exact MLE

scan = flow.effect_modifier_scan(df, node="Y", on="T")
#         stat   p_value  crit_5pct   flag
# X2      4.21    <0.001     1.3581   True     <- true modifier
# X3      3.05    <0.001     1.3581   True     <- true modifier
# X1      0.74     0.64      1.3581  False     <- prognostic only
```

For each candidate covariate the scores are ordered by it and the scaled cumulative sum `B_j = Σ_{i≤j} ψ_(i) / (sd(ψ)·√n)` is formed; under a stable coefficient `B` is a Brownian bridge, so `sup|B|` has the Kolmogorov distribution (5% critical value 1.358). A coefficient that truly varies with the covariate makes the ordered scores drift — flagged covariates are the measured shortlist of `VC` modifiers (`docs/varying-coefficients.md`):

```
spec["Y"] = td.ContinuousNode(
    terms=[td.CS("X1", "X2", "X3"), td.VC("T", "X2", "X3")]
)  # scan-informed
```

Notes: `on` resolves to the identified level-1-vs-0 contrast for a binary ordinal `LS` treatment and to `beta0` for a `VC` treatment. Candidates default to every column of `df` except the node and `on` — modifiers need not be parents. For few-level (heavily tied) candidates the ordering is only partial: read the scan as a ranking diagnostic rather than an exact-size test. Scores also enable influence-function analyses and robust standard errors later.

Case study: the stroke ITE analysis

Background and full detail for the `magic-mrclean` experiments — the public, synthetic counterpart of the clinical analysis in Dürr, Herzog, Bühler, Wegener & Sick, *Estimating Individualized Treatment Effects in Acute Ischemic Stroke with Causal*

Transformation Models (TRAM-DAG) (arXiv:2606.12623). The clinical MAGIC / MR CLEAN data is private and never part of this repo.

The DAG and the pipeline

Five nodes with all forward edges: `Age → mRS_pre → NIHSSa → T → mRS_3m`. Treatment `T` (thrombectomy) is confounded by age and stroke severity in the observational cohort; the estimand is the ATE of `T` on a good outcome, `P(mRS_3m ≤ 2 | do(T))`, averaged over the (younger) trial population — computed analytically from the outcome node's interventional PMF, no Monte Carlo.

```
        cd experiments
        uv run python sim_flow.py nl
# headline storyline (synthetic, default)
        uv run python all_ls_flow.py                  # all-edges-ls flow
        uv run python nihss6_flow.py                  # flexible nihss6
config
        uv run python validate_ls.py                  # all-ls flow vs
statsmodels MLE
        uv run python counterfactual_demo.py all_ls   # Pearl abduction
showcase
        uv run python all_ls_long.py                  # 4x-longer
convergence check
```

Experiments default to the public synthetic source `magic-mrclean/nl`; passing `magic` (clinical cohort, NIHSSa ≥ 6, N = 1275) only works inside the original paper monorepo. Every run uses the same 80/10/10 split (`random_state=42`) and writes plots, per-patient interventional PMFs (`rct_predicted_proba.csv`) and a checkpoint to `results/<name>/`.

The synthetic cohort (`magic-mrclean`)

A hand-specified SCM in the flow's own model family (logistic latents) shaped like the stroke study — same schema, realistic marginals, and **known ground truth** the real data cannot provide: the true ATE, true individual counterfactuals (shared-latent pairs), and a provably misspecified baseline. Two variants:

- **`ls`** — every parent effect is a linear shift: each node-conditional is exactly a classical proportional-odds model, so flow, R reference and truth must coincide. The clean equivalence baseline.
- **`nl`** — adds an accelerating age effect on disability, a heterogeneous treatment effect `tau(Age)` that fades in the elderly, and reduced treatment probability for the very old. An all-`ls` model must collapse `tau(Age)` to a constant and is

therefore biased; a flexible (`ci`/`cs`) flow can recover the truth. The trial cohort (`rct.csv`) enrolls younger than the observational cohort, as real trials do — exactly the extrapolation that breaks the misspecified model.

Known-truth recovery (seed 7), ATE on P(good) over the trial population:

nl variant	ATE	vs true +0.104
naive observational contrast	+0.303	confounded (overstates 2.9×)
all-`ls` flow	+0.076	undershoots (misses the age-varying effect)
flexible (`ci`/`cs`) flow	+0.101	**recovers the truth**

This reproduces the clinical-data finding in miniature (flexible +0.108 vs all-`ls` +0.054) but against a *known* answer.

R cross-check. `data/magic-mrclean/fit_ls.R` refits the all-`ls` DAG node-by-node in classical R (`MASS::polr` / `tram::Colr` / `glm`); its committed `ref_ls/` outputs let `tests/test_simulations.py` assert flow ≡ R fit (outcome-node coefficients and ATE) without an R installation.

Clinical-data results (context — not reproducible from this repo)

model	ATE on P(good)	source
MR CLEAN RCT (ground truth)	**+0.135** [+0.057, +0.213]	Berkhemer et al. 2015
this flow, nihss6 config	**+0.063**	`experiments/nihss6_flow.py`
this flow, all-ls	**+0.057**	`experiments/all_ls_flow.py`
classical proportional-odds MLE, same 80% split	+0.055	`experiments/validate_ls.py`
original TRAM-DAG `md_dag_ls` (all-ls)	+0.054	paper monorepo
original TRAM-DAG `nihss6`	+0.108 (seed 2: +0.092)	paper monorepo
classical MLE, full data	+0.082	paper monorepo

Reading notes:

- **The all-ls flow IS the classical MLE** when trained to convergence without early stopping (the default, `restore_best=False`): on the synthetic `ls` cohort its outcome-node coefficients match `statsmodels` *and* R `polr` to 4 decimals (Age 0.0526, NIHSSa 0.1630, T −0.9424; ATE +0.1429 vs +0.1428) — see `experiments/validate_ls.py` and `test_simulations.py::test_all_ls_flow_is_exact_mle`. The earlier +0.057-vs-+0.055 residual on the clinical data was exactly the early-stopping effect (see CHANGELOG).
- **Flexible (`ci`/`cs`) models are different**: their MLE *overfits the observational confounding*, so they need `restore_best=True` (early-stopping regularization) to recover the causal effect — confirmed on the synthetic `nl` cohort, where the flexible MLE undershoots (+0.076) but the early-stopped fit recovers the true ATE (+0.10), with a lower validation NLL. `run_experiment` defaults `restore_best` per style accordingly (off for all-`ls`, on for flexible).
- The treatment effect is **weakly identified** in this observational cohort: the likelihood around the T-coefficient is nearly flat, and refits drift to the 80%-split optimum of ≈ +0.054 — the original +0.108 is a near-likelihood-equivalent solution, not a sharper optimum. Both flow results sit inside the established acceptance band [+0.03, +0.14]; matching the band, not the point estimate, is the meaningful check.

Counterfactual demo

`experiments/counterfactual_demo.py` (a capability beyond the original scripts) abducts the latents of the 128 held-out test patients, verifies exact factual reconstruction, and predicts each patient's outcome under the opposite treatment (`results/<name>/counterfactuals_test.csv`, `plots/counterfactual_mrs3m.png`).

How fast can a CausalFlowDAG train?

Benchmark of learning-rate schedules, per-node freezing, batch sizes, devices and LBFGS — June 2026, Apple-silicon Mac mini, torch 2.12 (CPU unless noted). Reproduce with `cd experiments && uv run python bench_training.py` (grid ≈ 35 min; raw numbers in `results/bench-training/results.csv`). For a quick **cross-machine** comparison (fixed 200-epoch workloads, all available devices, machine fingerprint to JSON) use the self-contained `experiments/perf_machine.py` — runs on any box after `pip install tramdag`.

The options, and how to use them

Schedules (`fit(..., schedule=...)`) control the learning rate over training:

value	behavior
`None` (default)	constant lr — the classic behavior, exactly as before this PR
`"plateau"`	**per-node**: a node whose own validation NLL hasn't improved by `min_delta` (default 1e-4) for `plateau_patience` epochs (default 15) gets its lr × 0.3, floored at 1e-3 × the initial lr. Each node decays independently — valid because the per-node losses have independent gradients.
`"onecycle"`	warmup to `learning_rate`, then anneal to ~0 over exactly the `epochs` budget (torch `OneCycleLR`) — use only when you know the right budget
`"cosine"`	cosine decay from `learning_rate` over `epochs`

Early stopping / freezing (`fit(..., freeze_patience=N)`): a node whose validation NLL hasn't improved for `N` epochs is *frozen* — removed from the loss and backward pass (real compute saving), its weights fixed from then on. When all nodes are frozen the fit returns early; freeze epochs are recorded in `flow.history["frozen"]`. Under `schedule="plateau"`, freezing additionally waits until the node's lr has been decayed ≥ 100×, so nodes don't freeze while a smaller step size would still make progress. Freezing state is per-`fit()`-call: a second `fit` call trains all nodes again.

Switching it all off — e.g. for an exact comparison with classical methods (`statsmodels`, R `polr`/`tram`): simply omit both arguments. The defaults are unchanged by this PR, so

```
        flow.fit(train_df, epochs=4000, learning_rate=1e-2)  # constant lr, no freezing
        flow.fit(train_df, epochs=2000, learning_rate=1e-3)  # classic two-phase recipe
```

is still the exact-MLE path used by `experiments/validate_ls.py`. (Independent of all this, `restore_best=False` remains the default — see CHANGELOG.) The guard test `tests/test_fit_schedules.py::test_plateau_freeze_preserves_exact_mle` additionally shows that even *with* plateau+freezing the all-`ls` fit lands on the classical MLE within the usual tolerances.

LBFGS is *not* a `fit()` option — it's a classical full-batch optimizer that only makes sense for small, parametric (all-`ls`) models. Recipe (also in `experiments/bench_training.py::run_lbfgs`):

```
        flow = CausalFlowDAG(build_spec("ls"))
        flow._set_ranges(train_df)  # transform ranges from train
quantiles
        vals = flow._tensorize(train_df)
        opt = torch.optim.LBFGS(
            flow.parameters(),
            lr=1.0,
            max_iter=40,
            history_size=30,
            line_search_fn="strong_wolfe",
        )


        def closure():
            opt.zero_grad()
            loss = torch.stack([-lp.mean() for lp in
flow.node_log_prob(vals).values()]).sum()
            loss.backward()
            return loss


        for _ in range(10):
            loss = opt.step(closure)  # full-batch quasi-Newton steps
```

Fast (< 2 s to coefficient-level accuracy) but not robust across seeds (see Findings #2) — use it as a quick first shot with the plateau trainer as fallback.

Method: time-to-target, not loss-go-down

Each config runs once; `fit()` records per-epoch validation NLL *and* wall-clock time, so we read off the seconds until the fit is within a fixed gap of a cached long-run reference (3 torch seeds, medians):

workload	model / data	reference NLL	tight tol	practical tol
stroke-ls	all-`ls` stroke DAG, frozen `magic-mrclean/ls` (n=1275, full-data MLE)	10.3042 (train)	+1e-3	+5e-3
vaca-ci	all-`ci` flow, frozen `vaca` (n=5000, 90/10 split)	4.9632 (val)	+2e-3	+1e-2

Tight ≈ exact-MLE equivalence (statsmodels/R-polr match). *Practical* ≈ coefficient-equivalent: a stroke fit with gap ≈ 3e-3 already matches the R reference coefficients within the test tolerances (`tests/test_fit_schedules.py::test_plateau_freeze_preserves_exact_mle`).

Results

stroke-ls convergence vaca-ci convergence

Median seconds to target (batch 512, cpu; "—" = never reached within budget):

config	stroke-ls practical	stroke-ls tight	vaca-ci practical	vaca-ci tight	self-stops
baseline two-phase (old default)	9.0	**21.4**	2.1	2.8[1]	no (runs 40 s / 15 s)
constant 1e-2	9.1	21.5[2]	2.2	2.8[1]	no
onecycle (1500 / 300 ep)	—	—	3.5	4.5	no
onecycle (3000 ep)	16.8	— (gap 1–2e-3)			no
cosine	—	—	2.2	3.5[1]	no
plateau + freeze	**8.9**	— (gap 2e-3)	**2.0**	2.9	**yes — 13 s / 4 s total**
LBFGS (full-batch)	**1.6** (2/3 seeds)	— (gap 4–8e-3)	n/a	n/a	yes

[1] transient: the val-NLL curve dips through the target and then drifts away (stroke: needs the 1e-3 phase to *stay*; vaca: mild overfitting). Final gap for the vaca baseline is 0.037 — the old 520-epoch budget **underfits** vaca by ~0.03 nats. Plateau+freeze *stays* at its target. [2] constant lr at batch 512 stalls at gap 3–7e-3; only the lr-decay phase closes the last decade (that's exactly why the two-phase recipe existed).

Findings

1. **Per-node plateau decay + freezing is the best default-style trainer.** Same time-to-accuracy as the hand-tuned two-phase schedule, but it needs **no budget tuning**, decays each node's lr off its own validation curve, freezes converged nodes (a real FLOP saving — the per-node NLLs have independent gradients), and **stops itself**: 13 s total vs the baseline's 40 s on stroke-ls, 4 s vs 15 s on vaca-ci, at equal or better final NLL.
2. **LBFGS is spectacular but not robust.** Full-batch LBFGS reaches coefficient-level accuracy on the classical all-`ls` model in **< 2 s** (vs 9 s for Adam) on 2/3 seeds; the third stalls at gap 8e-3. An Adam warm start made it *worse* (different basin), every seed. Use it as a fast first shot with the plateau trainer as fallback, not as the default.
3. **OneCycle is a "spend exactly this budget" scheduler.** Accuracy arrives only at the end of its anneal: at 1500 epochs it misses everything, at 3000 it lands gap 1–2e-3 — but you must know the right budget in advance, which is the problem we're trying to remove.
4. **Full-batch loses on time-to-target** despite ~1.6× higher epoch throughput — too few optimizer steps per second of compute at these n. Batch 512 is a good default; very large batches (16k) only paid off in raw throughput at n=50k.
5. **MPS (Apple GPU) is 3–4× slower than the M-series CPU** at these model sizes (verified correct: identical reconstruction). Kernel-launch overhead dominates sub-millisecond ops. Stay on CPU locally; CUDA on Colab-class GPUs is a different regime (see the demo notebook's GPU-vs-CPU race).
6. **The old defaults waste or under-spend.** Stroke: 4000 epochs budgeted, converged work done after ~1500 (freezing recovers the difference automatically). Vaca: 520 epochs budgeted, ~0.03 nats short of converged. Fixed budgets are wrong in both directions; adaptive stopping fixes both.

Recommendation

For everyday fits:

```
flow.fit(
    train,
    val,
    epochs=4000,
    learning_rate=1e-2,
    batch_size=512,
    schedule="plateau",
    plateau_patience=30,
```

```
            freeze_patience=120,
        )
```

(generous `epochs` as a ceiling — the fit stops itself). For exact classical comparisons where the last 1e-3 matters, append a short constant-lr polish phase (`epochs=500, learning_rate=1e-3`) after the plateau fit, or run the old two-phase recipe.

We deliberately did **not** change `fit()`/`run_experiment` defaults in this PR (default changes are their own reviewed decision — see the `restore_best` episode in CHANGELOG.md). If this report convinces us, flipping `experiments/common.py::run_experiment` to the plateau recipe is a 3-line follow-up.

Varying-coefficient treatment effects: `VC(on, *modifiers, penalty=)`

A `VC` term gives a node a **treatment-effect head with its own bias–variance budget** (issue #28): it contributes

```
beta(modifiers) * x_on        with    beta(x) = beta0 + b_theta(x)
```

to the node's shift, where `b_theta` is a deliberately small (one 16-unit hidden layer), **penalized** network. The point is not expressiveness — it is that the effect function is *estimated with care* instead of falling out as a by-product:

```
        import tramdag as td

        spec = {
            "X1": td.ContinuousNode(),
            "X2": td.ContinuousNode(),
            "X3": td.ContinuousNode(),
            "T": td.OrdinalNode(levels=2, terms=[td.LS("X1"),
td.LS("X2")]),
            "Y": td.ContinuousNode(
                terms=[
                    td.CS("X1", "X2", "X3"),  # prognostic part g(x): as
flexible as you like
                    td.VC(
                        "T", "X2", "X3", penalty=1.0
                    ),  # effect part beta(X2, X3) * T: small + penalized
                ]
            ),
        }
        flow = td.CausalFlowDAG(spec, seed=0).fit(train, val,
restore_best=True)
```

```
        beta = flow.varying_coef("Y", df_new)  # (n,) array beta(x) —
deterministic, y-free
        beta0 = float(flow.nodes["Y"].shifts["T"].beta0)  # interpretable
main effect
```

Note that `X2`/`X3` appear **twice**: prognostically through `CS` and as effect modifiers through `VC`. That is the intended pattern — only the treatment `on` owns its edge (declaring it in a second term raises), modifiers may repeat.

Why not `CS("T", "X2", "X3")`? (anti-pattern)

For a binary treatment the multi-parent `CS` is *expressively* equivalent — any shift decomposes exactly as `s(x,t) = s(x,0) + [s(x,1) − s(x,0)]·t`. But it has **no effect-specific regularization**: the likelihood rewards fitting `s(x,t)` on average and nothing rewards a smooth arm-difference, so the read-out is the difference of two jointly-fitted unregularized networks — noise-amplifying. Measured on the `vc-shift` validation DGP task class, the `CS` reduced form reaches corr ≈ 0.5 against the true effect function **even when the model is exactly in-class** (tramdag-simu#18 / PR #21), while the `VC` term reaches corr ≈ 0.99 on the same protocol (`tests/test_vc_term.py`, acceptance bar 0.9). What makes causal forests / R-learners work is not that they *target* the effect but that they **regularize** it (Nie & Wager 2021; Athey–Tibshirani–Wager 2019); `VC` brings that ingredient into the TRAM framework.

Semantics

- **Scale**: `beta(x)` lives on the node's latent (log-odds) scale — added for a continuous node (`z = h(y) + …`), subtracted from the cutpoints for an ordinal node, exactly like an `LS` weight. With no modifiers it *is* `LS(on)` (identical model, testably bit-exact), so `VC` vs `LS` is a nested question.
- **Penalty**: the fitting objective is the penalized likelihood Σ_i `NLL`$_i$ `+ penalty · ‖b_theta weights‖²` (total-NLL scale — a fixed Gaussian prior whose shrinkage vanishes as n grows; `beta0` is never penalized). `penalty → ∞` shrinks `b_theta` to the zero function and recovers the classical `LS` fit. Default `penalty=1.0`; raise it when modifiers are many or n is small.
- **Identification / centering**: a constant moves freely between `beta0` and `b_theta`. The head's output layer is zero-initialised (`beta(x) = beta0` at step 0), and after `fit` the head is re-centered to mean zero over the training data (function-preserving), so `beta0` is the training-population main effect — the `Colr` reading when `beta` is constant.

- **Warm start**: `fit(vc_warm_start=True)` (default) initialises `beta0` from the classical all-`ls` solution of the node's conditional (deterministic L-BFGS on a throwaway proxy) once per term, so training starts at the classical answer and only learns deviations.
- **Treatments**: `x_on` continuous or binary (2-level) ordinal (the term is linear in `x_on`; a binary ordinal enters as its 0/1 level, so `beta` is the identified level-1-vs-0 contrast). Multi-level ordinal treatments are a planned follow-up.
- **Read-out**: `flow.varying_coef(node, data, on=...)` evaluates `beta0 + b_theta(modifiers)` closed-form — deterministic, y-free, no abduction; for a binary treatment it equals the abduct-difference `u(x, t=1, y) − u(x, t=0, y)` identically (pinned by a test).

Propensity-centered VC: `center=True` (R-learner orthogonalization)

```
td.VC("T", "X2", "X3", penalty=1.0, center=True, center_folds=5)
# contributes  beta(x) * (t - e_hat(x))  to the shift
```

Robinson/R-learner centering inside the likelihood — Dandl et al. (2024) found treatment-centering to be *the* decisive ingredient for effect estimation under confounding in model-based forests, and it reproduces here: on a strongly confounded DGP whose prognostic part the model deliberately under-specifies (true g(x) quadratic, model linear), the uncentered β absorbs the confounded misfit (mean $|\beta - \tau| \approx 1.1$–$1.2$, effectively destroying the effect estimate) while the centered β stays near truth (≈ 0.1–0.3) — a **5–10× bias reduction** (`tests/test_vc_centered.py`). If the prognostic part is correctly specified, centering changes little; it is insurance against the misspecification you don't know you have.

The naive implementations are wrong, so this is a **two-stage frozen** design:

- **Training** uses **out-of-fold** ê: `center_folds` refits of the treatment node *only*, each predicting the fold it never saw (the DML cross-fitting requirement — in-sample ê reintroduces the own-observation bias and can be *worse* than no centering). The OOF values enter the outcome loss as frozen data, so **no gradient reaches the treatment node** from the outcome node (per-node factorization intact; pinned by a gradient-isolation test). Fold bookkeeping is exposed in `flow.vc_center_info` and pinned by tests. `center="colname"` supplies your own cross-fitted propensities instead.
- **Inference** (`log_prob` / `sample` / `abduct` / `pmf` / `scores`) recomputes ê from the flow's **own fitted treatment node** — the full-data fit (the standard DML train/predict split) — detached, from the

current parent values. Under `do(T=t)` the regressor is re-derived as `t − ê(x)`; nothing is ever cached.
- **Interpretation**: `beta0` is now the effect at the treatment margin (the observed propensities); `varying_coef` is unchanged (centering moves the regressor, not β). The LS-nesting reading applies to the uncentered term only. Requires a binary ordinal treatment (continuous-treatment centering E[T|x] is a follow-up). `center=False` (default) is bit-identical to the uncentered term.

Validation

`tramdag.simulations.VCLogisticShift` (`data/vc-shift/`, frozen contract) is a logistic-shift SCM with known `beta_true(x) = −1 + 0.8·X2 − 0.6·X3`, a nonlinear prognostic part, and confounded assignment (X2 is confounder *and* modifier). Acceptance (`tests/test_vc_term.py`): recovery corr ≥ 0.9 at n = 5000 (measured ≈ 0.99; min over 3 seeds 0.986), fitted `beta0` matches `fit_classical` under a large penalty, and the read-out identities. Candidate follow-ups (separate issues): propensity-centered `beta(x)·(t − ê(x))` (#30), per-observation scores for effect-modifier scans (#29).