tramdag documentation

Contents

tramdag: Interpretable Neural Causal Models (TRAM-DAGs) in PyTorch **3**
- Start here . . . 3
- Install . . . 4
- Documentation . . . 4
- Use of AI tools . . . 4
- Citation . . . 4

The model **4**
- The transformation function . . . 4
- The components . . . 5
- The components in code . . . 5
- Ordinal nodes . . . 6
- What the model cannot do . . . 7

Notation **7**
- Symbols . . . 7
- The model's symbols . . . 7

Reading a fitted TRAM-DAG **8**
- Which term owns which edge . . . 8
- A linear shift is a log-odds ratio . . . 8
- Ordinal nodes: cutpoints and the interventional PMF . . . 9
- Continuous nodes: density and one fitted shift . . . 9
- The price of a complex intercept . . . 9
- An effect that varies with a covariate . . . 9

Fitting a TRAM-DAG **9**
- One module, one sub-model per node . . . 10
- The likelihood . . . 10
- One node alone . . . 10
- Path A: stochastic optimization with `fit` . . . 10
- Path B: classical optimization with `fit_classical` . . . 12
- Memory and disk . . . 12

Per-observation scores and the effect-modifier scan **12**
- The idea . . . 12
- Reading the running sum . . . 13
- The statistic . . . 13
- Read it as screening . . . 14
- Using it . . . 14
- How the code computes the scores . . . 14

Varying-coefficient treatment effects **15**
- Why not a joint `CS` over treatment and modifiers . . . 15

Semantics . . . 15
Propensity centering: `propensity="col"` . . . 16
Validation . . . 16

Paper replication: protocol, hyperparameters and results, per experiment **17**
What is common to all eight . . . 17
Triangle, continuous (`triangle.py`): paper Sec. 6.1, App. C.3 . . . 19
Triangle, mixed (`triangle_mixed.py`): paper Sec. 6.2, App. C.4, App. B . . . 20
VACA / CNF benchmark (`vaca.py`): paper Sec. 5.1 and 5.2, App. C.1 . . . 21
CAREFL benchmark (`carefl.py`): paper Sec. 5.3, App. C.2 . . . 23
D4: the marginal start, measured per variant . . . 24
D5: the plateau rule per node . . . 25
Runtime and the CI deviations . . . 25
Repository choices the paper does not state . . . 26

Code map: every module of `src/tramdag/` **26**
`spec.py`: declare the model . . . 26
`transforms.py`: the monotone map h and the ordinal transform . . . 27
`flow.py`: the model . . . 27
`modules.py`: the term modules . . . 27
`nodes.py`: the node model . . . 28
`fitting.py`: `NodeFitMixin` and `FitMixin` . . . 28
`readouts.py`: `ReadoutsMixin` . . . 28
`scores.py`: effect-modifier detection . . . 28
`callbacks.py`: the shipped `fit` callbacks . . . 28
`plots.py`: the figures . . . 28
What is *not* in the package . . . 29
Decisions behind the training defaults . . . 29

Upstream candidates for zuko **29**
1. Analytic `call_and_ladj` for `BernsteinTransform` . . . 29
2. Learnable/linear tails for `MonotonicRQSTransform` . . . 29
3. Public inverse of `_constrain_theta` . . . 30
4. Docstring fix: the Bernstein θ-shape off-by-one . . . 30
5. `Logistic` in `zuko.distributions` . . . 30
Anti-candidates (look upstreamable, are not) . . . 30

Additive vs joint complex intercept: reading per-parent effects **30**
The data . . . 31
Fit both models . . . 32
`intercept_contributions`: the centered decomposition . . . 32
The structural difference: separable curves vs an entangled cloud . . . 33
Takeaway . . . 33

Classical fitting of all-`ls` TRAM-DAGs (`fit_classical`) **34**
0. The zeroth example: logistic regression on `MASS::birthwt` . . . 35
1. Continuous case . . . 39
1b. A bimodal DAG: the demo data . . . 43
2. Ordinal case . . . 45
3. Warm start: classical fit, then further training . . . 47

TRAM-DAG: one causal model, all three rungs **48**
1. The benchmark process . . . 49
2. The spec is the DAG, and one fit . . . 50
3. Rung 1: the observational fit . . . 52
4. Rung 2: interventions . . . 52

5. Rung 3: individual counterfactuals . . . 54
What this notebook checked . . . 54

Per-node or joint fit: a benchmark **55**
1. The process . . . 56
2. The arms . . . 56
3. The runs . . . 57
4. Convergence . . . 58
5. What the numbers say . . . 59
Which one, when . . . 60

Training strategies: driving `fit` from the API side **60**
1. What `fit` records without any callback . . . 61
2. Plain Adam . . . 62
3. Two phases . . . 62
4. `EarlyStopping`: keep the best weights . . . 63
5. `EarlyStopping(patience=)`: stop each node on its own . . . 63
6. Writing your own . . . 64
7. One node alone . . . 66
8. The scoreboard: a runtime comparison . . . 66
Which one, when . . . 67

Heterogeneous treatment effects: the `VC` term **67**
1. A DGP with a known effect function . . . 68
2. Which covariates modify the effect? (`effect_modifier_scan`) . . . 69
3. The spec: prognostic part and effect head are separate . . . 71
4. Reading the effect out . . . 72
5. The read-out is an identity, not a summary . . . 73
6. Confounding: why `propensity=` exists . . . 73
What to take away . . . 74

References **75**

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

TRAM-DAGs model each variable of a structural causal model with a monotone normalizing flow, the statistician's *transformation model*. The whole model is one triangular normalizing flow from iid standard-logistic latents to the observed variables. Its triangular structure is exactly your causal DAG. One fit on observational data answers all three rungs of Pearl's causal hierarchy: observational, interventional (the do-operator) and counterfactual (abduction). The model keeps **interpretable effects**: every linear-shift coefficient is a log-odds ratio, as in a classical proportional-odds model.

> Beate Sick & Oliver Dürr, *Interpretable Neural Causal Models with TRAM-DAGs*, CLeaR 2025 (arXiv:2503.16206). This repository is the reference implementation, in PyTorch on zuko. It replicates the paper's experiments under continuous integration.

Start here

The demo notebook is the introduction. It fits the paper's bimodal benchmark, walks the three rungs, and checks every answer against analytic ground truth. It runs on CPU in a few minutes, in Colab or locally.

Install

`uv add tramdag` or `pip install tramdag` installs the package from PyPI. The `plots` extra (`tramdag[plots]`) adds matplotlib for the figures. A clone with `uv sync` gives the development setup (`CONTRIBUTING.md`).

Documentation

The documentation site publishes the guides, the executed notebooks and the API reference.

- The model: `docs/model.md` and `docs/notation.md`
- Reading a fitted model: `docs/interpretation.md`
- Fitting: `docs/fitting.md`
- Treatment effects that vary with covariates: `docs/varying-coefficients.md` and `docs/scores.md`
- The design of the package: `docs/architecture.md` and `docs/code-map.md`
- The paper replications: `docs/paper-replication.md` and `experiments/README.md`
- The tests: `tests/README.md`
- The notebooks: `notebooks/README.md`

Use of AI tools

LLM coding assistants helped to develop this project. The authors reviewed all content and take responsibility for it.

Citation

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

```
@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},
  pages     = {606-630},
  year      = {2025},
  eprint    = {2503.16206},
  url       = {https://arxiv.org/abs/2503.16206},
}
```

The model

A TRAM-DAG is one triangular normalizing flow whose sparsity is a causal DAG. This page states the model in the notation of the paper and maps each part of it onto the constructors in `tramdag.spec`. For the symbols see notation.md; to read a fitted model see interpretation.md; for a worked fit see the demo notebook.

The transformation function

The variables have a causal ordering. For each variable a TRAM-DAG fits a monotone transformation function h. The function maps the observed value to a latent scale, conditional on the parents of that variable:

$$
\begin{aligned}
u_1 &= h(x_1) \\
u_2 &= h(x_2 \mid x_1) \\
u_3 &= h(x_3 \mid x_1, x_2) \\
u_p &= h(x_p \mid x_1, x_2, \dots, x_{p-1})
\end{aligned}
$$

This is the observed-to-latent direction. It is Eq. 2 of the paper, $F_{X|\text{pa}}(x) = F_U(h(x \mid \text{pa}))$, and in code it is `u = h(x) + shift`.

Training evaluates h directly to score the likelihood, which is cheap. Sampling runs the inverse $x_i = h^{-1}(u_i \mid \text{pa})$, which has no closed form: bracketed bisection solves it, so sampling costs more.

Together the h's form one triangular flow. Each variable depends only on its causal parents $\text{pa}(x_i)$, which is a subset of its predecessors. The Jacobian sparsity of the flow is therefore the DAG itself.

Each $h(x_i \mid \text{pa})$ is one transformation model of x_i given its parents. In code it is a `Node`, which also fits on its own, without the DAG (fitting.md).

The latents $u_1, \dots, u_p$ are standard logistic. That choice is what makes the fitted parameters interpretable, because a shift on the latent scale is then a log-odds ratio.

The components

The transformation decomposes additively on the latent scale, which is what keeps the interpretation valid. Each node's h is

$$
u_i \;=\; h(x_i \mid \text{pa}(x_i)) \;=\; \underbrace{h_\vartheta(x_i)}_{\text{intercept}} \;+\; \underbrace{\textstyle\sum_j \beta_{ij}\, x_j}_{\text{linear shifts}} \;+\; \underbrace{\textstyle\sum_k g_{ik}(x_k)}_{\text{complex shifts}} \;+\; \underbrace{(\beta_0 + b_\Theta(x_{\text{mod}}))\, x_t}_{\text{varying coefficient}},
$$

and every causal parent enters through exactly one term. Take x_5 with parents $\text{pa}(x_5) = \{x_1, x_2, x_4\}$ as the example.

- **Simple intercept, SI.** $h_\vartheta(x_5)$ has constant parameters. It is a flexible monotone baseline, a Bernstein polynomial by default, and it is the same for every observation.
- **Complex intercept, CI.** The parameters ϑ are themselves a function of some parents. The whole transformation bends with the parent, which permits interactions that an additive shift cannot express.
- **Linear shift, LS.** $\beta_{51}x_1 + \beta_{52}x_2$. One interpretable number per parent.
- **Complex shift, CS.** $g(x_4)$, an unrestricted network of the parent. It stays additive on the latent scale.
- **Varying coefficient, VC.** $(\beta_0 + b_\Theta(x_{\text{mod}}))\, x_t$: a treatment x_t whose effect β varies with modifier covariates through a small penalized network. varying-coefficients.md is its guide.

Sampling has to invert only the intercept, because the shifts move to the other side:

$$
x_5 = h^{-1}(u_5 \mid x_1, x_2, x_4) = h_\vartheta^{-1}\Big(u_5 - \underbrace{\beta_{51}x_1 + \beta_{52}x_2}_{\text{LS}} - \underbrace{g(x_4)}_{\text{CS}}\Big).
$$

The components in code

Each node declares its transformation as an additive formula of terms. The formula is the node's first argument, a `+` sum of terms. Each constructor names the parents its term depends on. The paper's symbols `I`, `LS`, `CS`, `VC` are the classes `Intercept`, `LinearShift`, `ComplexShift`, `VaryingCoefficient`.

formula for a continuous node x_3	$u_3 = h(x_3 \mid \text{pa})$
nothing, or `I()`	$h_\vartheta(x_3)$, the simple intercept
`LS("X1")`	$h_\vartheta(x_3) + \beta x_1$
`I("X1")`	$h_{\vartheta(x_1)}(x_3)$, the complex intercept
`CS("X1")`	$h_\vartheta(x_3) + g_1(x_1)$
`LS("X1") + CS("X2")`	$h_\vartheta(x_3) + \beta x_1 + g_2(x_2)$
`CS("X1", "X2")`	$h_\vartheta(x_3) + g_{12}(x_1, x_2)$, one joint network
`CS("X1") + CS("X2")`	$h_\vartheta(x_3) + g_1(x_1) + g_2(x_2)$, two additive networks
`I("X1", "X2")`	$h_{\vartheta(x_1, x_2)}(x_3)$, one joint network
`I("X1", "X2", allow_interaction=False)`	$h_{\vartheta(x_1)+\vartheta(x_2)}(x_3)$, one network per parent, summed in coefficient space
`CS("X1") + VC("X2", t="T")`	$h_\vartheta(x_3) + g_1(x_1) + \beta(x_2)\, x_T$

Three rules follow from the model.

- **Exactly one intercept, first.** A formula written without one gets `I()` prepended. `I()` without parents is the simple intercept, `I(...)` with parents the complex one; `SI()` and `CI(...)` spell that out with the arity checked.
- **Joint versus additive is argument grouping.** Several parents in one term form one network over all of them, an interaction. The same parents in separate terms act additively. A node takes at most one intercept with parents, so for the intercept `allow_interaction` sets the grouping.
- **Every parent enters through exactly one edge-owning term.** The VC modifiers are the one exception: `CS("X2") + VC("X2", t="T")` is the intended pattern. There x_2 acts prognostically through the shift and also modifies the treatment effect.

The three knobs on a term

`transform=` on `I` picks the class of h_ϑ for a continuous node:

- `"bernstein"`, the default: a Bernstein polynomial whose tails extrapolate along the boundary slope;
- `"spline"`: a monotone rational-quadratic spline, whose tail slope has a fixed value (zuko-upstream.md explains the consequence);
- `"affine"`: location and scale only, so the node-conditional is a logistic GLM.

The keyword arguments of the transform class go in `transform_kwargs`, `I(transform="spline", transform_kwargs={"bins": 16})`. Each transform pre-scales the data from the training `range_q` and `$1 - $range_q` quantiles onto a fixed domain; `range_q=0` uses the minimum and maximum. Ordinal nodes have no transform to pick.

`input_transform=` on `I`, `CS`, `VC` transforms that term's continuous network inputs, with statistics frozen at the first fit: `"minmax"`, `"standardize"`, or a callable `fn(x, train)`. `LS` and the `VC` treatment stay raw, so their coefficients keep their units.

`units=`, `activation=`, `batch_norm=` on `I`, `CS`, `VC` size the term's network. The defaults are in the API pages of the terms.

Ordinal nodes

An ordinal node has no monotone transform to choose. Its intercept is the cutpoint vector of an ordered logit,

$$P(Y \le k \mid \text{pa}) = \sigma(\vartheta_k - s(\text{pa})),$$

with increasing cutpoints ϑ and s the total shift. The model subtracts the shift where a continuous node adds it. notation.md states the convention. interpretation.md shows what it does to the sign of a coefficient.

The probability of an observed level is the difference of two sigmoids. The code evaluates it in log space, which keeps a gradient where both sigmoids saturate (`ordinal_log_prob`).

Parents enter every network as features: a continuous parent raw, in one column; an ordinal parent one-hot, in one column per level. Abduction is exact for continuous nodes and truncated-logistic for ordinal ones, so `flow.sample(u=flow.abduct(df))` reproduces `df` exactly, level-exactly for ordinal nodes. A counterfactual takes three steps: abduction, action and prediction (Pearl, 2009). `abduct` gives the latents, `do=` sets the intervention, and `sample(u=..., do=...)` predicts.

What the model cannot do

- The model fixes the latent distribution to standard logistic. It does not learn it.
- A `VC` treatment is continuous or a binary ordinal node; a multi-level ordinal treatment is not supported. Propensity centering needs a binary ordinal treatment.
- The DAG is an input. `tramdag` fits the mechanisms of a graph you supply, and does not discover the graph.
- Identification is the usual causal one. No hidden confounding, and the graph has to be right.

Notation

This page defines the notation. Every guide, docstring and notebook in this repository follows it.

Symbols

Random variables are capital Latin letters, for example Y. Their realizations, observed or sampled values, are lowercase Latin letters, for example y. The cumulative distribution function of Y is F_Y and the density is f_Y. A hat marks an estimate: $\hat{F}$ is the fitted distribution, F the true and often unknown one.

Sets and vectors are bold, for example $\mathbf{X} = [X_1, \dots, X_J]$.

A single optimized parameter is a lowercase Greek letter, for example ϑ or β. A vector of parameters is bold lowercase, for example $\boldsymbol{\vartheta} = (\vartheta_1, \dots, \vartheta_M)^\top$. A tuple that collects several parameter groups, for example all weight matrices of a network, is bold capital, for example Θ.

A function that a parameter vector defines carries it as a subscript: $h_{\boldsymbol{\vartheta}}$ is the monotone transformation with coefficients $\boldsymbol{\vartheta}$.

The model's symbols

Symbol	Meaning	In code
X_j, Y	the variables of the DAG	node names in the spec
$\mathrm{pa}(X_j)$	the causal parents of X_j	`node_parents`
U_j	the standard-logistic latent of node j	`u = flow.abduct(df)`
h_{ϑ}	the monotone transformation of a node	the `I` term; `theta` in docstrings
ϑ_k	the ordinal cutpoints, elements of $\boldsymbol{\vartheta}$	`ordinal_cutpoints`
β	a linear-shift coefficient; e^{β} is an odds ratio	`LS`; `flow.ls_coefficients()`
g_j	a complex shift, an NN of one term's parents	`CS`

Symbol	Meaning	In code
β_0	the constant treatment effect of a VC term	`beta0`
b_Θ	the penalized effect-modification network	the VC net; `units=` sets its layers
λ	the L2 weight on b_Θ	`penalty=`
σ	the logistic function	`torch.sigmoid`

Every node's model is one additive formula on the latent scale. model.md writes it out. For a continuous node the shifts add. For an ordinal node the shift subtracts inside the sigmoid: $P(Y \le k \mid \text{pa}) = \sigma(\vartheta_k - \text{shift})$.

Reading a fitted TRAM-DAG

A TRAM-DAG is a normalizing flow, but its design keeps parts of it readable. This page collects the read-outs and says what each number means. For the model itself see model.md, and for the symbols see notation.md.

Code that runs checks the claims on this page. The links say where.

Which term owns which edge

`flow.to_matrix()` gives the labeled adjacency matrix of the model. Rows are parents and columns are children. A cell holds the term tag: `LS`, `CS`, `CI`, `VC` for a treatment edge, or `VCm` for a varying-coefficient modifier. An empty cell means no edge. A multi-parent term carries its parent group as a suffix, and several terms in one cell join with `+`.

This is the meta-adjacency view of the paper. It is also the fastest way to confirm that the model you wrote is the model you meant. `plot_dag(spec)` draws the same thing, with one edge style per term.

Exercised in the varying-coefficients notebook.

A linear shift is a log-odds ratio

`flow.ls_coefficients()` gives `{node: {parent: weights}}` for the `LS` terms only. A `CS` or `VC` shift is a network and has no single weight, so `flow.ls_coefficients()` skips them.

Two things decide how to read a weight.

The sign follows the latent-scale convention of notation.md. A continuous node adds its shift, so a parent that raises the child gets a *negative* weight. An ordinal node subtracts its shift, so the sign is the one you expect there.

The scale is the latent scale, not the data scale. For a continuous node the weights carry the transform's scaling. A single weight is therefore not comparable to a regression coefficient. Their *ratio* is comparable, because the scaling cancels. An ordinal node has no such scaling, so $\exp(\beta)$ is an odds ratio directly.

An ordinal parent gives one weight per level, from its one-hot encoding. The data fix those weights only up to a common constant. The one-hot columns sum to one in every row. Adding a constant to all of them, and subtracting it from the intercept, leaves the likelihood unchanged. Read them as differences. `w[k] - w[0]` is the level-k-against-level-0 log-odds ratio. It matches the coefficient of column `{parent}[k]` in `flow.design_matrix(df, node, drop_first=True)`, which drops the level-0 column.

Checked in `notebooks/classical_fit_tram_dag.py`, which compares the weights against `statsmodels` and R, and pinned in continuous integration by `experiments/triangle_mixed.py`, whose odds-ratio check is in paper-replication.md.

Ordinal nodes: cutpoints and the interventional PMF

An ordinal node's intercept is its cutpoint vector, not a transform (model.md). `tramdag.transforms.ordinal_cu` turns the unconstrained parameters into the increasing cutpoints, which is how you read a fitted baseline.

`flow.pmf(df, node, do=...)` gives the class probabilities per row, in closed form. It needs no sampling, and `do=` makes it interventional by overriding columns before the parents are read. With that combination, you can answer a treatment-effect question exactly, not by Monte Carlo.

Exercised in the classical-fitting notebook, and under `do=` in `experiments/triangle_mixed.py`. Every run checks that script against committed ground truth.

Continuous nodes: density and one fitted shift

`flow.density(df, node, grid, do=...)` is the continuous counterpart of `pmf`. For every row it evaluates $p(\text{node} = g \mid \text{pa})$ at each grid value, in closed form from the transform. Use it to look at a fitted conditional without sampling it.

`flow.shift_curve(node, parent, grid)` evaluates one fitted shift term along a grid of parent values. Use it to plot a `CS` term against the function that it has to learn. It goes through the term's own evaluation, so a custom term works too.

A caution on both: the data fix a `CS` term only up to a constant, because that constant can move into the intercept. Compare a fitted curve to the truth after mean-centering both, not point by point.

`density` runs in the classical-fitting notebook, section 1b. `experiments/triangle.py` compares the `shift_curve` of each `CS` variant with the true function against committed ground truth.

The price of a complex intercept

`I(...)` with parents is the flexible end of the family. The parents control the transform parameters, so the node can bend in ways no additive shift can express. The cost is that there is no single coefficient to read.

paper-replication.md measures that trade-off on the paper's triangle process, fitted once with an `LS` edge and once with a `CS` edge.

A middle option exists. `I("a", "b", allow_interaction=False)` builds one network per parent and sums their parameter vectors, so the parents act additively on the transform. `flow.intercept_contributions(df, node)` then decomposes that sum into mean-centered per-term parts, using the usual additive model convention, so the parts become comparable. This is a post-hoc read-out and changes nothing about the fitted model.

Worked through in the intercept notebook.

An effect that varies with a covariate

`VC(*modifiers, t=...)` fits an effect that varies with the modifiers. varying-coefficients.md is the guide. Before you declare a `VC` term, `flow.effect_modifier_scan` shortlists the modifiers on a cheap all-`LS` fit (scores.md).

Fitting a TRAM-DAG

This page explains how to train a `CausalFlowDAG` and its nodes. It covers the likelihood, the Adam paths `fit` and `fit_jointly`, the classical path `fit_classical`, and which recipe to use. The API pages hold the signatures and defaults. `notebooks/training_strategies.py` runs the

Adam recipes end to end and compares their runtime. `notebooks/fit_strategies_benchmark.py` compares `fit`, `fit` with `n_jobs` and `fit_jointly`.

One module, one sub-model per node

A `CausalFlowDAG` is a single `torch.nn.Module` with one independent sub-model per variable. A sub-model has an intercept that produces the transform parameters θ and one shift module per shift term. It also has the monotone transform h, which holds range buffers but no weights. The nodes share no parameters. The DAG lives entirely in which parents each node reads: there is no edge weight matrix and no shared trunk.

The likelihood

The flow maps iid standard-logistic latents U to the observed X in causal order, and node i reads its parents as data. The Jacobian is therefore triangular, and the joint log-likelihood decomposes per node, $\log p(x) = \sum_i \log p(x_i \mid \mathrm{pa}(x_i))$.

- **Continuous node.** Change of variables through the monotone transform: $u = h(x;\theta) + s$ and $\log p(x \mid \mathrm{pa}) = \log f_{\text{logistic}}(u) + \log |h'(x)|$.
- **Ordinal node.** The ordered-logit head of model.md, the log of the cutpoint-interval probability.

Because parents enter as data, the per-node terms have disjoint parameters, and maximizing each term maximizes the sum. So every node can fit on its own: `fit` minimizes each node's mean negative log-likelihood, plus the node's `VC` penalty, one node after the other. `fit_jointly` minimizes the sum of the same losses in one loop.

One node alone

A `Node` is a TRAM regression of one variable on its parents, and it needs no DAG. It fits, scores and saves like the flow, for one node. `notebooks/training_strategies.py` fits one with Adam, and `notebooks/classical_fit_tram_dag.py` fits one classically against R `glm`. A lone node built with `seed=` starts from the weights it has in a flow with that seed and the default `init` (code-map.md). A fit with the flow's fit `seed=` then gives the weights of that node in the flow.

Path A: stochastic optimization with `fit`

`fit` is the general-purpose trainer. With `fit_jointly`, it is one of the two Adam paths, the only trainers for a model with a `CS`, `CI` or `VC` term (`fit_classical` is not). Per node, it is one minibatch Adam loop that keeps the final weights unless a callback restores others. The `fit` docstring lists every option.

- **Nodes.** The nodes fit in topological order, or in parallel worker processes with `n_jobs`.
- **Epochs.** `epochs` has no default. A fixed budget under-spends on one workload and wastes on the next, so every caller states its own.
- **Per-node arguments.** `epochs`, `learning_rate`, `optimizer` and `callbacks` take one value for every node or a dict by node name. A missing name gets the default; for `epochs` it raises. A `collections.defaultdict` gives its own fallback.
- **Shuffle.** Every node shuffles its rows from the fit's `seed=`. So a node fits as `Node.fit` with that seed, and `fit_jointly` sees the same row order.
- **Calibration.** The first fit calibrates each node on the training frame. Every term freezes its data-dependent state there: the intercept its `range_q` quantiles and each `input_transform=` its statistics. A loaded checkpoint is never recalibrated.
- **Validation.** With `validation_data=` or `validation_split=`, `fit` records each node's validation NLL after every epoch. `validation_split` takes the last rows unshuffled, so shuffle the frame first if its row order means anything.

- **Callbacks.** A callback gets the node (under `fit_jointly`, the flow) at fit begin, after every epoch and at fit end. A `True` return after an epoch stops that node. The shipped callback is `EarlyStopping`, which restores the best-validation weights. A learning-rate scheduler is a few lines of callback around torch's `lr_scheduler`.
- **Centered `VC` propensities** ride the training frame as a column (varying-coefficients.md).

The marginal start

`fit(marginal_init=True)` puts the optimizer near the data before the first step. It changes the parameters the optimizer starts from, not the model or the loss. It sets the ϑ of each Bernstein or ordinal simple intercept so that, at shift $s = 0$, the node's CDF equals the empirical marginal $\widehat{F}$ of its training column. The match is exact for an ordinal node and a smooth approximation for a continuous node. An intercept with parents has no single ϑ and keeps its seeded initialization. Without the flag, every simple intercept starts with all its parameters at zero.

The latent U is standard logistic, so at $s = 0$ the model says $P(X \leq x) = \sigma(h(x))$. The start therefore targets $h = \text{logit}\,\widehat{F}$.

- An ordinal node targets the cumulative class log-odds and inverts `ordinal_cutpoints` (`ordinal_marginal_init_theta`).
- A continuous node sets the Bernstein control points to logit $\widehat{F}$ at equally spaced points of the calibrated range. Then it inverts the zuko parameterization (`BernsteinUT.marginal_init_theta`). The polynomial smooths $\widehat{F}$, and a sharp feature such as the dip between two modes smooths most.

The spline and affine transforms have no marginal start, and the flag skips them. A `range_q=0` Bernstein transform puts the range ends at the data extremes, where the logit is infinite, so `fit` refuses the flag there.

The zero start is close to a straight line with a steeper h than the data. It gives a narrower distribution and a higher NLL at shift zero. The likelihood of an all-`LS` model has one optimum, so the start only shortens the way. A model with a network shift or a learning-rate anneal can reach a different local optimum. There the start can improve or worsen the causal estimate. D4 in paper-replication.md measures this per variant.

Which recipe

All-`LS` models train to the MLE and keep the final weights. Flexible models with a `CI`, `CS` or `VC` term overfit observational confounding at the MLE and need the best-validation weights to recover the causal effect.

recipe	when
`fit_classical`	all-`LS` spec: exact, deterministic, seconds
plain Adam	an all-`LS` spec whose `fit_classical` report is not converged, quick looks
multi-phase Adam, one `fit` call per rate	a tighter MLE without a scheduler
`EarlyStopping()`	any `CI`, `CS` or `VC` model; register it, `fit` has no default
`EarlyStopping(patience=)`	also stop each node once its best epoch is that old
`EarlyStopping(min_epochs=, smoothing=)`	ignore epochs before `min_epochs`; judge the best epoch on a moving average of the validation NLL, for small validation sets
a plateau `Callback` around torch's `ReduceLROnPlateau`	a decaying rate per node

recipe	when
`fit_jointly`	one schedule or stop rule on the summed NLL, as in the reference implementation

Three details are easy to get wrong.

- A second `fit` call continues training and `history` accumulates. That is what makes a multi-phase schedule a loop of `fit` calls.
- A callback instance passed to the flow's `fit` serves every node in turn, so its attributes describe the last node only. Use a dict by node name, or a `defaultdict`, to keep one instance per node. Under `n_jobs` the callbacks stay in the workers; read `node.history` instead.
- A post-fit `load_state_dict` skips the `VC` re-centering. Restore weights from `on_fit_end` instead, as `EarlyStopping` does.

Path B: classical optimization with `fit_classical`

`fit_classical` is the optimizer for all-`LS` models, where every node-conditional is an ordered logit or a transformation model with linear shifts. The `fit_classical` docstring holds the options and the report.

- **Full batch, float64, L-BFGS, then Newton.** There are no minibatches or schedules, so the same initialization gives bit-identical results. L-BFGS stops short of the optimum on an ill-conditioned NLL. One cause is a parent on a large raw scale next to a rare level. Newton steps on the exact Hessian then finish the fit.
- **The model stays float32.** Read-outs use the fitted values rounded by about 1e-7 relative. Compare the report's float64 `coefficients` with other software.
- **Unidentified directions do not move.** The `fit_classical` docstring names them and says how to compare them with other software.

A converged `fit_classical` leaves the model at the MLE. A `fit` call from there stays put, which makes it a warm start. `Node.warm_start` uses it for the `beta0` of a `VC` (varying-coefficients.md).

Memory and disk

Neither path writes to disk. The parameters, the optimizer state and the `history` dicts live in RAM. The only disk access is the explicit `save()` and `load()`. `fit(n_jobs=N)` forks `N` copy-on-write copies of the flow and the training frame.

Per-observation scores and the effect-modifier scan

A treatment effect is often not the same for every patient. A covariate that changes the effect is an effect modifier. A `VC` term models an effect $\beta(x)$ that changes with the modifiers, but you must tell it which covariates they are. The effect-modifier scan finds candidates for that list. It uses a simple model that takes seconds to fit, before you fit the flexible one. varying-coefficients.md covers the `VC` term itself.

The idea

1. **Fit a simple model.** It assumes one constant effect β for everyone. Every parent enters as a linear shift (`LS`), and `fit_classical` fits this all-`LS` model in seconds.
2. **Ask each row which way β should move.** The score of row i is $\psi_i = \partial \ell_i / \partial \beta$, the slope of that row's log-likelihood ℓ_i. A positive score means the row fits better with a larger β. A negative score means it fits better with a smaller one.

3. **Check that the answers cancel in every group.** At the fitted β the scores sum to about zero, because β is the best compromise. If the effect is really constant, the scores of any group of rows also sum to about zero, apart from random variation.
4. **Sort and add up.** Sort the rows by a candidate covariate, for example age. The sum of the first j scores is the score sum of the group "age up to the j-th value". So the running sum checks all these groups at once.

Plotted over the sorted rows, the running sum forms a path. Suppose young patients all ask for a larger β and old patients for a smaller one. The path then climbs first and falls back later. A large swing of the path along a covariate means that one constant β hides a pattern along that covariate.

Reading the running sum

Section 2 of `notebooks/varying_coefficients.py` draws the running sum for three covariates. The true effect changes with `X2` and `X3` and not with `X1`. The figure has two panels with the same effect. In the left panel `X1` has a quadratic effect on the outcome, and in the right panel a linear one.

The figure plots the running sum after a scaling, and the grey band spans ± 1.358 on that scale (The statistic explains both). If a path leaves the band, the scan flags its covariate.

- The paths of `X2` and `X3` swing far out of the band and come back. They flag in both panels.
- In the right panel the path of `X1` stays in the band. The simple model describes the linear effect of `X1` correctly, and `X1` does not flag.
- In the left panel the path of `X1` swings out as well. The simple model cannot describe the quadratic effect, and `X1` flags although it does not change the treatment effect. The section Read it as screening explains why.

The statistic

The scan is the structural-change test (Zeileis & Hornik, 2007) behind model-based recursive partitioning (Dandl et al., 2024; Zeileis et al., 2008). That method is a tree that splits the data where the coefficients of a model change. The scan applies its test to the treatment coefficient of a TRAM-DAG.

For one candidate, the scan sorts the n rows by that candidate. It then forms the scaled running sum, also called a CUSUM (cumulative sum):

$$B_j = \frac{1}{\mathrm{sd}(\psi)\sqrt{n}} \sum_{i \le j} \psi_{(i)}, \qquad j = 1, \dots, n,$$

with $\psi_{(i)}$ the score of the i-th row in the sorted order. The statistic `stat` is $\max_j |B_j|$, the largest distance of the path from zero.

- The path ends near 0, because the scores sum to about zero. It starts near 0, because one score divided by $\sqrt{n}$ is small.
- The scaling by $\mathrm{sd}(\psi)\sqrt{n}$ makes the distribution of `stat` approximately independent of the sample size and of the size of the scores.
- With a constant effect, B behaves approximately like a Brownian bridge: a random path that starts and ends at 0.
- The largest distance of a Brownian bridge from zero has a known distribution, the Kolmogorov distribution. It gives the `p_value` without simulation.
- A `stat` above 1.358, the 5 % critical value, sets `flag`.

The p-value is approximate. The scan uses the raw treatment score. The full test (Zeileis & Hornik, 2007) removes its correlation with the scores of the other fitted coefficients, and the scan does not.

The error has a direction. For a candidate that is also an `LS` parent of the node, the fitted `LS` weight already removes part of the pattern. The scan then flags too rarely for that candidate. Read the p-value as a guide, not as an exact error rate.

Read it as screening

A flag means that one constant β does not fit all rows along the covariate. Two causes give this:

- The effect really changes with the covariate.
- The simple model describes the covariate's own effect on the outcome, the prognostic part, wrongly.

The second cause works through the score. The treatment score of row i is $\psi_i = r_i$ times the term's regressor: t_i, or $t_i - \hat{e}_i$ for a centered `VC`. Here t_i is the treatment of the row. The factor $r_i = \partial \ell_i / \partial s_i$ is the derivative of ℓ_i with respect to the row's total shift s_i, and it acts like a residual. If the simple model gets the prognostic effect of `X1` wrong, r_i follows a pattern in `X1`. The treatment score takes on that pattern, so the path of `X1` swings.

So a flag means "look here". What you find is a modifier or a misspecification, and both are worth knowing.

Two cases make the p-value less reliable:

- **Few distinct values.** A binary or three-level candidate gives many tied rows. Inside a tie the order is arbitrary, and the path depends on it. Read the p-value as a ranking, not as an exact test.
- **No converged fit.** The scores sum to zero only at the optimum. Scan an all-`LS` model whose `fit_classical` report says `converged`. The scan also takes the β_0 of a `VC` term, but a `VC` model stopped at its best-validation weights is not at the optimum. Read such a scan with more care.

Using it

`flow.effect_modifier_scan(df, node, t=)` runs the scan for the treatment `t` of `node` and returns one row per candidate, sorted by `stat`. By default it scans every node except `node`, `t` and the descendants of `t`. A modifier comes before the treatment, and a descendant would pick up the effect itself. A candidate does not have to be a parent. For a binary ordinal treatment the scan tests the level-1-against-0 contrast, and for a `VC` treatment its β_0.

`flow.scores(df, node)` returns the scores themselves, one row per row of `df` and one column per interpretable shift coefficient of the node. A `CS` term has no single coefficient and so no column. The scores also serve influence analyses and robust standard errors. The `effect_modifier_scan` and `node_scores` docstrings give the options and the column names.

How the code computes the scores

Every shift coefficient γ adds γz to the node's total shift s. Here z is the value of a parent, or one column of its one-hot encoding. The chain rule therefore gives

$$\frac{\partial \ell_i}{\partial \gamma} = \frac{\partial \ell_i}{\partial s_i} z_i = r_i \, z_i,$$

so one derivative r_i per row serves every coefficient. The derivative has a closed form for both node kinds, and the computation uses no autograd.

Here h is the fitted monotone transformation and σ the logistic function.

For a continuous node with value y, the latent value is $u = h(y) + s$ and $\ell = \log f(u) + \log h'(y)$, with f the standard-logistic density. Then

$$\frac{\partial \ell}{\partial s} = 1 - 2\sigma(u).$$

For an ordinal node, $P(Y = y) = \sigma(b) - \sigma(a)$ with the shifted cutpoints $a = \vartheta_{y-1} - s$ and $b = \vartheta_y - s$. Then

$$\frac{\partial \ell}{\partial s} = \frac{\sigma'(a) - \sigma'(b)}{\sigma(b) - \sigma(a)}, \qquad \sigma' = \sigma(1 - \sigma).$$

For the first and last class, the outer cutpoint is $\mp\infty$ and its σ' is zero.

The function only reads the fitted model. It changes neither fitting nor sampling. The tests check that each column sums to about zero at a fitted optimum, and that the scores agree with float64 finite differences.

Varying-coefficient treatment effects

A `VC` term gives a node a treatment-effect head with its own bias-variance budget. It contributes $\beta(x)\, x_t$ with $\beta(x) = \beta_0 + b_\Theta(x)$ to the node's shift, where b_Θ is a deliberately small, penalized network of the modifiers. The point is not expressiveness. The point is that the flow estimates the effect function with care, not as a by-product. Every claim here runs in `notebooks/varying_coefficients.py`, which is the one place for the code, or in the tests that the text names.

Modifiers may appear twice in a node, as prognostic parents through `CS` or `LS` and as effect modifiers through `VC`. The design intends this pattern. Only the treatment named by `t=` owns its edge, and a second term that declares it raises an error.

Why not a joint `CS` over treatment and modifiers

For a binary treatment the multi-parent `CS` is equivalent in expressiveness. Any shift decomposes as $s(x,t) = s(x,0) + [s(x,1) - s(x,0)]\, t$. But the `CS` form has no effect-specific regularization. The likelihood rewards a good average fit of $s(x,t)$, and nothing rewards a smooth difference between the arms. So the read-out is the difference of two jointly fitted, unregularized networks. It amplifies random variation. On the `vc_hetero` process the `VC` term clears the recovery bar of the tests (see Validation). The `CS` reduced form stays below a correlation of 0.8 on the same protocol, although the model is in-class (`test_cs_reduced_form_falls_short_of_the_bar`). Causal forests and R-learners do not work because they target the effect. They work because they regularize it (Athey et al., 2019; Nie & Wager, 2021). `VC` brings that ingredient into the TRAM framework.

Semantics

- **Scale.** $\beta(x)$ lives on the node's latent scale, like an `LS` weight. A continuous node adds it, and an ordinal node subtracts it from the cutpoints. With no modifiers, `VC(t="T")` is `LS("T")` bit-exactly, so `VC` against `LS` is a nested question.
- **Penalty.** The objective is $\sum_i \mathrm{NLL}_i + \lambda \|b_\Theta\|^2$ on the total-NLL scale, with `penalty=` as λ. It is a fixed Gaussian prior whose shrinkage vanishes as n grows, it never applies to β_0, and $\lambda \to \infty$ recovers the `LS` fit. Raise it when the modifiers are many or n is small.

- **Identification.** A constant moves freely between β_0 and b_Θ. The head's output layer is zero-initialized, so $\beta(x) = \beta_0$ at step 0. After `fit` each node re-centers its head to mean zero over the training data, which preserves the function. β_0 is therefore the main effect in the training population, the Colr reading when β is constant.
- **Warm start.** `flow.nodes[name].warm_start(train_df)` fits the all-`LS` version of the node classically and copies the treatment weight into `beta0`. Call it before `fit`. Training then starts at the classical answer and learns only deviations. `flow.warm_start(train_df)` does this for every node with a `VC` term and returns the reports by node name.
- **Treatments.** Continuous or binary ordinal, entering as its 0/1 level so that β is the identified level-1-against-0 contrast. The term is linear in the treatment.
- **Read-out.** `flow.varying_coef(df, node)` evaluates $\beta(x)$ in closed form, deterministic and free of the outcome. `tramdag.plots.plot_varying_coef(flow, df, node, by=)` draws it along one modifier. For a continuous outcome, $\beta(x) = \text{logit}\, P(Y \le y \mid x, do(T = 1)) - \text{logit}\, P(Y \le y \mid x, do(T = 0))$ at every y. This is an interventional quantity and needs no counterfactual. Because `abduct` inverts a continuous node exactly, $\beta(x)$ also equals the abduction difference $u(x, 1, y) - u(x, 0, y)$, which a test pins. For an ordinal outcome `abduct` samples the latent, so this identity does not apply.

Propensity centering: `propensity="col"`

With a `propensity=` column, the term contributes $\beta(x)\,(t - \hat{e}(x))$ instead of $\beta(x)\,t$. This is Robinson's R-learner orthogonalization (Nie & Wager, 2021; Robinson, 1988) inside the likelihood. In model-based forests, this ingredient is decisive for effect estimation under confounding (Dandl et al., 2024). That finding reproduces here on the `confounded` process, where the model deliberately under-specifies its prognostic part. The uncentered $\hat{\beta}$ absorbs the misfit, and centering brings it back near the truth. `tests/test_vc_centered.py` requires at least a factor of two. If the prognostic part is correctly specified, centering changes little.

Centering guards against a misspecified prognostic part. This also applies to randomized data: with $\hat{e} = 0.5$ the regressor is $\pm\frac{1}{2}$, and a misfit that both arms share cannot follow it. Centering does not guard against unmeasured confounding. The propensity uses only the observed parents of the treatment, so a hidden common cause of T and Y biases $\beta(x)$ with or without centering.

The design is two-stage and frozen, because the naive versions are wrong.

- **Training** uses out-of-fold propensities, one value per row. Compute them and pass them as the training-frame column that `propensity=` names. Any propensity model works if a fit that never saw a fold predicts that fold. This is the cross-fitting requirement of double machine learning (Chernozhukov et al., 2018). In-sample propensities reintroduce the own-observation bias and can be worse than no centering. The values enter the loss as frozen data, so no gradient reaches the treatment node and the per-node factorization stays intact. `fit` refuses a centered spec whose training frame lacks the column. The flow's validation NLL uses the live propensity of the fitted treatment node.
- **Inference**, in `log_prob`, `sample`, `abduct`, `pmf` and `scores`, recomputes $\hat{e}$ from the flow's own fitted treatment node on the current parent values, detached. Under `do(T=t)` the regressor becomes $t - \hat{e}(x)$. The flow caches nothing.
- **Interpretation.** β_0 is the mean of $\beta(x)$ over the training rows, with or without centering, because `fit` re-centers the head in both cases. `varying_coef` stays the same, because centering moves the regressor and not β. Centering requires a binary ordinal treatment.

Validation

The `vc_hetero` process in `tests/conftest.py` is a logistic-shift SCM with a known $\beta(x) = -1 + 0.8\,X_2 - 0.6\,X_3$. It has a nonlinear prognostic part and confounded assignment, and X_2 is confounder and modifier at once. In this configuration the `CS` reduced form fails hardest. `tests/test_vc_term.py` requires three things. It requires a recovery correlation of at least 0.9

with 4500 training rows. It requires a fitted β_0 that matches `fit_classical` under a large penalty. It requires the read-out identities. `tests/test_vc_centered.py` tests the centering claims on the `confounded` process. scores.md covers the scan that shortlists modifiers before you declare a `VC` term.

Paper replication: protocol, hyperparameters and results, per experiment

The eight variants under `experiments/` replicate the TRAM-DAG paper (Sick & Dürr, 2025) against its own R code (`tensorchiefs/tram-dag`). For each experiment this page lists the DGP, the model, every hyperparameter with its source, what deviates from the paper and why, and the numbers. The paper states five training numbers: n = 40000, 500 epochs, Adam, lr 0.001 and Bernstein `len_theta` 20 (order 19). It shows its results as figures. Where it gives no number, the "paper" column names the figure and what it shows. `experiments/ground_truth/*.json` holds the bounds and centers. The tables show the measurements of the pinned run. experiments/README.md explains the YAML variants, the frozen data and the check. The tables of D4, D5 and of the runtime section are study runs of their own. They are not the pinned run.

Seeds are DGP 42, init 7 and shuffle 0. The triangle scripts of the reference run unseeded. The comparison scripts seed R's RNG, which torch cannot replay. So every seed here is a repository choice.

What is common to all eight

item	reference	here
latent	standard logistic, shifts added on the continuous scale, subtracted for ordinal $P(Y \le k) = \sigma(\vartheta_k - s)$	same (notation.md)
Bernstein basis	`len_theta` unconstrained coefficients, `to_theta` softplus-cumsum, domain → [0, 1] with tangent-linear extrapolation outside; the domain comes from the train 5 %/95 % quantiles in the triangle scripts (`quantile(..., c(0.05, 0.95))`) and from min/max in the comparison scripts (`scale_df`)	zuko Bernstein, `n_coeffs` unconstrained (zuko ties two control points on, so `n_coeffs=20` is order 21 against the reference's order 19; the free-parameter count is what matches), domain = train `range_q`/1−`range_q` quantiles → [−5, 5], linear extrapolation. Triangle: `range_q: 0.05`, a match up to the reparametrization. VACA and CAREFL: `range_q: 0`, the reference's min/max

item	reference	here
networks	triangle scripts `create_param_net` with `hidden_features = c(2, 25, 25, 2)` continuous and `c(2, 2, 2, 2)` mixed, sigmoid (the ReLU line is commented out); the vector reads as in/out dims around the hidden stack, so the hidden layers are (25, 25) and (2, 2). Comparison scripts `make_model`: `dense(10, tanh) → dense(100, tanh) → dense(len_theta)`, one net per node	`units` and `activation` set per variant to exactly those stacks. The package defaults in code-map.md replicate the PyTorch reference `buehlpa/TramDag` instead and are not the paper's nets
init	triangle scripts: `LinearMasked` layers with Keras `random_normal` (N(0, 0.05^2)) on weights and biases, the LS `beta` layer included; comparison scripts: `layer_dense` default, glorot-uniform weights and zero biases	`init: normal` (triangle) and `init: glorot` (VACA/CAREFL) through `CausalFlowDAG(init=)`; torch's default init is the framework default, and under the full-batch protocol the init decides the fit (see VACA)
optimizer	Keras Adam, eps 1e-7	torch Adam, eps 1e-8; measured: no effect (VACA identical to four digits)
calibrated start	none	`marginal_init: true` for the two triangle LS models, `atan-cs` and `exp-cs`, deviation D4 below; off where it moves the endpoint (`linear-cs`, `sin-cs`) and impossible for the `range_q: 0` domain of VACA and CAREFL. Calibration without the flag never touches the weights
intercept output layer	Keras dense with bias	bias-free, deviation D3: the same function class, because the bias adds a constant to all unconstrained coefficients
plateau rule (VACA/CAREFL)	`update_learning_rate`: one optimizer, reduce when the summed validation NLL has not improved for 50 epochs (strict <), factor 0.1, min 1e-7	torch `ReduceLROn-Plateau(patience=49, threshold=0, threshold_mode="abs", factor=0.1, min_lr=1e-7)`, the same rule verified against torch's source, but per node on the node's own validation NLL, deviation D5 below; `experi-ments/helpers.py::fit_paper` drives it

The deviations have these labels:

- **D3** is the bias-free intercept output layer (the table above).
- **D4** is the marginal start (the section D4 below).
- **D5** is the plateau rule per node (the section D5 below).

Triangle, continuous (`triangle.py`): paper Sec. 6.1, App. C.3

DGP (`summerof24/triangle_structured_continous.R`):

- x1 ~ 0.5 N(0.25, 0.1^2) + 0.5 N(0.73, 0.05^2)
- h(x2 | x1) = 5 x2 + 2 x1 = u2
- h(x3 | x1, x2) = 0.63 x3 − 0.2 x1 − f(x2) = u3, u ~ logistic

f has three forms: `linear` −0.3 x, `atan` 0.75 atan(5 (x + 0.12)), `sin` 2 sin(3x) + x. True weights in the flow's convention: β12 = +2, β13 = −0.2, β23 = +0.3 (linear only). A fitted `CS` learns −f(x2) + const.

Model: `x1: SI`, `x2: SI + LS(x1)`, `x3: SI + LS(x1) + {LS(x2) | CS(x2)}`, Bernstein, the CS net hidden (25, 25) with sigmoid.

hyperparameter	paper / R code	here
train / validation	`dgp(40000)` / `dgp(40000)`, two draws	40000 / 40000, two draws
epochs	500	300 (`linear-cs`: 500), one continuous run; the CI deviation
lr	0.001 (`optimizer_adam()` default)	0.004, the CI deviation
batch	32 (Keras `fit()` default)	256, the CI deviation
`len_theta` / `n_coeffs`	20	20
schedule / early stop / init	none / none, final weights / random_normal	none / none / `init: normal`
coefficient read-out	after every epoch (Keras loop)	`fit(callbacks=)`, every epoch

The scripts follow the R code in these points. Each epoch shuffles one compiled model, so one continuous optimizer run fits it. No term has an `input_transform`, so the nets read raw parents. The Bernstein domain comes from the train 5 %/95 % quantiles. The fit keeps the final weights. The `linear-cs` variant is the misspecified case of App. C.3.3 (Fig. 17): the DGP effect is linear, and the complex shift must recover the straight line instead of bending to follow the sample. `atan-cs` is the headline complex-shift run (Fig. 7, 15, 16). `sin-cs` shows that the recovery is not specific to atan, and its grid covers x2 ∈ [−1, 0], the half period that Fig. 18 plots.

Results

variant	metric	paper	paper protocol, `init: normal` (batch 32 / lr 0.001 / 500 epochs)	CI config, pinned run (measured)
linear-ls	β12 / β13 / β23	Fig. 14 trajectories; Sec. 6.1 text: 1.98 / −0.21 / 0.26	1.987 / −0.170 / 0.282	1.981 / −0.161 / 0.281

variant	metric	paper	paper protocol, `init: normal` (batch 32 / lr 0.001 / 500 epochs)	CI config, pinned run (measured)
linear-cs	β13; max $\lvert\hat{g} - (-f)\rvert$ on $[-1, 1]$	Fig. 17: fitted CS is a straight line	−0.173; 0.122	−0.180; 0.023
atan-cs	β13; cs max err	Fig. 7 right / 15 / 16: CS on −f; text: β12 = 2.07, β13 = −0.203	−0.168; 0.059	−0.165; 0.081
sin-cs	β13; cs max err	Fig. 18: CS follows the non-monotone f on x2 ∈ [−1, 0]	−0.195; 1.10	−0.168; 0.245 on the paper's [−1, 0] window
all	|E[x3 | do(x1 = −1)] flow − DGP|	Figs. 16/17: histograms overlap	0.09 to 0.14	0.068 to 0.131 (linear-cs, atan-cs, sin-cs); linear-ls 0.107
all	val NLL x3	(none)	2.4607 to 2.4764	2.4606 to 2.4614

The pinned column gives the measurement of the pinned run, from its `metrics.json`. A `value` entry of a run metric in `experiments/ground_truth/*.json` is this measurement. The DGP, analytic and CAREFL entries are constants. Each `max` bound on an error measure sits at 1.5x to 4x of it. The `fit_seconds` bounds are 3x the time on the CI runner. No cell in this column is a bound.

β13 measures −0.16 to −0.18 instead of −0.2 in the continuous triangle variants. β13 multiplies x1, whose two mixture components sit at 0.25 and 0.73. So x1 has sd 0.254 against 0.375 for x2 and 2.918 to 3.748 for x3. At n = 40000, SE(β13) is about 0.040, the sd of β13 over 12 data seeds of `linear-ls`. The value is within about 1 SE of the truth. At n = 5000, β13 is too weakly identified for the frozen 5000-row CSVs to serve as fit checks. So the fit checks train on the paper's n = 40000 protocol.

The (25, 25) hidden stack is what lands `sin-cs` on Fig. 18, including its small −1-endpoint deviation. A literal `[2, 25, 25, 2]` reading puts a 2-sigmoid bottleneck on the input. It cannot reproduce the figure at any protocol (cs max err 1.22).

Triangle, mixed (`triangle_mixed.py`): paper Sec. 6.2, App. C.4, App. B

DGP (`triangle_structured_mixed.R`): x1 and x2 as above. x3 is ordinal with four levels and cutpoints θ = (−2, 0.42, 1.02) from the R code, which the paper does not state. The level is #{k : u3 > θ_k + 0.2 x1 + f(x2)}, with f either `linear` −0.3 x or `exp` 0.5 exp(x). The paper adds the ordinal shift and the flow subtracts it, so the fitted weights are β13 = −0.2, β23 = +0.3.

Model: `x1: SI`, `x2: SI + LS(x1)`, `x3: OrdinalNode(4, LS(x1) + {LS(x2) | CS(x2)})`, the CS net hidden (2, 2) with sigmoid.

hyperparameter	paper / R code	here
train / validation	`dgp(40000)` / `dgp(10000)`	40000 / 10000, two draws

hyperparameter	paper / R code	here
epochs, lr, batch, schedule, init	500, 0.001, 32, none, random_normal	350 (`exp-cs`) / 200 (`linear-ls`) continuous, none, `init: normal`; lr 0.004 (`linear-ls`: 0.002) / batch 256, the CI deviations
`n_coeffs` (x1, x2)	20	20
odds-ratio check (App. C.4)	odds(x2 ≤ −1) under do(x1 += 1), theory $e^2 \approx 7.39$	40000 rows, seed 99
counterfactual PMF (App. B)	Fig. 10 explains why point counterfactuals fail for the ordinal node	2000 rows × 200 draws

The scripts follow the R code as in the continuous triangle. They use one compiled model with shuffled batches and keep the final weights. They have no schedule and no `input_transform`, and the domain is the 5 %/95 % quantiles. The CS net uses hidden (2, 2) with sigmoid, read as in/out dims around the hidden stack like the continuous (25, 25). `exp-cs` pairs the ordinal node with a nonlinear f. `linear-ls` is the headline mixed case (Fig. 9, 19, and the C.4 odds-ratio check). It is the only mixed variant with a true β23.

Results

variant	metric	paper	paper protocol, `init: normal` (500 epochs)	CI config, pinned
linear-ls	β12 / β13 / β23	Fig. 19: 2 / 0.2 / −0.3 (paper convention; −0.2 / +0.3 in the flow's)	1.987 / −0.246 / 0.319	1.978 / −0.258 / 0.333
linear-ls	odds ratio predicted / DGP	$e^2 \approx 7.39$; C.4 text: 7.74, CI [7.16, 8.38]	7.29 / 7.19	7.23 / 7.19
linear-ls	CF PMF TV vs analytic; P(true level) flow / analytic / mode bound	(App. B qualitative)	0.044; 0.718 / 0.728 / 0.806	TV 0.050; 0.716 / 0.728 / 0.806
exp-cs	β13; cs max err	Fig. 20: distributions match	−0.207; 0.142	−0.205; 0.073
exp-cs	CF PMF TV; P(true level)	(none)	0.023; 0.917 / 0.921	0.021; 0.921 / 0.921

The pinned column follows the rule of the continuous table: every cell is a measurement.

VACA / CNF benchmark (`vaca.py`): paper Sec. 5.1 and 5.2, App. C.1

DGP (Sánchez-Martín et al., 2022, App. E.1): x1 ~ 0.5 N(−2, 1.5) + 0.5 N(1.5, 1). x2 = −x1 + N(0, 1). x3 = x1 + 0.25 x2 + N(0, 1). The error term is Gaussian, outside the logistic-latent family, and the all-`CI` flow must fit it. The analytic target is E[x3 | do(x2 = a)] = −0.25 + 0.25 a. The paper's Sec. 5.2 text says a ∈ {−3, −2, 0}. Its Fig. 5 panels and the R code

(`Figure_Triangle_Linear_Bimodal.R`) intervene at $a \in \{-3, -1, 0\}$, and this repository follows the code. do(x2 = −3) is off-manifold extrapolation and takes a looser tolerance.

Model: `x1: SI`, `x2: CI(x1)`, `x3: CI(x1, x2)`, Bernstein `n_coeffs = 31` (reference M = 30, `len_theta = 31`), the `make_model` nets.

hyperparameter	R code	here
train / validation	nTrain 2500 / `dgp(5000)`	2500 / 5000, two draws
epochs	10000 (`comparison/Figure_Triangle_Linear_Bimodal.R`: EPOCHS = 10000, M = 30, nTrain = 2500)	10000, one run, 1:1
lr	0.001	0.001
batch	full batch (one `apply_gradients` per epoch)	2500 = n_train
schedule	the plateau rule above	the same rule per node, deviation D5
input scaling	`scale_df`: everything min-max to [0, 1]	`input_transform: minmax` on the CI terms
Bernstein domain	train min/max (`scale_df`)	train min/max (`range_q: 0`), as in CAREFL
n_compare	(none)	50000

Results: the check is the flow's error against the analytic mean, not a pinned flow value.

metric	paper	here, pinned run (measured)
|E[x3 | do(x2 = −3)] − (−1.0)|	Fig. 5: densities overlap	0.102
|E[x3 | do(x2 = −1)] − (−0.5)|	Fig. 5	0.016
|E[x3 | do(x2 = 0)] − (−0.25)|	Fig. 5	0.040
sd(x1) flow vs analytic 2.0767	Fig. 4: bimodal x1 fitted; the default CNF (Javaloy et al., 2023) fails	2.064, error 0.013
val NLL x3	(none)	1.4341

The do(x2) errors vary with the init seed. D5 below gives five seeds. At do(x2 = −1) the per-node error goes from 0.006 to 0.018, and seed 1 gives the smallest. The paper shows one run's densities. The committed bounds sit at 1.5x to 4x of the pinned measurement.

The table below traces what each protocol ingredient is worth under the reference protocol with its global plateau, one change at a time. It uses the paper text's grid −3 / −2 / 0 and min-max inputs unless the row says otherwise.

variant	error	reading
raw parents into the nets	0.731 / 0.426 / 0.017	the tanh nets saturate: 40 % of the rows have |x1| > 2, 43 % |x2| > 2
global plateau, torch init	0.523 / 0.334 / 0.129	the summed NLL keeps improving through x1 while x3 overfits
… without any plateau	0.562 / 0.354 / 0.142	the rule is what stops the full-batch overfit
… Adam eps 1e-7	0.523 / 0.334 / 0.129	no effect

variant	error	reading
… Bernstein on train min/max	0.154 / 0.012 / 0.062	helps, not the cause
… glorot init	0.035 / 0.006 / 0.007	the cause (a different random draw than the config's seed 7)
… glorot + min/max	0.035 / 0.056 / 0.057	

Rejected alternatives:

- relu with raw parents holds the bound only at isolated epochs.
- tanh on raw parents saturates (first table row).
- Minibatch stepping leaves a common-mode offset in all three do-means, although the observational fit is perfect.

CAREFL benchmark (`carefl.py`): paper Sec. 5.3, App. C.2

This benchmark is the reference run 1:1 on the reference's own data, except D5 (the plateau per node). `carefl_fig5.r` sets `USE_EXTERNAL_DATA = TRUE` and trains on CAREFL's own committed 2500 rows (`X.csv`, x3/x4 sd-standardized by 6.0104/1.9114) with `val = train`. CAREFL's repository also commits the observation `xObs.csv`, the analytic truth curves and its own predictions on the grid `seq(-3, 2.9, 0.1)`. This repository freezes all of it under `experiments/data/carefl-cf`, external input with no generator, so the Fig. 6 curves are comparable point by point and every metric is in the reference's standardized units.

DGP (Khemakhem et al., 2021): x1, x2 ~ Laplace(0, 1/√2). x3 = x1 + 0.5 x2³ + ε, x4 = −x2 + 0.5 x1² + ε with ε ~ Laplace(0, 1/√2), x3/x4 divided by their sample sds. Counterfactuals are analytic by abduction of the error term. The observation is `xObs.csv` = (2, 1.5, 0.8465, −0.2616); the paper prints (2, 1.5, 0.81, −0.28), slightly off it. The queries are x3^cf | do(x2 = α) and x4^cf | do(x1 = α) on the committed grid.

Model: `x1, x2: SI`, `x3, x4: CI(x1, x2)`, Bernstein `n_coeffs = 31`, the same `make_model` nets as VACA, `range_q: 0` for the reference's `scale_df` min-max Bernstein domain.

hyperparameter	R code (`carefl_fig5.r`)	here
train / validation	CAREFL's own `X.csv`, sd-standardized, `val = train`	the same committed `X.csv`, `val = train`; a fresh standardized draw is scored, never trained or annealed on
epochs, lr	7000 @ 0.001	7000 @ 0.001
batch, schedule, input scaling, init	full batch, plateau 0.1/50/1e-7, `scale_df`, glorot	same, the plateau per node (D5)
Bernstein domain	train min/max (`scale_df`)	train min/max (`range_q: 0`)
scoring	the single `x_obs`, curves over α (Fig. 6)	the same, plus 300 held-out rows at α ∈ {−1.5, 0, 1.5}, all in standardized units

Results, in standardized units.

metric	paper / CAREFL	here, pinned run (measured)
Fig. 6 max |x3^cf error|	Fig. 6: flow tracks the DGP curve; CAREFL's committed x3 preds err up to 1.21 at α = −3	0.049

metric	paper / CAREFL	here, pinned run (measured)
Fig. 6 max |x4^cf error|	CAREFL's committed x4 preds: max 0.174	0.156, at α = 0.1 (0.156 at α = 0 too); 0.148 at the α = −3 grid edge
held-out CF MAE x3, α = −1.5 / 0 / 1.5	(none)	0.014 / 0.009 / 0.013
held-out CF MAE x4, α = −1.5 / 0 / 1.5	(none)	0.059 / 0.025 / 0.052

Two ingredients carry the agreement. One study run per ingredient changes only that ingredient. Training on a fresh 2500-row draw instead of the committed rows raises the Fig. 6 error. It is 0.21 for x4 and 0.11 for x3. The 5 %/95 % domain instead of min-max raises it too, to 0.27 for x4 and 0.29 for x3. Both maxima sit at the α = −3 grid edge, where min-max gives 0.15 and 0.04. The reference protocol gives 0.16 for x4 and 0.05 for x3.

The largest x4 error of the reference run sits at the bottom of the parabola, α = 0 to 0.1. There the flow lies above the truth. The fresh-draw run lies above the truth at α = 0 too, by 0.12 for x4.

Rejected alternatives:

- Raw parents saturate the sigmoid on the Laplace parents, as tanh does.
- relu underfits x3 and grows a fragile Bernstein tail. One held-out row inverts to x4 ≈ 580.
- Minibatch underweights the sparse x2 tail where the Fig. 6 grid ends. The held-out x3 error at do(x2 = 1.5) and the Fig. 6 x3 error both grow.

D4: the marginal start, measured per variant

`marginal_init` starts every simple intercept at its column's empirical marginal (fitting.md). The reference starts `bernp$beta` at zero. Measured at the CI protocol, one run per variant with the start off and on, seeds unchanged:

variant	epochs to within 0.01 of the final train NLL, off → on	endpoint, off → on	verdict
triangle linear-ls	28 → 5	identical (val NLL 2.4606, do(x1) err 0.107)	on
triangle linear-cs	28 → 5	cs max err 0.023 → 0.046; do(x1) err 0.131 → 0.084	off: the misspecified line is what this variant measures, and the start bends it
triangle atan-cs	28 → 6	cs max err 0.099 → 0.081; do(x1) err 0.080 → 0.068	on
triangle sin-cs	35 → 18	cs max err 0.245 → 0.254; do(x1) err 0.078 → 0.102	off: the start gains no accuracy and does not help the curve
mixed linear-ls	42 → 9	identical (val NLL 1.2268, do(x1) err 0.046)	on
mixed exp-cs	28 → 6	cs max err 0.141 → 0.073; do(x1) err 0.018 → 0.013	on

variant	epochs to within 0.01 of the final train NLL, off → on	endpoint, off → on	verdict
VACA, CAREFL	(none)	(none)	impossible: `range_q: 0` has no marginal start

The start is only an initialization. Where the optimizer reaches the same basin, the endpoint is identical. Only the epoch count changes. Where the model has a network shift or the anneal decides the endpoint, the basin changes. Of the four such cases with a marginal start, two switch it on and two leave it off.

D5: the plateau rule per node

The reference reduces the learning rate of one optimizer when the summed validation NLL stops improving. The replication fits every node on its own with `fit`, so no summed NLL exists during its fit. Each node runs the same rule on its own validation NLL, with its own optimizer. The rule and its constants are those of the reference.

The table gives the VACA do(x2) errors at a = −3 / −1 / 0. It also gives the x3 validation NLL, with one run per init seed on the same data. The global column runs the reference rule through `fit_jointly`: one plateau on the summed validation NLL.

init seed	per node (here)	global (reference)
1	0.087 / 0.006 / 0.039, 1.4339	0.046 / 0.111 / 0.093, 1.4384
2	0.077 / 0.008 / 0.042, 1.4339	0.357 / 0.212 / 0.123, 1.4379
3	0.103 / 0.018 / 0.036, 1.4341	0.036 / 0.088 / 0.056, 1.4385
4	0.096 / 0.011 / 0.040, 1.4337	0.011 / 0.062 / 0.047, 1.4367
7 (config)	0.102 / 0.016 / 0.040, 1.4341	0.095 / 0.093 / 0.054, 1.4383

The per-node rule gives a lower x3 validation NLL on every seed. Its error at do(x2 = −1) and do(x2 = 0) is smaller on every seed. At the off-manifold point do(x2 = −3) the global rule errs less on four of the five seeds.

Runtime and the CI deviations

The reference protocol is too slow for the 2-core CI runners. The triangle runs 500 epochs × 1250 steps of batch 32, and the step is overhead-bound. The triangle scripts therefore use batch 256 at lr 0.004, with 8× fewer optimizer steps per epoch. The grid below, at the paper's 500 epochs with glorot init, selects them. Every entry is the cs max err unless the column says otherwise.

batch / lr / epochs	linear-ls β13	linear-cs	atan-cs	mixed exp-cs (TV)	steps vs paper
32 / 0.001 / 500 (paper)	−0.170	0.110	0.080	0.071 (0.019)	1×
128 / 0.001 / 500	−0.170	0.170	0.041	0.126 (0.023)	1/4
128 / 0.004 / 500	−0.178	0.160	0.094	0.070 (0.014)	1/4
128 / 0.004 / 250	−0.170	0.147	0.070	0.131 (0.035)	1/8

batch / lr / epochs	linear-ls β13	linear-cs	atan-cs	mixed exp-cs (TV)	steps vs paper
256 / 0.004 / 500 (CI batch and lr)	−0.171	0.132	0.073	0.072 (0.017)	1/8
256 / 0.008 / 500	−0.181	0.163	0.113	0.073 (0.017)	1/8
512 / 0.010 / 500	−0.169	0.183	0.104	0.119 (0.015)	1/16

Batch 256 / lr 0.004 is the only row that keeps every cs error within about 0.02 of the paper protocol. Larger batches or rates bend the misspecified linear-cs curve (0.160 to 0.183), and lr 0.008 hurts atan-cs.

The epoch floors, measured at that batch and rate:

- **Triangle** holds every bound at 300 epochs. At 150 the run fails atan's do(x1) bound (0.211 > 0.170). `linear-cs` trains 500, because at 300 its fitted cs curve flattens past |x2| > 0.5. The DGP line spans only ±0.3. relu is out: it fits atan badly.
- **Triangle-mixed** `exp-cs` has two slow parts, the 2-unit sigmoid CS net and the do(x1) mean, so it trains 350 epochs. `linear-ls` has no CS net to wait for, and at lr 0.002 it trains 200 epochs. The lower lr 0.002 lands closer to the 500-epoch values. relu at the sd-0.05 normal init fails here too.
- **VACA and CAREFL** run their references 1:1 except D5; the rejected shortcuts are in their sections.

Repository choices the paper does not state

- the seeds (the reference seeds R's RNG with 42, which torch cannot replay; a full-batch run ignores the shuffle seed),
- the validation draw sizes for the triangle scripts (the R code's `test`),
- `n_compare`, and `n_scatter` = 2000 points per VACA scatter panel (Fig. 4),
- `n_heldout = 300` and the α set {−1.5, 0, 1.5}, scored next to the single `x_obs`, drawn at the seed offset 999 from `dgp_seed`,
- the mixed cutpoints from the R code,
- the odds-ratio sample (40000 rows),
- the counterfactual-PMF sample (2000 rows × 200 draws).

Code map: every module of `src/tramdag/`

This page lists each module with its role and the decisions behind it. The API pages document every signature and default. Names in parentheses are private machinery and are not part of the API.

`spec.py`: declare the model

Every term is a `Term` subclass under its pythonic name. The paper's symbol is the same object, so `LS is LinearShift`. A spec is plain data: it holds no tensors, so it serializes and compares without torch.

Name	Role
`Term`	One additive term of a node's transformation. `+` on terms builds plain lists. The contract for a custom term is in architecture.md.
`SI()` / `CI()` / `Intercept`	`Intercept` (`I`) is the one intercept class. `SI` and `CI` are its two spellings with the arity checked.
`LinearShift` / `ComplexShift` / `VaryingCoefficient`	The three shifts: an interpretable coefficient, a network, and the penalized treatment-effect head (varying-coefficients.md).
`ContinuousNode` / `OrdinalNode`	The two node kinds.
`validate_and_sort()`	Edge-ownership validation plus Kahn topological sort. The order makes the flow triangular. Kahn's sort emits ready nodes in sorted batches, so the order is deterministic.
`spec_to_dict()` / `spec_from_dict()`	Checkpoint (de)serialization. A custom term writes its import path, so a checkpoint loads without a registry. `strict=True` refuses a term that omits a shaping option (`Term.acting_options`).

`transforms.py`: the monotone map h and the ordinal transform

Name	Role
`StandardLogistic`	The TRAM base distribution. zuko has none (zuko-upstream.md).
`BernsteinUT` / `SplineUT` / `AffineUT`	The three transforms of a continuous node (model.md).
`make_univariate_transform()`	Name to transform instance.
`ordinal_log_prob()` and the other `ordinal_*` functions	The ordinal likelihood and its inverses. The log probability is in log space (model.md).

`flow.py`: the model

`CausalFlowDAG` holds one `Node` per variable in topological order. Construction seeds each node's weights from the constructor seed and the node's name. A node starts from the same weights in every spec that contains it. `Node(seed=)` uses the same stream, so a lone node starts from the weights it has in the flow. Without a seed, the weights come from torch's global RNG. `nodes.seeded` holds this seeding. It runs in `torch.random.fork_rng`, so construction does not change torch's global RNG state. Each random act takes its own seed: the constructor for the weights, `fit(seed=)` for the shuffle of every node, `sample(seed=)` and `abduct(seed=)` for the latent draws. The class composes `FitMixin` and `ReadoutsMixin`. The modules below list their methods.

- `CausalFlowDAG.outcome()` gives the one node without children.
- `sample` covers observational, interventional (`do=`, graph mutilation) and counterfactual (`u=`) sampling. `abduct` recovers the latents. `pmf` and `density` give the analytic conditionals.
- `save` and `load` write checkpoints with the nodes' histories and provenance. `load` requires a complete checkpoint and fails loudly otherwise.
- The flow calibrates every node before the first fit and rolls the flags back when a node refuses the frame. A term names, validates and recomputes its own side columns through the `ShiftModule` hooks, and the flow fits the binary propensity.

`modules.py`: the term modules

There is one module class per term, and the term class holds it as `module` (`ComplexShift.module is ComplexShiftModule`). The module holds the term's network and owns the runtime hooks (architecture.md).

The default architectures replicate the PyTorch reference buehlpa/TramDag, file `tram_models.py`, so a fitted model stays comparable to it. They are not the paper's nets, which paper-replication.md lists per experiment.

Name	Role
`ShiftModule` / `InterceptModule`	The behavior hooks a term module owns.
`LinearShiftModule`	A linear layer without bias, because the intercept slot owns the constant. `.weight` is the interpretable coefficient.
`ComplexShiftModule` / `VaryingCoefficientModule`	The network shift and the varying-coefficient head.
`intercept_module()`	The one factory for the intercept slot: a free theta, one joint net, or one net per parent summed in coefficient space.

`nodes.py`: the node model

`Node` is one sub-model per variable, a TRAM regression on its parents. `tramdag.Node` exports it. It builds its terms through each term's `module`. The continuous-versus-ordinal branches live in its methods `log_prob`, `sample`, `abduct`, `marginal_theta` and `encode` (architecture.md). The shared helpers (`encode`, `schema_entry`, `tensorize`, the column checks and the checkpoint IO) serve both the node and the flow.

`fitting.py`: `NodeFitMixin` and `FitMixin`

`Node.fit()` and `Node.fit_classical()` fit one node, through `NodeFitMixin`. The flow's `fit()` and `fit_classical()` loop over the nodes, through `FitMixin`, serially or in `n_jobs` forked processes. `fit_jointly()` trains all nodes in one loop; it shares `_train_loop` with `Node.fit`. The mechanics are in fitting.md.

`readouts.py`: `ReadoutsMixin`

The read-out methods (`varying_coef`, `ls_coefficients`, `to_matrix`, `intercept_contributions`, `design_matrix`, `shift_curve`) live here once and are methods of the flow through the mixin.

`scores.py`: effect-modifier detection

`node_scores()` and `effect_modifier_scan()` are the analytic scores and the fluctuation scan (sc ores.md). The per-term score columns come from each term's `score_columns` hook.

`callbacks.py`: the shipped `fit` callbacks

`EarlyStopping` is the only shipped callback. Anything else is torch's `lr_scheduler` and a few lines of callback (fitting.md).

`plots.py`: the figures

matplotlib is optional (`tramdag[plots]`) and the first plot call imports it, never the package import. `plot_dag()`, `plot_marginals()`, `plot_training()` and `plot_varying_coef()` each take `ax=` to draw into a given layout. `plot_dag` draws every edge by the term that owns it.

What is *not* in the package

Three groups of files are research code:

- the SCM generators
- the frozen datasets
- the replication scripts

They live in `experiments/`, outside the installed package; `experiments/README.md` describes them and `tests/README.md` what the framework tests measure instead.

Decisions behind the training defaults

The signatures hold the defaults. This list keeps the reasons.

- **Weight initialization is `CausalFlowDAG(init=)`.** Initialization happens at construction, so the constructor seed makes the initial weights reproducible. `"torch"` is the `nn.Linear` default. `"glorot"` is the Keras `Dense` default, the paper's reference, and it is decisive under the paper's full-batch protocol. `zero_shift_networks=True` starts every `CS` shift at 0.
- **Network widths and `activation=`** follow the default classes of the PyTorch reference, for parity. The `VC` head has no counterpart there. `"sigmoid"` and `"tanh"` are the paper's activations.
- **`batch_norm=` is off**, because neither reference uses it. When on, it needs more than one row per batch, and inference needs `eval()` mode. `fit` and `load` leave the flow there.
- **The transform domain** is `transforms.BOUND`, a constant. The order of the Bernstein coefficients is in zuko-upstream.md.

Upstream candidates for zuko

This document lists what tramdag works around in zuko. The list ranks them by value against effort, as candidate upstream PRs or issues. tramdag uses zuko in one place only. It uses these three transforms from `zuko.transforms`:

- `BernsteinTransform`
- `MonotonicRQSTransform`
- `MonotonicAffineTransform`

The three wrappers in `src/tramdag/transforms.py` hold them. The flow machinery, the base distribution and the DAG structure are tramdag's own. The suggested order of attack is 4 → 1 → 3 → 2 → 5.

1. Analytic `call_and_ladj` for `BernsteinTransform`

`BernsteinTransform` inherits `MonotonicTransform.call_and_ladj`. That method gets the Jacobian by `torch.autograd.grad` in the forward pass. It therefore roughly doubles the training graph.

It also keeps the per-node loss out of `torch.compile`: autograd in the forward pass gives a double backward, which `aot_autograd` does not support.

The derivative has a closed form: `f'(x) = order·Δθ` against the order−1 basis. zuko already computes it in `_offset_and_slope` for the tail slopes. An override is therefore small and needs no API change. `MonotonicRQSTransform` already has such an override. The override removes the autograd call from the forward pass and makes the loss compilable.

2. Learnable/linear tails for `MonotonicRQSTransform`

zuko pads the knot derivatives with `exp(0)=1`. It extrapolates as the identity outside `[-B, B]`. Therefore the tail slope stays fixed at 1 for every θ, while the 5 % tails of the data sit beyond the

pre-scaling range.

Upstream, zuko can accept boundary derivatives of shape `(*, K+1)`, or it can take an opt-in `tails="linear"`. Identity tails are deliberate in the NSF design, so the change must stay backward-compatible.

3. Public inverse of `_constrain_theta`

`BernsteinUT.marginal_init_theta` inverts zuko's *private* `_constrain_theta` (cumsum-of-softplus) in closed form. It hard-codes two internal details of that function:

- the `log(2)·n/2` centering
- the tying of the first two and the last two diffs

Any upstream reparametrization therefore breaks the marginal start silently. A public `unconstrain_theta(theta)` that takes the target control points is small upstream. It removes the coupling to these private details. The framing is "identity/linear initialization support", a standard flow trick.

4. Docstring fix: the Bernstein θ-shape off-by-one

zuko's docstring says that θ has shape `(*, M-2)` for a degree-M polynomial. `n` unconstrained coefficients become `n+2` control points, which is degree `n+1`. The correct claim is therefore `(*, M-1)`. The paper's `len_theta=20` is order 19, not order 21.

5. `Logistic` in `zuko.distributions`

Neither zuko nor torch ships a Logistic distribution. tramdag therefore hand-rolls `StandardLogistic`, with `log_prob`, `sample` and `icdf`. Logistic latents are standard in TRAM-DAGs and in discrete flows, and zuko has a precedent in `GeneralizedNormal`. But zuko can point at `TransformedUniform(SigmoidTransform().inv)` instead. tramdag keeps its `_U_EPS` clamp locally in both cases. This candidate has the lowest value of the five.

Anti-candidates (look upstreamable, are not)

- **Quantile pre-scaling and `_ScaledUT`**: this is a modeling choice. zuko's `ComposedTransform` with `MonotonicAffineTransform` already composes it.
- **The ordinal ordered-logit transform and log-space `ordinal_log_prob`**: this transform is not a bijection. It stays outside zuko's flow scope. It is torch.distributions material, if anywhere.
- **LS, CS, CI and VC terms, the marginal-start policy, propensity centering and scores**: these are TRAM-DAG semantics. zuko's `MaskedMLP` already supports arbitrary adjacency. tramdag does not use it, because it needs per-term interpretability.
- **The spline `slope` knob and the Bernstein `eps` knob**: zuko already exposes both upstream.

Additive vs joint complex intercept: reading per-parent effects

A complex intercept groups its parents jointly, `CI("x1", "x2")`, or additively, `CI("x1", "x2", allow_interaction=False)`. `docs/model.md` states the rule, and `docs/interpretation.md` how to read the additive form with `flow.intercept_contributions(data, node)`.

This notebook fits both models on data with an interaction. The joint model fits better. Only the additive model splits into per-parent parts.

```
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd

from tramdag import CI, CausalFlowDAG, ContinuousNode
from tramdag.callbacks import EarlyStopping
```

The data

`x1, x2 ~ U(-2, 2)` and $x_3 = x_1 + x_2 + \gamma\, x_1 x_2 + \frac{1}{2}\,\varepsilon$, $\varepsilon \sim \text{Logistic}(0, 1)$. The term $\gamma\, x_1 x_2$ is an interaction of the two parents.

```
GAMMA = 0.6


def simulate(n, seed):
    rng = np.random.default_rng(seed)
    x1 = rng.uniform(-2, 2, n)
    x2 = rng.uniform(-2, 2, n)
    eps = rng.logistic(0, 1, n)
    x3 = x1 + x2 + GAMMA * x1 * x2 + 0.5 * eps
    return pd.DataFrame({"x1": x1, "x2": x2, "x3": x3})


train = simulate(4000, seed=1)
val = simulate(1000, seed=2)
train.head()
```

x1

x2

x3

0

0.047286

0.755745

1.239427

1

1.801855

0.126365

0.853817

2

-1.423362

0.970406

-0.294179

3

1.794598

1.590401

4.880817

4

-0.752674

-0.168717

-0.689205

Fit both models

Only the grouping of the two intercept parents differs.

```
def make_flow(joint: bool):
    spec = {
        "x1": ContinuousNode(),
        "x2": ContinuousNode(),
        "x3": ContinuousNode(
            CI("x1", "x2") if joint else CI("x1", "x2", allow_interaction=False)
        ),
    }
    return CausalFlowDAG(spec, seed=0)


flow_joint = make_flow(joint=True)
flow_add = make_flow(joint=False)

for f in (flow_joint, flow_add):
    f.fit(
        train,
        epochs=1200,
        learning_rate=1e-2,
        validation_data=val,
        callbacks=EarlyStopping(),
        seed=0,
    )
```

The joint model can represent the interaction, so it has the lower validation NLL per row.

```
nll_joint = sum(flow_joint.nll(val).values())
nll_add = sum(flow_add.nll(val).values())
print(f"val NLL  joint   : {nll_joint:.4f}")
print(f"val NLL  additive: {nll_add:.4f}")
print(
    f"gap over the {len(val)} validation rows: {len(val) * (nll_add - nll_joint):.1f}"
)
assert nll_joint < nll_add, "the joint model should fit the interaction better"
```

```
val NLL  joint   : 4.1660
val NLL  additive: 4.1990
gap over the 1000 validation rows: 33.0
```

`intercept_contributions`: the centered decomposition

It returns the contribution of each parent's network to the transform parameters `theta`, centered over the rows that you pass. The removed means collect in `baseline`. The last check shows that every column mean is about 0.

```
res_add = flow_add.intercept_contributions(train, "x3")
res_joint = flow_joint.intercept_contributions(train, "x3")
```

```
print("additive terms :", list(res_add["contributions"]))
print("joint terms    :", list(res_joint["contributions"]))

col_means = {
    k: float(np.abs(v.mean(0)).max()) for k, v in res_add["contributions"].items()
}
print("per-term column means (≈0)   :", col_means)
assert max(col_means.values()) < 1e-5
```

```
additive terms : ['x1', 'x2']
joint terms    : ['x1+x2']
per-term column means (≈0)   : {'x1': 2.0974874814783107e-07, 'x2': 1.0287761398330986e-
07}
```

The structural difference: separable curves vs an entangled cloud

Pick the transform coefficient that varies most across the data and plot each term's centered contribution to it against a parent value.

- **Additive**: `net(x1)` depends only on x1, so its contribution is a curve in x1, and the same applies to x2. These curves are the per-parent partial effects.
- **Joint**: the single x1+x2 component depends on both parents. Against x1 alone it is a cloud, and x2 (color) sets its height. The joint panel picks its own most-varying coefficient and has its own y-axis.

```
c1 = res_add"contributions"
c2 = res_add"contributions"
cj = res_joint"contributions"
k = int((c1.var(0) + c2.var(0)).argmax())
kj = int(cj.var(0).argmax())
x1v, x2v = train["x1"].values, train["x2"].values

fig, axes = plt.subplots(1, 2, figsize=(11, 4.2))
o1, o2 = np.argsort(x1v), np.argsort(x2v)
axes[0].plot(x1v[o1], c1[o1, k], color="#1b9e77", lw=2.5, label="net(x1)  vs x1")
axes[0].plot(x2v[o2], c2[o2, k], color="#d95f02", lw=2.5, label="net(x2)  vs x2")
axes[0].axhline(0, color="0.7", lw=0.8)
axes[0].set_title("additive: separable per-parent curves")
axes[0].set_xlabel("parent value")
axes[0].set_ylabel(f"centered contribution to theta[{k}]")
axes[0].legend()

sc = axes[1].scatter(x1v, cj[:, kj], c=x2v, s=8, alpha=0.6, cmap="coolwarm")
axes[1].axhline(0, color="0.7", lw=0.8)
axes[1].set_title("joint: against x1 a cloud (height set by x2)")
axes[1].set_xlabel("x1")
axes[1].set_ylabel(f"centered contribution to theta[{kj}]")
fig.colorbar(sc, ax=axes[1], label="x2")
fig.tight_layout()
plt.show()
```

Takeaway

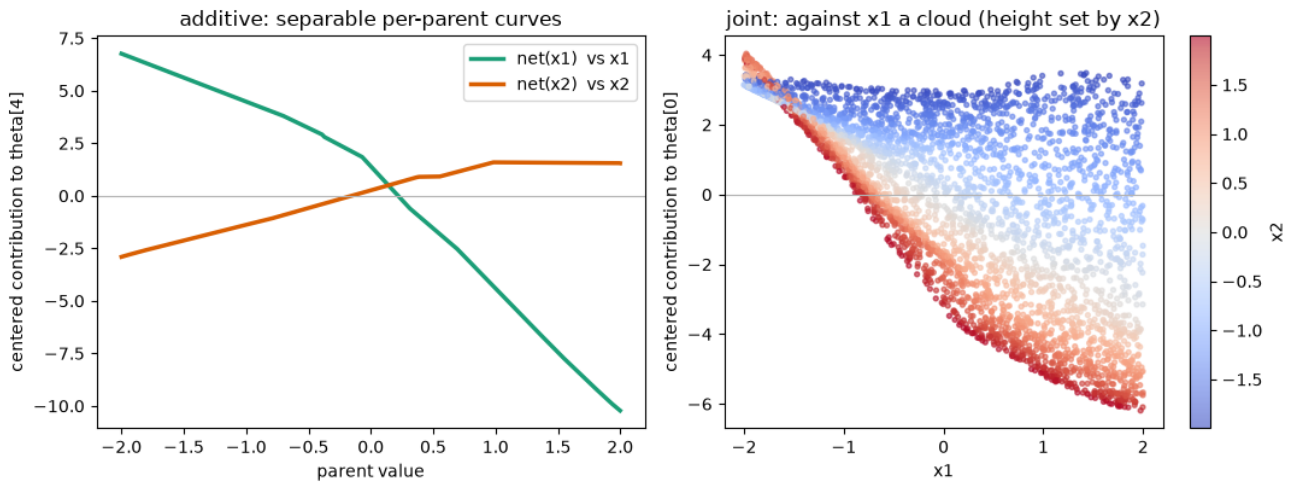


Figure 1: png

you want…	use	what you get
a per-parent partial-effect plot (“what does x1 do?”)	**additive** `CI("x1", "x2", allow_interaction=False)`	`intercept_contributions` → mean-centered, **separable** components
interactions between parents in the transform	**joint** `CI("x1", "x2")`	one entangled network: flexible, **not** separable

The contributions live in the unconstrained parameter space of the transform, before the monotonicity constraint. They are partial effects on the parameters, not an additive split of the curve `h`.

Classical fitting of all-`ls` TRAM-DAGs (`fit_classical`)

When **every edge of a TRAM-DAG is a linear shift (`LS`)**, each node-conditional is a classical model

- an ordered-logit / proportional-odds model for ordinal nodes
- a Colr model (R’s `tram::Colr`) for continuous ones.

For such a model `flow.fit_classical()` is the optimizer. It is a deterministic full-batch L-BFGS, and its report says when it reached the maximum-likelihood estimate (`docs/fitting.md`). This notebook checks the result against `statsmodels` and R.

The R references below are constants from `notebooks/classical_fit_tram_dag.R`. To check them, run `Rscript notebooks/classical_fit_tram_dag.R` from the repository root. It needs `MASS` and `tram`.

```
import time
from pathlib import Path

import numpy as np
import pandas as pd
import torch
from statsmodels.miscmodels.ordinal_model import OrderedModel
```

```python
from tramdag import LS, SI, CausalFlowDAG, ContinuousNode, Node, OrdinalNode
from tramdag.callbacks import EarlyStopping

HERE = [Path.cwd(), *Path.cwd().parents]
REPO = next(p for p in HERE if (p / "pyproject.toml").exists())
DATA = REPO / "experiments" / "data"
NB_DATA = REPO / "notebooks" / "data"


def claim(text, value, bound, fmt="{:.2e}"):
    """Assert a computed quantity stays inside its bound, and print both.

    Every agreement this notebook asserts goes through here, so no number is
    typed into a markdown cell where it can drift away from the code. The
    prose says what is checked. The bound is the only number in the source,
    and it leaves room for another machine.
    """
    ok = "ok  " if value <= bound else "FAIL"
    print(f"{ok}{text}: {fmt.format(value)} <= {fmt.format(bound)}")
    assert value <= bound, f"{text}: {fmt.format(value)} exceeds {fmt.format(bound)}"
```

0. The zeroth example: logistic regression on `MASS::birthwt`

A two-level `OrdinalNode` whose terms are all `LS` is logistic regression. The ordinal transform is

$$P(Y \le 0 \mid x) = \sigma\Big(\theta_0 - \textstyle\sum_p w_p x_p\Big)\,,$$

and with two levels there is a single cutpoint θ_0, so

$$\operatorname{logit} P(Y = 1 \mid x) = -\theta_0 + \textstyle\sum_p w_p x_p.$$

The shift weights are the logistic-regression coefficients. The intercept is minus the cutpoint, because an ordinal node subtracts its shift (`docs/notation.md`). Section 2 uses the same model with $K - 1$ cutpoints.

The data is `birthwt` from MASS, the low-birth-weight study of Hosmer & Lemeshow. The outcome is `low`, and the predictors are `age`, `lwt` and `smoke` (`notebooks/data/README.md`). This is not a causal model.

The R you can copy-paste

Paste this into an R session to get the reference fit:

```r
library(MASS)
m <- glm(low ~ age + lwt + smoke, data = birthwt, family = binomial)
coef(m)
logLik(m)
```

which prints

```
(Intercept)         age         lwt       smoke
 1.36822527 -0.03899458 -0.01213854  0.67076374
'log Lik.' -111.4396765 (df=4)
```

(R 4.2.3, MASS 7.3-58.2.) The cell below holds these numbers as `R_GLM`.

```
R_GLM = {
    "intercept": 1.3682252685,
    "age": -0.0389945827,
    "lwt": -0.0121385423,
    "smoke": 0.6707637407,
}
R_LOGLIK = -111.43967649

bw = pd.read_csv(NB_DATA / "birthwt.csv")
print(
    bw.head(3).to_string(index=False),
    f"\n\n{len(bw)} births, {bw['low'].sum()} of them low birth weight",
)
```

```
 low  age  lwt  smoke  bwt
   0   19  182      0 2523
   0   33  155      0 2551
   0   20  105      1 2557

189 births, 59 of them low birth weight
```

`low` is the outcome. `smoke` is a binary parent, so it is an ordinal node too. `age` and `lwt` are source nodes. The flow also models their marginals, which `glm` does not. They get the cheapest transform, `SI(transform="affine")`. The joint NLL decomposes per node, so the marginals cannot change the `low` node.

```
spec_bw = {
    "age": ContinuousNode(SI(transform="affine")),
    "lwt": ContinuousNode(SI(transform="affine")),
    "smoke": OrdinalNode(2),
    "low": OrdinalNode(2, LS("age") + LS("lwt") + LS("smoke")),
}

flow_bw = CausalFlowDAG(spec_bw, seed=0)
flow_bw.fit_classical(bw)
```

```
{'converged': True,
 'final_nll': 9.137612929448062,
 'seconds': 1.2732161150001957,
 'coefficients': {'low': {'age': array([-0.03899379]),
   'lwt': array([-0.01213884]),
   'smoke': array([-0.00437635,  0.66640591])}},
 'nodes': {'age': {'converged': True,
   'stop_reason': 'tolerance',
   'n_iter': 7,
   'final_nll': 3.089470463926682,
   'grad_norm': 1.0146105993900928e-06,
   'decrement': 1.3917636508756161e-12,
   'seconds': 1.206336879000105,
   'coefficients': {}},
  'lwt': {'converged': True,
   'stop_reason': 'tolerance',
   'n_iter': 8,
   'final_nll': 4.789085054515023,
   'grad_norm': 1.1217799113443504e-05,
   'decrement': 8.869592011349744e-11,
   'seconds': 0.007313385999964339,
```

```
   'coefficients': {}},
  'smoke': {'converged': True,
   'stop_reason': 'tolerance',
   'n_iter': 5,
   'final_nll': 0.6694294930549758,
   'grad_norm': 2.9695561377940383e-08,
   'decrement': 1.8507472827317275e-15,
   'seconds': 0.006837462000021333,
   'coefficients': {}},
  'low': {'converged': True,
   'stop_reason': 'tolerance',
   'n_iter': 30,
   'final_nll': 0.5896279179513815,
   'grad_norm': 0.00030424912919857543,
   'decrement': 2.679244908939033e-11,
   'seconds': 0.05272838800010504,
   'coefficients': {'age': array([-0.03899379]),
    'lwt': array([-0.01213884]),
    'smoke': array([-0.00437635,  0.66640591])}}}}
```

`age` and `lwt` are continuous parents, so they enter raw and their weights compare directly. `smoke` is ordinal, so it enters one-hot over both levels, w_0 and w_1. Only their difference is identified (`docs/interpretation.md`), and it is the `smoke` coefficient of R. With $s \in \{0, 1\}$, $\mathbf{1}[s = 0] = 1 - s$, so the one-hot pair collapses to

$$w_0 \, \mathbf{1}[s = 0] + w_1 \, \mathbf{1}[s = 1] \;=\; w_0 + (w_1 - w_0)\, s,$$

and the node reads

$$\operatorname{logit} P(\texttt{low} = 1) \;=\; (-\theta_0 + w_0) \;+\; w_{\text{age}}\, \texttt{age} \;+\; w_{\text{lwt}}\, \texttt{lwt} \;+\; (w_1 - w_0)\, s.$$

The first bracket is the `(Intercept)` of R. The cutpoints are $[-\infty, \theta_0, +\infty]$, so the code takes the finite one. The `statsmodels` formula is the R call above.

```python
import statsmodels.formula.api as smf  # noqa: E402

from tramdag.transforms import ordinal_cutpoints  # noqa: E402

res_bw = smf.logit("low ~ age + lwt + smoke", data=bw).fit(disp=False)

w = flow_bw.ls_coefficients()["low"]
with torch.no_grad():
    theta_0 = float(ordinal_cutpoints(flow_bw.nodes["low"].intercept(1))[0, 1])

rows_bw = [
    (
        "intercept",
        -theta_0 + float(w"smoke"),
        res_bw.params["Intercept"],
        R_GLM["intercept"],
    ),
    ("age", float(w"age"), res_bw.params["age"], R_GLM["age"]),
    ("lwt", float(w"lwt"), res_bw.params["lwt"], R_GLM["lwt"]),
    (
        "smoke",
```

```
        float(w"smoke" - w"smoke"),
        res_bw.params["smoke"],
        R_GLM["smoke"],
    ),
]
print(f"{'':<11}{'fit_classical':>14}{'sm.Logit':>10}{'R glm':>10}{'max|diff|':>11}")
spreads = []
for name, flow_value, sm_value, r_value in rows_bw:
    spreads.append(max(abs(flow_value - sm_value), abs(flow_value - r_value)))
    print(
        f"{name:<11}{flow_value:>14.4f}{sm_value:>10.4f}{r_value:>10.4f}"
        f"{spreads[-1]:>11.2e}"
    )
claim("max coefficient spread of the three fits", max(spreads), 6e-5)
```

```
            fit_classical  sm.Logit     R glm  max|diff|
intercept          1.3682    1.3682    1.3682   1.08e-06
age               -0.0390   -0.0390   -0.0390   7.92e-07
lwt               -0.0121   -0.0121   -0.0121   3.01e-07
smoke              0.6708    0.6708    0.6708   1.85e-05
ok  max coefficient spread of the three fits: 1.85e-05 <= 6.00e-05
```

L-BFGS on the flow, the Newton solver of `statsmodels` and the IRLS of R find the same maximum likelihood. `flow.pmf` gives the fitted probabilities of `glm` row by row. `nll()` gives the mean NLL per node; times $-n$ it is the log-likelihood.

```
ll_flow = -flow_bw.nll(bw)["low"] * len(bw)
p_flow = flow_bw.pmf(bw, node="low")[:, 1]
p_classical = res_bw.predict(bw).values
print(f"log-likelihood   flow {ll_flow:.6f}   R glm {R_LOGLIK:.6f}")
claim("|logLik_flow - logLik_glm|", abs(ll_flow - R_LOGLIK), 3e-6)
claim(
    f"max |P_flow(low=1) - P_glm(low=1)| over {len(bw)} rows",
    float(np.abs(p_flow - p_classical).max()),
    1e-5,
)
```

```
log-likelihood   flow -111.439677   R glm -111.439676
ok  |logLik_flow - logLik_glm|: 6.89e-07 <= 3.00e-06
ok  max |P_flow(low=1) - P_glm(low=1)| over 189 rows: 6.27e-06 <= 1.00e-05
```

The same regression without the DAG. A lone `Node` drops the marginals of `age` and `lwt`. `Node.fit_classical` finds the same maximum likelihood as the flow, R and `statsmodels`.

```
low = Node(
    "low", spec_bw["low"], {"age": "continuous", "lwt": "continuous", "smoke": 2}
)
low.fit_classical(bw[["age", "lwt", "smoke", "low"]])
w_low = low.ls_coefficients()
ll_low = -low.nll(bw) * len(bw)
print(f"log-likelihood   node alone {ll_low:.6f}   R glm {R_LOGLIK:.6f}")
print(
    f"smoke   node alone {float(w_low'smoke' - w_low'smoke'):.4f}"
    f"   R glm {R_GLM['smoke']:.4f}"
)
assert abs(ll_low - R_LOGLIK) < 1e-4, "the lone node misses the glm likelihood"
```

```
log-likelihood   node alone -111.439666   R glm -111.439676
smoke   node alone 0.6708   R glm 0.6708
```

1. Continuous case

`birthwt` also holds `bwt`, the birth weight in grams that `low` is cut from. With a flexible monotone transform and a logistic latent, the same three predictors give the Colr model of R:

```
tram::Colr(bwt ~ age + lwt + smoke, order = 21)
```

`glm` cannot fit this model. The R references come from this call (R 4.2.3, tram 1.0.4).

```
R_COLR_BWT = {
    "age": -0.021971,
    "lwt": -0.010233,
    "smoke": 0.668698,
    "loglik": -1499.6678,
}

spec_bwt = {
    "age": ContinuousNode(SI(transform="affine")),
    "lwt": ContinuousNode(SI(transform="affine")),
    "smoke": OrdinalNode(2),
    "bwt": ContinuousNode(
        SI(transform_kwargs={"n_coeffs": 20}) + LS("age") + LS("lwt") + LS("smoke")
    ),
}

flow_bwt = CausalFlowDAG(spec_bwt, seed=0)
flow_bwt.fit_classical(bw)

w = flow_bwt.ls_coefficients()["bwt"]
fitted = {
    "age": w"age",
    "lwt": w"lwt",
    "smoke": w"smoke" - w"smoke",
    "loglik": -flow_bwt.nll(bw)["bwt"] * len(bw),
}

print(f"{'':<9}{'age':>9}{'lwt':>9}{'smoke':>9}{'logLik':>11}")
for name, r in [("flow", fitted), ("R Colr", R_COLR_BWT)]:
    print(
        f"{name:<9}{r['age']:>9.4f}{r['lwt']:>9.4f}{r['smoke']:>9.4f}"
        f"{r['loglik']:>11.2f}"
    )
diff = {k: abs(fitted[k] - R_COLR_BWT[k]) for k in fitted}
print(
    f"{'|diff|':<9}{diff['age']:>9.4f}{diff['lwt']:>9.4f}{diff['smoke']:>9.4f}"
    f"{diff['loglik']:>11.2f}"
)
```

```
                age      lwt    smoke     logLik
flow        -0.0223  -0.0103   0.6641   -1500.55
R Colr      -0.0220  -0.0102   0.6687   -1499.67
|diff|       0.0004   0.0000   0.0046       0.88
```

The agreement is not exact, because the two libraries place the Bernstein basis differently. The flow puts it on the `range_q` quantiles with linear tails (`docs/model.md`). `Colr` uses the full range of the data. The two fits thus maximize slightly different likelihoods. The next cells compare the differences with the standard errors.

Standard errors and confidence intervals

The sampling distribution of the maximum likelihood estimator is

$$\hat{\theta} \;\dot{\sim}\; N(\theta, I^{-1})$$

where I is the Fisher information matrix. θ is unknown, so I is evaluated at $\hat{\theta}$. The standard errors are on the diagonal of I^{-1}. For a contrast c, the standard error is $\sqrt{c^\top I^{-1} c}$.

The **observed** Fisher information is the Hessian of the *negative* log-likelihood at the MLE,

$$I_{ab} = \left.\frac{\partial^2(-\ell)}{\partial\theta_a\,\partial\theta_b}\right|_{\hat{\theta}},$$

which autograd gives: one backward pass per component gives a row of I.

Here I is singular, so I^{-1} does not exist. The one-hot `smoke` pair and the Bernstein intercept have directions without curvature (see "weakly identified directions" in `docs/fitting.md`). I is symmetric, so it has an orthonormal eigendecomposition. The **pseudo-inverse** inverts only the directions that have curvature:

$$I = \sum_{k=1}^{P} \lambda_k\, e_k e_k^\top \qquad\Longrightarrow\qquad I^{+} = \sum_{k:\;\lambda_k > \tau} \frac{1}{\lambda_k}\, e_k e_k^\top, \qquad \tau = 10^{-7}\,\lambda_{\max}.$$

Terms with $\lambda_k \le \tau$ are dropped and not inverted. This is correct for a contrast that avoids the flat subspace. The `leak` column measures how far a contrast enters it:

$$\text{leak}(c) = \max_{k:\;\lambda_k \le \tau} \left|c^\top e_k\right|.$$

Only interpret rows whose leak is close to zero. The spectrum has a flat block near zero, one weakly curved Bernstein direction below τ, and the curved directions. The threshold drops the weak direction with the flat block. The cell below computes the standard errors again with a threshold between the two and compares. The Hessian runs in float64.

```
from scipy.stats import norm  # noqa: E402


def observed_information(flow, node, data):
    """Give one node's NLL Hessian, the gradient norm, and parameter offsets."""
    flow.double()
    names, params = zip(*flow.nodes[node].named_parameters(), strict=True)
    nd = flow.nodes[node]
    nll = -nd.row_log_prob(nd.tensorize(data)).sum()
    grad = torch.cat(
        [g.reshape(-1) for g in torch.autograd.grad(nll, params, create_graph=True)]
    )
    hess = torch.stack(
        [
            torch.cat(
                [
                    h.reshape(-1)
                    for h in torch.autograd.grad(grad[i], params, retain_graph=True)
                ]
```

```
            )
            for i in range(grad.numel())
        ]
    )
    offsets, at = {}, 0
    for name, p in zip(names, params, strict=True):
        offsets[name] = at
        at += p.numel()
    result = hess.detach().numpy(), float(grad.detach().norm()), offsets
    flow.float()
    return result


def spectrum_split(evals, tau):
    """Split the eigenvalues at or below ``tau`` into a flat and a weak block.

    The flat block is the directions near zero. The weak block is the
    directions above the widest log-gap among the eigenvalues at or below
    ``tau``. ``tau`` drops them too. The result gives the size of
    the flat block and its largest eigenvalue, and the smallest weak
    eigenvalue, so that a caller can place a second threshold between them.
    """
    low = np.sort(np.maximum(np.abs(evals[np.abs(evals) <= tau]), 1e-300))
    cut = int(np.argmax(np.diff(np.log(low)))) + 1 if low.size > 1 else low.size
    return {
        "n_flat": cut,
        "flat_max": float(low[cut - 1]) if cut else 0.0,
        "weak_min": float(low[cut]) if cut < low.size else 0.0,
    }


def ls_conf_int(flow, node, data, level=0.95, rcond=1e-7):
    """Give Wald intervals for the linear-shift coefficients of one node."""
    hess, grad_norm, offsets = observed_information(flow, node, data)
    evals, evecs = np.linalg.eigh(hess)
    tau = rcond * np.abs(evals).max()
    flat = evecs[:, np.abs(evals) <= tau]
    cov = np.linalg.pinv(hess, rcond=rcond)
    z = norm.ppf(0.5 + level / 2)

    rows = []
    for parent, w in flow.ls_coefficients()[node].items():
        c = np.zeros(hess.shape[0])
        key = next(n for n in offsets if n.startswith(f"shifts.{parent}"))
        if len(w) == 1:
            c[offsets[key]], estimate, term = 1.0, w[0], parent
        else:
            c[offsets[key]], c[offsets[key] + 1] = -1.0, 1.0
            estimate, term = w[1] - w[0], f"{parent} (1 vs 0)"
        se = float(np.sqrt(c @ cov @ c))
        rows.append(
            {
                "term": term,
                "estimate": estimate,
                "se": se,
```

```
                "lower": estimate - z * se,
                "upper": estimate + z * se,
                "leak": float(np.abs(c @ flat).max()) if flat.size else 0.0,
            }
        )
    diagnostics = {
        "n_params": hess.shape[0],
        "grad_norm": grad_norm,
        "lambda_max": float(np.abs(evals).max()),
        **spectrum_split(evals, tau),
    }
    return pd.DataFrame(rows).set_index("term"), diagnostics

table, diag = ls_conf_int(flow_bwt, "bwt", bw)
print(table.round(6).to_string())
print(
    f"\n{diag['n_params']} parameters, {diag['n_flat']} of them without curvature"
    f"   |grad| = {diag['grad_norm']:.1e}"
)
```

```
                estimate        se     lower     upper  leak
term
age            -0.022342  0.025265 -0.071860  0.027176   0.0
lwt            -0.010276  0.004113 -0.018337 -0.002216   0.0
smoke (1 vs 0)  0.664065  0.259577  0.155303  1.172827   0.0

24 parameters, 12 of them without curvature   |grad| = 2.5e-02
```

The reference standard errors come from the same `Colr` fit.

```
R_COLR_BWT_SE = {"age": 0.025305, "lwt": 0.004116, "smoke": 0.260131}

print()
print(
    f"{diag['n_flat']} flat directions, largest eigenvalue {diag['flat_max']:.1e}"
    f"   weak direction {diag['weak_min']:.1e}"
    f"   tau = {1e-7 * diag['lambda_max']:.1e}"
)
tau2 = np.sqrt(diag["flat_max"] * diag["weak_min"])
table2, _ = ls_conf_int(flow_bwt, "bwt", bw, rcond=tau2 / diag["lambda_max"])
claim(
    "max |SE(tau) - SE(tau2)|, tau2 between the flat block and the weak direction",
    float((table["se"] - table2["se"]).abs().max()),
    1e-3,
)
rows_se = [("age", "age"), ("lwt", "lwt"), ("smoke (1 vs 0)", "smoke")]
se_gap = max(abs(float(table.loc[t, "se"]) - R_COLR_BWT_SE[r]) for t, r in rows_se)
claim("max |SE_flow - SE_Colr|", se_gap, 1e-3)
claim(
    "max |coef_flow - coef_Colr| / SE",
    max(diff[r] / float(table.loc[t, "se"]) for t, r in rows_se),
    0.07,
)
```

```
12 flat directions, largest eigenvalue 2.0e-10   weak direction 2.8e-07   tau = 1.2e-01
ok  max |SE(tau) - SE(tau2)|, tau2 between the flat block and the weak direction: 3.33e-
08 <= 1.00e-03
ok  max |SE_flow - SE_Colr|: 5.54e-04 <= 1.00e-03
```

```
ok  max |coef_flow - coef_Colr| / SE: 1.78e-02 <= 7.00e-02
```

The standard errors of all three parents agree with those of `Colr`, and the coefficient differences are a small fraction of one standard error.

1b. A bimodal DAG: the demo data

The data is the bimodal VACA triangle `x1 → x2 → x3 ← x1` of the demo notebook. Each node of the all-LS model is a Colr model. The DGP noise is Gaussian and the latent is logistic, so the model is misspecified. `fit_classical` still finds its MLE.

```
if False:

    def sample_vaca(n, seed):
        """Draw n rows from the VACA bimodal triangle SCM."""
        rng = np.random.default_rng(seed)
        mix, a, b = rng.uniform(size=n), rng.normal(size=n), rng.normal(size=n)
        x1 = np.where(mix < 0.5, -2.0 + np.sqrt(1.5) * a, 1.5 + b)
        x2 = -x1 + rng.normal(size=n)
        x3 = x1 + 0.25 * x2 + rng.normal(size=n)
        return pd.DataFrame({"x1": x1, "x2": x2, "x3": x3})

    df = sample_vaca(1000, seed=42)
    df.to_csv(NB_DATA / "vaca.csv", index=False)
```

The R you can copy-paste

`tram::Colr` fits the same model on the tracked `notebooks/data/vaca.csv`. The two libraries count the basis differently: `N_COEFFS = 20` below is `order = 21` in tram, because zuko adds two control points (`docs/zuko-upstream.md`).

```
library(tram)
d <- read.csv("notebooks/data/vaca.csv")

m2 <- Colr(x2 ~ x1, data = d, order = 21)
coef(m2)     # -> x1 1.800042
logLik(m2)   # -> -1401.213542

m3 <- Colr(x3 ~ x1 + x2, data = d, order = 21)
coef(m3)     # -> x1 -1.777289   x2 -0.455282
logLik(m3)   # -> -1411.901958
```

`Colr` reports the shift on the same side as the flow, so the signs match.

```
R_COLR = {
    "x2": {"coef": {"x1": 1.800042}, "loglik": -1401.213542},
    "x3": {"coef": {"x1": -1.777289, "x2": -0.455282}, "loglik": -1411.901958},
}
```

The reference values come from `classical_fit_tram_dag.R` (R 4.2.3, tram 1.0.4).

```
df = pd.read_csv(NB_DATA / "vaca.csv")
N_COEFFS = 20

spec_vaca = {
    "x1": ContinuousNode(SI(transform_kwargs={"n_coeffs": N_COEFFS})),
    "x2": ContinuousNode(SI(transform_kwargs={"n_coeffs": N_COEFFS}) + LS("x1")),
    "x3": ContinuousNode(
        SI(transform_kwargs={"n_coeffs": N_COEFFS}) + LS("x1") + LS("x2")
```

```
    ),
}

flow_c = CausalFlowDAG(spec_vaca, seed=0)
rep = flow_c.fit_classical(df)
print(f"\n{'':<12}{'fit_classical':>14}{'R Colr':>12}{'|diff|':>10}")
loglik = {k: -v * len(df) for k, v in flow_c.nll(df).items()}
for node, parents in flow_c.ls_coefficients().items():
    for p, w in parents.items():
        c, r = w[0], R_COLRnode[p]
        print(f"{p + ' -> ' + node:<12}{c:>14.4f}{r:>12.4f}{abs(c - r):>10.4f}")
    c, r = loglik[node], R_COLRnode
    print(f"{'logLik ' + node:<12}{c:>14.4f}{r:>12.4f}{abs(c - r):>10.4f}")
```

```
             fit_classical      R Colr    |diff|
x1 -> x2            1.7986      1.8000    0.0014
logLik x2       -1401.8708  -1401.2135    0.6573
x1 -> x3            -1.7796     -1.7773    0.0023
x2 -> x3            -0.4549     -0.4553    0.0004
logLik x3       -1410.9656  -1411.9020    0.9364
```

Why this one is not an exact-MLE check

The coefficients agree less closely than in Sections 0 and 2, because of the basis placement of Section 1. The relative gap of the shift coefficients stays small.

```
gaps = [
    (abs(float(w[0]) - R_COLRnode[parent]), abs(R_COLRnode[parent]))
    for node in R_COLR
    for parent, w in flow_c.ls_coefficients()[node].items()
]
claim("max |coef_flow - coef_Colr|", max(g for g, _ in gaps), 0.01)
claim("max relative coefficient gap", max(g / r for g, r in gaps), 0.02)
```

```
ok  max |coef_flow - coef_Colr|: 2.34e-03 <= 1.00e-02
ok  max relative coefficient gap: 1.31e-03 <= 2.00e-02
```

The fitted conditional density

`flow.density(df, node, grid)` evaluates $p(\text{node} = g \mid \text{pa})$ at each grid value, in closed form from the transform and with no sampling. It is the continuous counterpart of `flow.pmf`.

A density must integrate to one over a wide grid, which checks the log-determinant of the transform. Its mode must move as the linear shifts say.

```
grid = np.linspace(df["x3"].min() - 2, df["x3"].max() + 2, 601)
rows = df.iloc[:200]
dens = flow_c.density(rows, "x3", grid)
mass = np.trapezoid(dens, grid, axis=1)
claim("worst |integral of the density - 1|", float(np.abs(mass - 1.0).max()), 5e-3)
```

```
ok  worst |integral of the density - 1|: 1.29e-03 <= 5.00e-03
```

A continuous node adds the shift to $h(x)$, so a positive weight lowers the mode.

```
modes = grid[dens.argmax(axis=1)]
shift = -(
    float(flow_c.ls_coefficients()"x3"[0]) * rows["x1"]
    + float(flow_c.ls_coefficients()"x3"[0]) * rows["x2"]
```

```
)
corr = float(np.corrcoef(modes, shift)[0, 1])
print(f"corr(mode of the fitted density, the linear predictor) = {corr:.4f}")
assert corr > 0.95, f"the density's mode does not track the shifts: r = {corr:.4f}"
```

```
corr(mode of the fitted density, the linear predictor) = 0.9951
```

Classical against Adam: the same optimum. An Adam fit with `EarlyStopping`, validated on the training rows, reaches the same coefficients. The classical fit gives determinism and the exact MLE, not speed. The cell prints both timings and does not assert them.

```
flow_a = CausalFlowDAG(spec_vaca, seed=0)
t0 = time.perf_counter()
flow_a.fit(
    df,
    epochs=2000,
    batch_size=4096,
    learning_rate=1e-1,
    validation_data=df,
    callbacks=EarlyStopping(patience=60),
    seed=0,
)
t_adam = time.perf_counter() - t0

coef_c, coef_a = flow_c.ls_coefficients(), flow_a.ls_coefficients()
print(f"{'coef':<10}{'classical':>12}{'adam':>12}{'|diff|':>10}")
adam_gaps = []
for node, p in [("x2", "x1"), ("x3", "x1"), ("x3", "x2")]:
    c, aw = coef_cnode[0], coef_anode[0]
    adam_gaps.append(abs(c - aw))
    print(f"{p}->{node:<6}{c:>12.4f}{aw:>12.4f}{adam_gaps[-1]:>10.4f}")
print(f"\nclassical {rep['seconds']:.2f}s  vs  adam {t_adam:.1f}s")
claim("max |coef_classical - coef_adam|", float(max(adam_gaps)), 1e-3)
```

```
coef         classical        adam    |diff|
x1->x2          1.7986      1.7986    0.0001
x1->x3         -1.7796     -1.7799    0.0003
x2->x3         -0.4549     -0.4550    0.0002

classical 4.22s  vs  adam 30.9s
ok  max |coef_classical - coef_adam|: 3.14e-04 <= 1.00e-03
```

2. Ordinal case

For an ordinal outcome the classical model is the ordered logit of `statsmodels.OrderedModel`. The data is the paper's mixed triangle (`experiments/data/triangle-mixed/linear`): `x3` has four levels and the continuous parents `x1` and `x2` (`docs/paper-replication.md`).

```
tm = pd.read_csv(DATA / "triangle-mixed" / "linear" / "obs.csv")

spec_tm = {
    "x1": ContinuousNode(),
    "x2": ContinuousNode(LS("x1")),
    "x3": OrdinalNode(4, LS("x1") + LS("x2")),
}

flow_o = CausalFlowDAG(spec_tm, seed=0)
flow_o.fit_classical(tm)
```

```
{'converged': True,
 'final_nll': 1.0638599166788296,
 'seconds': 11.120403674999807,
 'coefficients': {'x2': {'x1': array([2.02991896])},
  'x3': {'x1': array([0.00974636]), 'x2': array([0.38319398])}},
 'nodes': {'x1': {'converged': True,
   'stop_reason': 'tolerance',
   'n_iter': 125,
   'final_nll': -0.5284964811303434,
   'grad_norm': 7.935367721321228e-06,
   'decrement': 3.953522251566813e-10,
   'seconds': 1.6824130819999255,
   'coefficients': {}},
  'x2': {'converged': True,
   'stop_reason': 'tolerance',
   'n_iter': 241,
   'final_nll': 0.3826475461052615,
   'grad_norm': 2.3250106041379636e-06,
   'decrement': 8.507781824995219e-10,
   'seconds': 9.39822827599994,
   'coefficients': {'x1': array([2.02991896])}},
  'x3': {'converged': True,
   'stop_reason': 'tolerance',
   'n_iter': 15,
   'final_nll': 1.2097088517039114,
   'grad_norm': 6.470020896085258e-06,
   'decrement': 3.1231225589370035e-10,
   'seconds': 0.039762316999940595,
   'coefficients': {'x1': array([0.00974636]), 'x2': array([0.38319398])}}}}
```

Refitting the same design in statsmodels

`flow.design_matrix(df, node, drop_first=True)` gives the encoding of the flow's own shifts, so `statsmodels` fits the same design.

```
design = flow_o.design_matrix(tm, "x3", drop_first=True)
res = OrderedModel(tm["x3"].astype(int), design, distr="logit").fit(
    method="bfgs", disp=False
)

fitted = flow_o.ls_coefficients()["x3"]
print(f"{'coefficient':<14}{'fit_classical':>14}{'statsmodels':>13}{'|diff|':>9}")
for name in ("x1", "x2"):
    a_, b_ = float(fittedname), res.params[name]
    print(f"{name:<14}{a_:>14.4f}{b_:>13.4f}{abs(a_ - b_):>9.4f}")
```

```
coefficient    fit_classical  statsmodels   |diff|
x1                    0.0097       0.0097   0.0001
x2                    0.3832       0.3832   0.0001
```

Standard errors

The helper of Section 1 applies unchanged, and `statsmodels` reports the same quantities.

```
table_o, diag_o = ls_conf_int(flow_o, "x3", tm)
ci_sm = res.conf_int()
print(
```

```
    f"{'':<6}{'flow SE':>9}{'sm SE':>8}{'|est|/SE':>10}   "
    f"{'flow 95%':>18}{'statsmodels 95%':>20}"
)
for term in ("x1", "x2"):
    r = table_o.loc[term]
    print(
        f"{term:<6}{r['se']:>9.4f}{res.bse[term]:>8.4f}"
        f"{abs(r['estimate']) / r['se']:>10.2f}   "
        f"[{r['lower']:+.3f}, {r['upper']:+.3f}]  "
        f"[{ci_sm.loc[term, 0]:+.3f}, {ci_sm.loc[term, 1]:+.3f}]"
    )

se_gap = max(
    abs(float(table_o.loc[term, "se"]) - float(res.bse[term])) for term in ("x1", "x2")
)
claim("max |SE_flow - SE_statsmodels|", se_gap, 1e-3)
```

```
        flow SE   sm SE  |est|/SE             flow 95%     statsmodels 95%
x1       0.1080  0.1080      0.09   [-0.202, +0.221]  [-0.202, +0.221]
x2       0.0735  0.0735      5.21   [+0.239, +0.527]  [+0.239, +0.527]
ok  max |SE_flow - SE_statsmodels|: 8.14e-07 <= 1.00e-03
```

The standard errors agree with `statsmodels`. The cell below checks that the coefficient gap is a small fraction of one standard error.

```
print(f"|grad| at the classical optimum: {diag_o['grad_norm']:.1e}")
for term in ("x1", "x2"):
    observed = abs(float(table_o.loc[term, "estimate"]) - float(res.params[term]))
    print(f"{term:<4} |flow - statsmodels| = {observed:.2e}")
    claim(
        f"{term}: gap against statsmodels / SE",
        observed / float(table_o.loc[term, "se"]),
        2.5e-3,
    )
```

```
|grad| at the classical optimum: 3.2e-02
x1   |flow - statsmodels| = 8.95e-05
ok  x1: gap against statsmodels / SE: 8.28e-04 <= 2.50e-03
x2   |flow - statsmodels| = 5.52e-05
ok  x2: gap against statsmodels / SE: 7.51e-04 <= 2.50e-03
```

3. Warm start: classical fit, then further training

A continued `fit()` from the classical solution stays put. This confirms the optimum and makes the classical fit a warm start.

```
before = {k: v.copy() for k, v in flow_o.ls_coefficients()["x3"].items()}

flow_o.fit(tm, epochs=300, learning_rate=1e-3, batch_size=256, seed=0)
after = flow_o.ls_coefficients()["x3"]

print("coefficient drift after 300 more Adam epochs from the classical MLE:")
for p in ["x1", "x2"]:
    d = float(np.abs(after[p] - before[p]).max())
    claim(f"{p}: max |drift| / SE", d / float(table_o.loc[p, "se"]), 0.1)
```

```
coefficient drift after 300 more Adam epochs from the classical MLE:
ok  x1: max |drift| / SE: 3.39e-02 <= 1.00e-01
```

```
ok  x2: max |drift| / SE: 9.29e-03 <= 1.00e-01
```

TRAM-DAG: one causal model, all three rungs

A TRAM-DAG (Sick & Dürr, 2025) is one normalizing flow wired like the adjacency matrix of a causal DAG. Fit it once on observational data. Then answer all three rungs of Pearl's ladder with the same fitted model:

rung	query	call
L1 association	$p(x)$, sampling	`flow.log_prob(df)`, `flow.sample(n)`
L2 intervention	$p(x \mid do(x_j{=}a))$	`flow.sample(n, do={...})`
L3 counterfactual	what would x_i have been, had x_j been a?	`u = flow.abduct(df)`, then `flow.sample(do={...}, u=u)`

This notebook uses the first benchmark of the paper. It is a three-variable process with a bimodal source, and its ground truth is known analytically. Every claim below is therefore a computed check, not a statement. A failed check stops the notebook.

The model theory is in `docs/model.md`. How to read a fitted model is in `docs/interpretation.md`.

The notebook runs on CPU in well under five minutes.

The first code cell installs `tramdag` from PyPI when the package is missing, as on Colab. To track active development instead of the latest release, install with `pip install "git+https://github.com/tensorchiefs/tramdag.git@main"`.

```
import importlib.util
import subprocess
import sys
import time

if importlib.util.find_spec("tramdag") is None:
    subprocess.check_call(
        [sys.executable, "-m", "pip", "install", "-q", "tramdag[plots]"]
    )

import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import torch

from tramdag import CausalFlowDAG, ContinuousNode, I, plot_dag
from tramdag.callbacks import EarlyStopping
from tramdag.plots import plot_marginals, plot_training

DEVICE = "cuda" if torch.cuda.is_available() else "cpu"
plt.rcParams["figure.dpi"] = 110
t_total = time.perf_counter()
print(f"torch {torch.__version__}  device: {DEVICE}")
```

```
torch 2.14.1+cu130  device: cpu
```

1. The benchmark process

The process comes from Sánchez-Martín et al. (2022, App. E.1) and from App. C.1 of Sick & Dürr (2025):

$$
\begin{aligned}
x_1 &\sim \tfrac{1}{2}\,\mathcal{N}(-2,\,1.5) + \tfrac{1}{2}\,\mathcal{N}(1.5,\,1) \quad \text{(bimodal source)} \\
x_2 &= -x_1 + \mathcal{N}(0,1) \\
x_3 &= x_1 + 0.25\,x_2 + \mathcal{N}(0,1)
\end{aligned}
$$

The DAG is $x_1 \to x_2$, $x_1 \to x_3$, $x_2 \to x_3$. The source is bimodal, which is the part that a location-scale model cannot fit; the monotone transform of each node is what fits it (`docs/model.md`).

The code writes the process out, so the notebook needs nothing but `tramdag`. `draw_latents` and `simulate` stay separate on purpose. Keeping the noise lets section 5 intervene on the same individuals, which makes an individual counterfactual checkable.

```
def draw_latents(n, rng):
    """Draw every noise variable of the process, n rows each."""
    return {
        "x1_mix": rng.uniform(size=n),
        "x1_a": rng.normal(size=n),
        "x1_b": rng.normal(size=n),
        "x2": rng.normal(size=n),
        "x3": rng.normal(size=n),
    }


def simulate(latents, do=None):
    """Run the process forward. A variable named in `do` is clamped instead."""
    do = do or {}
    n = len(latents["x2"])

    if "x1" in do:
        x1 = np.full(n, float(do["x1"]))
    else:
        x1 = np.where(
            latents["x1_mix"] < 0.5,
            -2.0 + np.sqrt(1.5) * latents["x1_a"],
            1.5 + latents["x1_b"],
        )

    x2 = np.full(n, float(do["x2"])) if "x2" in do else -x1 + latents["x2"]
    x3 = x1 + 0.25 * x2 + latents["x3"]
    return pd.DataFrame({"x1": x1, "x2": x2, "x3": x3})


def sample_dgp(n, seed, do=None):
    """Draw n fresh rows, optionally under an intervention."""
    return simulate(draw_latents(n, np.random.default_rng(seed)), do)


N = 20_000
df = sample_dgp(N, seed=43)
train, val = df.iloc[: N - 2_000], df.iloc[N - 2_000 :]

fig, ax = plt.subplots(figsize=(5.5, 3))
```

```
ax.hist(df["x1"], bins=80, density=True, alpha=0.7)
ax.set_title(f"the source $x_1$ has two modes ({N:,} rows)")
ax.set_xlabel("$x_1$")
ax.set_ylabel("density")
fig.tight_layout()
plt.show()
```

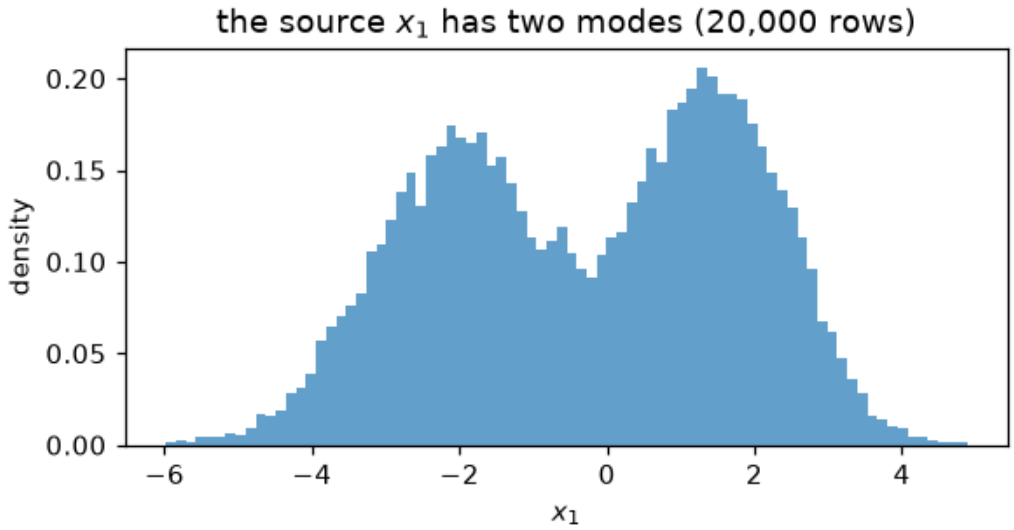


Figure 2: png

2. The spec is the DAG, and one fit

A spec is one entry per node. Each entry names the terms through which the parents enter. `I(...)` puts the parents in control of the transform parameters, which is the most flexible term. `notebooks/classical_fit_tram_dag.py` writes the same DAG with interpretable terms.

Each node fits on its own Adam loop (`docs/fitting.md`). `validation_data=` scores the validation rows per epoch, and `verbose=` prints progress. The strategy attaches through `callbacks=`. `EarlyStopping` stops each node when its own validation score stops improving, and loads its best weights back. Thus `epochs=200` is a ceiling and not a budget. The `seed=` of `CausalFlowDAG` makes the initial weights reproducible. The `seed=` of `fit` makes the minibatch order reproducible.

`notebooks/training_strategies.py` works through the other recipes.

```
spec = {
    "x1": ContinuousNode(),
    "x2": ContinuousNode(I("x1")),
    "x3": ContinuousNode(I("x1", "x2")),
}
plot_dag(spec)
plt.show()
```

```
flow = CausalFlowDAG(spec, device=DEVICE, seed=0)

t0 = time.perf_counter()
flow.fit(
    train,
    epochs=200,
    batch_size=2048,
    validation_data=val,
```

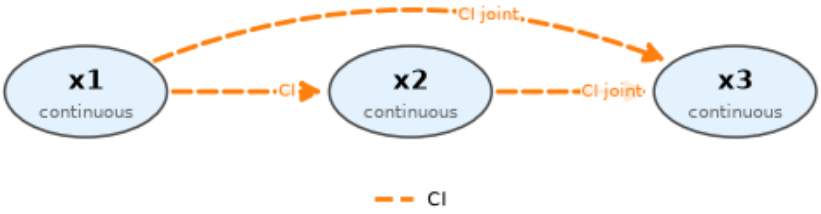


Figure 3: png

```
    verbose=50,
    learning_rate=1e-1,
    callbacks=EarlyStopping(patience=40),
    seed=0,
)
t_fit = time.perf_counter() - t0
stopped = {name: len(nd.history["train"]) for name, nd in flow.nodes.items()}
print(f"\nfitted on {DEVICE} in {t_fit:.1f}s")
print(f"each node stopped at epoch: {stopped}")

assert max(stopped.values()) < 200, (
    "a node used all 200 epochs, so its early stop never triggered; raise the "
    "ceiling before trusting anything below"
)
```

```
x1: epoch 50/200  train 2.0584  val 2.0804


x1: epoch 100/200  train 2.0588  val 2.0815


x1: epoch 150/200  train 2.0588  val 2.0807


x1: epoch 178/200  train 2.0585  val 2.0816


x2: epoch 50/200  train 1.4213  val 1.4231


x2: epoch 95/200  train 1.4187  val 1.4210


x3: epoch 50/200  train 1.4208  val 1.4368


x3: epoch 92/200  train 1.4194  val 1.4371

fitted on cpu in 18.8s
each node stopped at epoch: {'x1': 178, 'x2': 95, 'x3': 92}
```

```
plot_training(flow, stops=True)
plt.show()
```

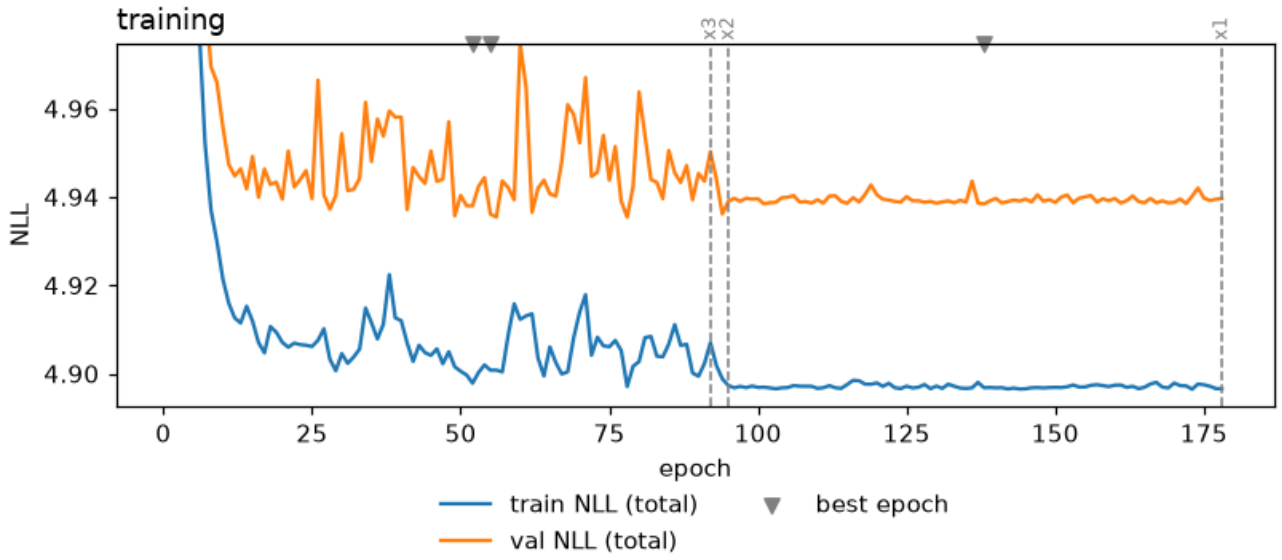


Figure 4: png

3. Rung 1: the observational fit

Samples from the fitted flow must reproduce the observed joint distribution. The marginals are the picture. The correlation matrix is the check, because it also covers the dependence between variables, which a marginal plot does not show.

```
plot_marginals(flow, df, seed=1)
plt.show()
```

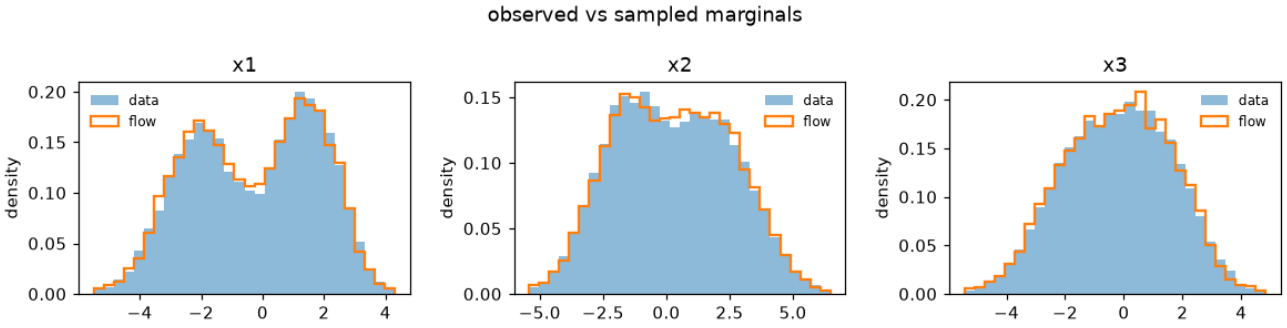


Figure 5: png

```
samp = flow.sample(len(df), seed=1)
corr_gap = float(np.abs(samp.corr().to_numpy() - df.corr().to_numpy()).max())
print(f"largest absolute difference between correlation matrices: {corr_gap:.4f}")
assert corr_gap < 0.05, f"the sampled dependence structure is off by {corr_gap:.4f}"
```

```
largest absolute difference between correlation matrices: 0.0178
```

4. Rung 2: interventions

`do=` mutilates the graph. It clamps x_2, cuts the edge into it, and resamples everything downstream. Under the process, $x_3 \mid do(x_2{=}a) = x_1 + 0.25a + \mathcal{N}(0,1)$, so $\mathbb{E}[x_3] = -0.25 + 0.25a$ exactly. That analytic target is what makes an error visible.

```
fig, axes = plt.subplots(1, 3, figsize=(11, 3.2), sharey=True)
errors = []
print("E[x3 | do(x2=a)]     analytic    TRAM-DAG      error")
for i, (ax, a) in enumerate(zip(axes, (-3.0, -1.0, 0.0), strict=True)):
    truth = sample_dgp(len(df), seed=543, do={"x2": a})
    fitted = flow.sample(len(df), do={"x2": a}, seed=2 + i)
    bins = np.linspace(truth["x3"].quantile(0.001), truth["x3"].quantile(0.999), 70)
    ax.hist(truth["x3"], bins=bins, density=True, alpha=0.5, label="process")
    ax.hist(
        fitted["x3"], bins=bins, density=True, histtype="step", lw=1.6, label="TRAM-DAG"
    )
    ax.set_title(f"$p(x_3 \\mid do(x_2={a:g}))$")
    ax.set_xlabel("$x_3$")

    analytic = -0.25 + 0.25 * a
    got = float(fitted["x3"].mean())
    errors.append(abs(got - analytic))
    print(
        f"   a = {a:+.0f}:            {analytic:+.4f}     {got:+.4f}     {errors[-1]:.4f}"
    )
axes[0].legend()
fig.tight_layout()
plt.show()
```

```
E[x3 | do(x2=a)]     analytic    TRAM-DAG      error

   a = -3:            -1.0000     -1.0759     0.0759

   a = -1:            -0.5000     -0.5713     0.0713

   a = +0:            -0.2500     -0.3172     0.0672
```

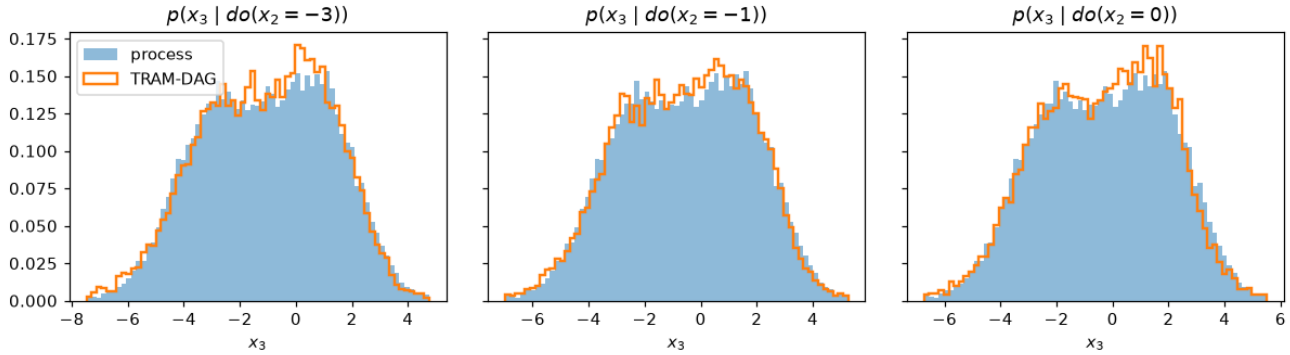


Figure 6: png

The bound leaves room for another machine and another draw.

```
assert max(errors) < 0.15, f"interventional mean off by {max(errors):.4f}"
```

5. Rung 3: individual counterfactuals

Take 1,000 held-out individuals. Abduction inverts the flow and recovers the latent noise of each one. That noise is everything about the individual that the model does not attribute to the parents. Pushing it back through the flow must return the observed row, which is the first check.

Action and prediction then rerun each individual under $do(x_1 = 0)$ with the same noise. The process keeps its own noise, so the true individual counterfactual is known and the flow can be scored per individual. Real data never permits this check, because only one of the two outcomes exists.

```
lat = draw_latents(1_000, np.random.default_rng(7))
factual = simulate(lat)
cf_true = simulate(lat, do={"x1": 0.0})

u = flow.abduct(factual)
recon_err = float(np.abs(flow.sample(u=u).to_numpy() - factual.to_numpy()).max())
print(f"abduction then reconstruction: max |error| = {recon_err:.2e}")
```

```
abduction then reconstruction: max |error| = 5.50e-06
```

Abduction inverts a monotone transform in float32. The bound is a precision bound, not a statistical one.

```
assert recon_err < 1e-4, f"abduction did not round-trip: {recon_err:.2e}"

cf_flow = flow.sample(do={"x1": 0.0}, u=u)
fig, axes = plt.subplots(1, 2, figsize=(9, 3.6))
for ax, c in zip(axes, ["x2", "x3"], strict=True):
    r = float(np.corrcoef(cf_true[c], cf_flow[c])[0, 1])
    ax.scatter(cf_true[c], cf_flow[c], s=4, alpha=0.4)
    lims = [cf_true[c].min(), cf_true[c].max()]
    ax.plot(lims, lims, "k--", lw=1)
    ax.set_title(f"${c[0]}_{c[1]}$ under $do(x_1{{=}}0)$   (r = {r:.4f})")
    ax.set_xlabel("true, shared noise")
    ax.set_ylabel("TRAM-DAG")
    assert r > 0.99, f"counterfactual {c} correlates only {r:.4f} with the truth"
fig.suptitle("L3: individual counterfactuals, scored one unit at a time")
fig.tight_layout()
plt.show()
```

What this notebook checked

rung	call	checked against
L1	`flow.sample(n)`	the correlation matrix of the data
L2	`flow.sample(n, do=...)`	the analytic mean $-0.25 + 0.25a$
L3	`flow.abduct(df)`, then `flow.sample(do=..., u=u)`	the true per-unit counterfactual

Where to go next:

- `docs/model.md`, the model and its notation.
- `docs/interpretation.md`, reading a fitted model, including ordinal nodes and log-odds ratios.
- `notebooks/training_strategies.py`, the fitting API and every shipped strategy.
- `notebooks/classical_fit_tram_dag.py`, the agreement with `statsmodels` and R.

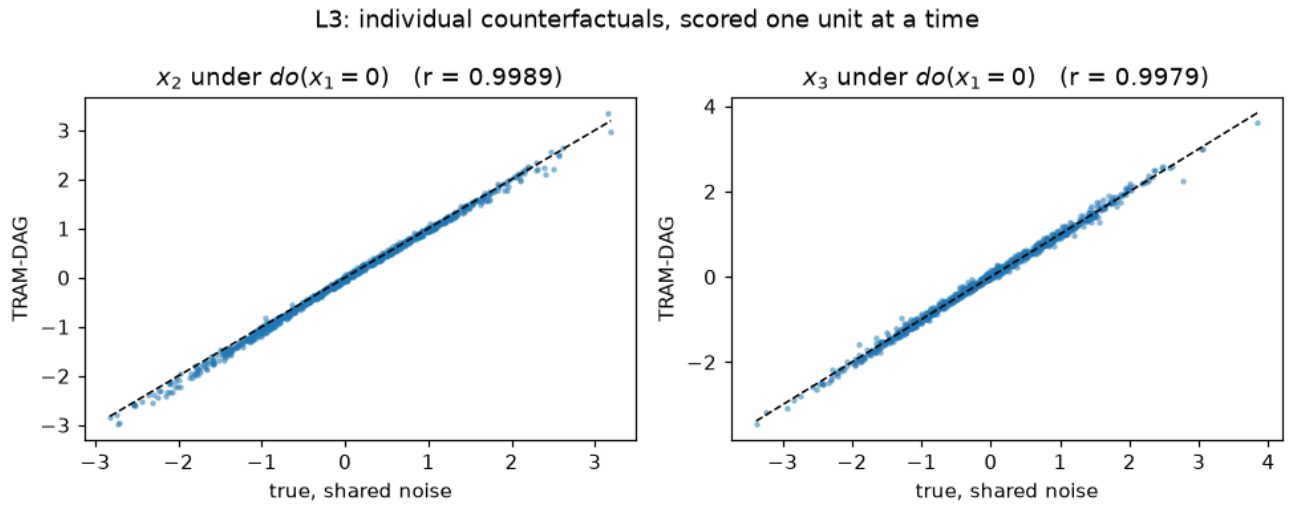


Figure 7: png

- `notebooks/varying_coefficients.py`, effects that vary with a covariate.
- repo · paper

```
elapsed = time.perf_counter() - t_total
print(f"whole notebook: {elapsed:.0f}s")
assert elapsed < 300, f"the notebook took {elapsed:.0f}s, over its five-minute claim"
```

```
whole notebook: 22s
```

Per-node or joint fit: a benchmark

The joint likelihood of a TRAM-DAG is a sum of per-node terms with disjoint parameters. So `fit` (one optimizer and one loop per node) and `fit_jointly` (one optimizer and one loop for all nodes) have the same optimum. They differ in the path to it: the batches, the step sizes, and what a schedule or a stop rule reads.

This notebook runs both on one process with a known truth, in six arms: per node in series, per node in processes (`n_jobs`), and jointly, each with a constant learning rate and with a plateau schedule. The per-node plateau reads the node's own validation NLL. The joint plateau reads the summed one, as the reference implementation does (deviation D5 in `docs/paper-replication.md`). The plateau has patience 10 instead of 49, for the short budget.

Each arm reports the wall-clock seconds, the epoch at which the summed validation NLL first comes within `EPS` of its best, the best summed validation NLL, and the error of one intervention against the truth.

```
import time
from collections import defaultdict

import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import torch

from tramdag import CausalFlowDAG, ContinuousNode, I
from tramdag.callbacks import Callback, EarlyStopping
```

```
plt.rcParams["figure.dpi"] = 110
torch.set_num_threads(1)


def claim(text, value, bound, fmt="{:.3f}"):
    """Assert a computed quantity stays inside its bound, and print both."""
    ok = "ok  " if value <= bound else "FAIL"
    print(f"{ok}{text}: {fmt.format(value)} <= {fmt.format(bound)}")
    assert value <= bound, f"{text}: {fmt.format(value)} exceeds {fmt.format(bound)}"
```

1. The process

The bimodal VACA triangle of the Colab demo: x_1 is a two-component mixture, $x_2 = -x_1 + \varepsilon_2$, $x_3 = x_1 + 0.25\,x_2 + \varepsilon_3$. Under $do(x_2 = a)$, $x_3 = x_1 + 0.25\,a + \varepsilon_3$, so the true interventional distribution is a fresh draw of the process with x_2 clamped.

```
def sample_dgp(n, seed, do=None):
    """Draw n rows of the VACA triangle, optionally with x2 clamped."""
    rng = np.random.default_rng(seed)
    mix, a, b, e2, e3 = rng.uniform(size=n), *rng.normal(size=(4, n))
    x1 = np.where(mix < 0.5, -2.0 + np.sqrt(1.5) * a, 1.5 + b)
    x2 = np.full(n, float(do["x2"])) if do else -x1 + e2
    x3 = x1 + 0.25 * x2 + e3
    return pd.DataFrame({"x1": x1, "x2": x2, "x3": x3})


def w1(a, b):
    """Give the 1-Wasserstein distance of two samples of equal size."""
    return float(np.mean(np.abs(np.sort(a) - np.sort(b))))


N_TRAIN, N_VAL = 8_000, 2_000
df = sample_dgp(N_TRAIN + N_VAL, seed=0)
train, val = df.iloc[:N_TRAIN], df.iloc[N_TRAIN:]
DO = {"x2": -1.0}
truth = sample_dgp(20_000, seed=1, do=DO)["x3"].to_numpy()

SPEC = {
    "x1": ContinuousNode(),
    "x2": ContinuousNode(I("x1")),
    "x3": ContinuousNode(I("x1", "x2")),
}
```

2. The arms

Every arm gets the same budget and `EarlyStopping()` without patience, so each fit runs all epochs and keeps its best weights. Torch runs on one thread, so each serial arm and each `n_jobs` worker uses one core. The plateau arms add torch's `ReduceLROnPlateau` through a callback. The callback reads the validation NLL of what it hooks: one node under `fit`, the summed NLL of the flow under `fit_jointly`.

```
EPOCHS = 100
SEEDS = [0, 1, 2]
EPS = 0.01
KW = dict(epochs=EPOCHS, batch_size=512, learning_rate=1e-2, validation_data=val)
```

```
class Plateau(Callback):
    """Cut the learning rate tenfold once the validation NLL stalls."""

    def on_fit_begin(self, target, optimizer):
        """Attach a fresh schedule to this fit's optimizer."""
        self.schedule = torch.optim.lr_scheduler.ReduceLROnPlateau(
            optimizer, factor=0.1, patience=10
        )

    def on_epoch_end(self, target, epoch, optimizer):
        """Step the schedule on the node's or the flow's validation NLL."""
        last = target.history"val"
        self.schedule.step(sum(last.values()) if isinstance(last, dict) else last)


def callbacks(plateau):
    """Give the callbacks of one fit: a fresh plateau when asked, plus EarlyStopping."""
    return [Plateau(), EarlyStopping()] if plateau else [EarlyStopping()]


ARMS = {
    "per node": lambda f, p, s: f.fit(
        train, seed=s, callbacks=defaultdict(lambda: callbacks(p)), **KW
    ),
    "per node, n_jobs=3": lambda f, p, s: f.fit(
        train, seed=s, n_jobs=3, callbacks=defaultdict(lambda: callbacks(p)), **KW
    ),
    "joint": lambda f, p, s: f.fit_jointly(train, seed=s, callbacks=callbacks(p), **KW),
}
```

3. The runs

Each arm runs once per seed. The seed sets the initial weights and the minibatch order. Every node runs all epochs, so the per-node curves sum epoch by epoch.

```
rows, curves = [], {}
for name, run in ARMS.items():
    for plateau in (False, True):
        arm = name + (" + plateau" if plateau else "")
        for seed in SEEDS:
            flow = CausalFlowDAG(SPEC, seed=seed)
            start = time.perf_counter()
            run(flow, plateau, seed)
            seconds = time.perf_counter() - start
            curve = np.array([sum(v.values()) for v in flow.history["val"]])
            sample = flow.sample(len(truth), do=DO, seed=seed)["x3"].to_numpy()
            final_lr = max(lr[0] for lr in flow.history"lr".values())
            rows.append(
                {
                    "arm": arm,
                    "seed": seed,
                    "seconds": seconds,
                    "epochs_to_eps": int(np.argmax(curve <= curve.min() + EPS)) + 1,
                    "best_val_nll": curve.min(),
                    "w1_do": w1(sample, truth),
```

```
                    "final_lr": final_lr,
                }
            )
            curves[arm, seed] = curve
        print(f"{arm:28s} done")

runs = pd.DataFrame(rows)
board = runs.groupby("arm", sort=False).agg(["mean", "std"])
board = board.drop(columns="seed")
print(board.to_string(float_format=lambda v: f"{v:.3g}"))
```

```
per node                     done


per node + plateau           done


per node, n_jobs=3           done


per node, n_jobs=3 + plateau done


joint                        done


joint + plateau              done
                    seconds     epochs_to_eps     best_val_nll       w1_do     final_lr
                      mean   std       mean  std       mean    std  mean   std    mean    std
arm
per node                15.7  0.662        35    1      4.98 0.00139 0.0947 0.0463    0.01      0
per node + plateau      15.3 0.0318       34.7 0.577      4.98 0.00305  0.117  0.064  3.7e-
06 5.47e-06
per node, n_jobs=3      7.68 0.0183        35    1      4.98 0.00139 0.0947 0.0463    0.01      0
per node, n_jobs=3 + plateau   7.43  0.156       34.7 0.577      4.98 0.00305  0.117  0.064  3.7e-
06 5.47e-06
joint                  14.5 0.0547        35    1      4.98 0.00139 0.0744 0.0434    0.01      0
joint + plateau           14.6 0.0837        35    1      4.98 0.00213 0.0773 0.0271   1e-
06        0
```

4. Convergence

The summed validation NLL per epoch, seed 0, one panel per schedule.

```
fig, axes = plt.subplots(1, 2, figsize=(9, 3.4), sharey=True)
for ax, plateau in zip(axes, (False, True), strict=True):
    for i, name in enumerate(ARMS):
        arm = name + (" + plateau" if plateau else "")
        ax.plot(np.arange(1, EPOCHS + 1), curves[arm, 0], color=f"C{i}", label=name)
    ax.set_title("plateau schedule" if plateau else "constant learning rate")
    ax.set_xlabel("epoch")
axes[0].set_ylabel("summed validation NLL")
best = runs["best_val_nll"].min()
axes[0].set_ylim(best - 0.02, best + 0.3)
axes[1].legend()
fig.tight_layout()
```

```
plt.show()
```

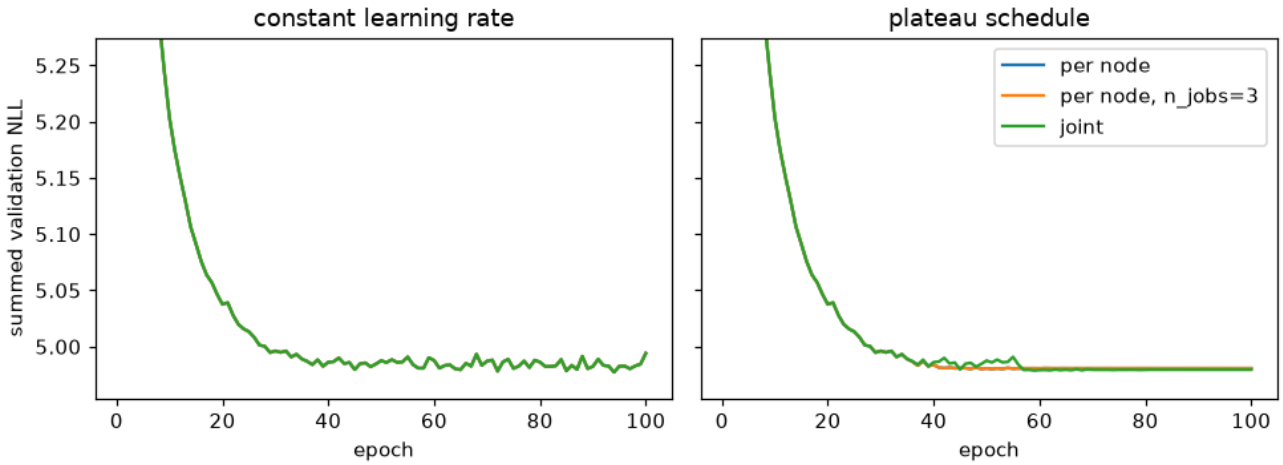


Figure 8: png

5. What the numbers say

The checks hold on any machine. The seconds change with the machine, so the notebook prints them and does not assert them. The `n_jobs` arm runs three processes, the other arms one.

The optimum is the same, so all arms reach the same best validation NLL. The gap in intervention error between the arms stays within two seed standard deviations. All arms come within `EPS` of their best after a similar number of epochs. Every plateau arm ends with the rate of each node cut, so the schedule acts.

```
means = runs.groupby("arm", sort=False).mean()
claim(
    "spread of the mean best validation NLL over the arms",
    means["best_val_nll"].max() - means["best_val_nll"].min(),
    0.003,
)
spread = runs.groupby("arm")["w1_do"].std().max()
claim(
    "spread of the mean W1 error under do(x2=-1) over the arms, per seed std",
    (means["w1_do"].max() - means["w1_do"].min()) / spread,
    2.0,
)
claim(
    "largest over smallest mean epochs to within EPS of the best",
    means["epochs_to_eps"].max() / means["epochs_to_eps"].min(),
    1.3,
)
claim(
    "largest final learning rate of a plateau arm",
    runs.loc[runs["arm"].str.endswith("plateau"), "final_lr"].max(),
    1e-4,
    fmt="{:.1e}",
)
```

```
ok  spread of the mean best validation NLL over the arms: 0.001 <= 0.003
ok  spread of the mean W1 error under do(x2=-1) over the arms, per seed std: 0.673 <= 2.000
```

```
ok  largest over smallest mean epochs to within EPS of the best: 1.010 <= 1.300
ok  largest final learning rate of a plateau arm: 1.0e-05 <= 1.0e-04
```

Serial and forked per-node fits with the same seed train the same nodes on the same batches. On one thread their curves agree to 1e-6.

```
claim(
    "largest gap of the serial and forked per-node curves",
    max(
        np.abs(curves["per node", s] - curves["per node, n_jobs=3", s]).max()
        for s in SEEDS
    ),
    1e-6,
    fmt="{:.2e}",
)
```

```
ok  largest gap of the serial and forked per-node curves: 0.00e+00 <= 1.00e-06
```

Which one, when

The section “Which recipe” of `docs/fitting.md` holds the decision table.

Training strategies: driving `fit` from the API side

`fit` runs one minibatch Adam loop per node and keeps the final weights. It owns the loop, the per-epoch validation score and the progress output. The caller controls the rest through `optimizer=` and `callbacks=`.

This notebook runs every shipped strategy on one small workload and ends with a runtime scoreboard. The table in `docs/fitting.md` tells which strategy to pick.

```
import time
from collections import defaultdict
from pathlib import Path

import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import torch

from tramdag import CausalFlowDAG, ContinuousNode, I, Node
from tramdag.callbacks import Callback, EarlyStopping
from tramdag.plots import plot_training

plt.rcParams["figure.dpi"] = 110

REPO = next(
    p for p in [Path.cwd(), *Path.cwd().parents] if (p / "pyproject.toml").exists()
)
```

The data is the tracked VACA sample of `notebooks/data/`, from the process of the Colab demo. The notebook runs from the repository root or from `notebooks/`.

```
df = pd.read_csv(REPO / "notebooks" / "data" / "vaca.csv")
train, val = df.iloc[:900], df.iloc[900:]

SPEC = {
    "x1": ContinuousNode(),
```

```
    "x2": ContinuousNode(I("x1")),
    "x3": ContinuousNode(I("x1", "x2")),
}
CEILING = 600
scoreboard = []


def build():
    """Give a fresh unfitted flow, seeded so every recipe starts identically."""
    torch.manual_seed(0)
    return CausalFlowDAG(SPEC)


def record(name, flow, seconds):
    """Add one finished recipe to the scoreboard and print its line."""
    nll = sum(flow.nll(val).values())
    node_epochs = sum(len(nd.history["train"]) for nd in flow.nodes.values())
    scoreboard.append(
        {
            "strategy": name,
            "val_nll": nll,
            "node_epochs": node_epochs,
            "seconds": seconds,
        }
    )
    print(
        f"{name:22s} val NLL {nll:.4f}   {node_epochs:4d} node-epochs   {seconds:5.1f}s"
    )
    return nll


print(f"{len(train)} train rows, {len(val)} validation rows")
```

```
900 train rows, 100 validation rows
```

1. What `fit` records without any callback

With `validation_data=`, `fit` scores the validation NLL of each node once per epoch. `flow.history` holds one dict of per-node values per epoch, for the keys that the docstring of `CausalFlowDAG.history` lists. It is a view over the per-node `flow.nodes[name].history`.

```
flow = build()
flow.fit(train, epochs=20, batch_size=256, validation_data=val)

print("history keys:", sorted(flow.history))
print("epochs recorded:", len(flow.history["train"]))
print("per-node train NLL, last epoch:")
for node, value in flow.history"train".items():
    print(f"    {node}: {value:.4f}")
print("learning rate, last epoch:", flow.history"lr")

assert len(flow.history["train"]) == len(flow.history["val"]) == 20
assert set(flow.history"train") == set(SPEC)
```

```
history keys: ['lr', 'train', 'val', 'val_epoch']
epochs recorded: 20
per-node train NLL, last epoch:
```

```
    x1: 2.6743
    x2: 1.4055
    x3: 1.4272
learning rate, last epoch: {'x1': {0: 0.01}, 'x2': {0: 0.01}, 'x3': {0: 0.01}}
```

Without `validation_data=` there is no `val` key, and `EarlyStopping` refuses rather than guesses.

```
bare = build()
bare.fit(train, epochs=5, batch_size=256)
print("history keys without validation:", sorted(bare.history))
assert "val" not in bare.history

try:
    build().fit(train, epochs=5, batch_size=256, callbacks=EarlyStopping())
except RuntimeError as err:
    print("\nEarlyStopping without validation refuses:\n   ", err)
else:
    raise AssertionError("EarlyStopping should refuse without validation data")
```

```
history keys without validation: ['lr', 'train']

EarlyStopping without validation refuses:
   EarlyStopping reads node.history['val']; pass validation_data= or validation_split= to the fit ca
```

2. Plain Adam

One rate, one loop, and the weights of the last epoch: the baseline for the other recipes. An all-LS model trained this way to convergence reaches the maximum-likelihood estimate (`classical_fit_tram_dag.py`).

```
flow = build()
t0 = time.perf_counter()
flow.fit(train, epochs=CEILING, batch_size=256, learning_rate=1e-2, validation_data=val)
nll_plain = record("plain Adam", flow, time.perf_counter() - t0)
history_plain = [sum(d.values()) for d in flow.history["val"]]
```

```
plain Adam             val NLL 5.0084   1800 node-epochs    20.0s
```

3. Two phases

A second `fit` call continues the training with a new optimizer. A second call at a lower rate is thus a coarse-then-fine schedule. The history accumulates over the calls.

```
flow = build()
t0 = time.perf_counter()
flow.fit(train, epochs=200, batch_size=256, learning_rate=1e-2, validation_data=val)
after_coarse = sum(flow.nll(val).values())
flow.fit(train, epochs=100, batch_size=256, learning_rate=1e-3, validation_data=val)
nll_two_phase = record("two phases", flow, time.perf_counter() - t0)

print(f"    after the coarse phase: {after_coarse:.4f}")
print(f"    after the fine phase:   {nll_two_phase:.4f}")
assert len(flow.history["train"]) == 300, "the second fit did not continue the history"
```

```
two phases             val NLL 4.9707    900 node-epochs    10.0s
    after the coarse phase: 4.9783
    after the fine phase:   4.9707
```

4. `EarlyStopping`: keep the best weights

A flexible model can reach its best validation score part way through and then drift. `EarlyStopping` keeps a copy of each node's best epoch and loads it at the end of that node's fit. After the fit, the validation score of each node equals the minimum over its epochs.

```
flow = build()
t0 = time.perf_counter()
stopper = EarlyStopping()
flow.fit(
    train,
    epochs=CEILING,
    batch_size=256,
    learning_rate=1e-2,
    validation_data=val,
    callbacks=stopper,
)
nll_best = record("EarlyStopping", flow, time.perf_counter() - t0)

restored = flow.nll(val)
for name, nd in flow.nodes.items():
    recorded = nd.history["val"]
    best = int(np.argmin(recorded)) + 1
    print(
        f"    {name}: best epoch {best} of {len(recorded)}, "
        f"minimum {min(recorded):.4f}, last {recorded[-1]:.4f}"
    )
    assert abs(restored[name] - min(recorded)) < 1e-4, (
        f"the restored weights of {name} do not match its best recorded epoch"
    )
```

```
EarlyStopping          val NLL 4.9465   1800 node-epochs    20.1s
    x1: best epoch 483 of 600, minimum 2.0991, last 2.1029
    x2: best epoch 174 of 600, minimum 1.4721, last 1.5049
    x3: best epoch 327 of 600, minimum 1.3754, last 1.4007
```

5. `EarlyStopping(patience=)`: stop each node on its own

With `patience`, a node also stops once its best epoch is that many epochs old, so each node stops at its own epoch. A dict by node name gives each node its own instance; a `defaultdict` makes one for every node.

```
flow = build()
t0 = time.perf_counter()
stoppers = defaultdict(lambda: EarlyStopping(patience=25))
flow.fit(
    train,
    epochs=CEILING,
    batch_size=256,
    learning_rate=1e-2,
    validation_data=val,
    callbacks=stoppers,
)
record("EarlyStopping(25)", flow, time.perf_counter() - t0)

stopped = {name: len(nd.history["train"]) for name, nd in flow.nodes.items()}
for name, spent in stopped.items():
```

```
    best = stoppers[name].best_epoch
    print(f"    {name}: stopped after {spent} of {CEILING} epochs, best was {best}")
    assert spent < CEILING, f"patience never triggered for {name}"
    assert spent - best >= 25
```

```
EarlyStopping(25)      val NLL 4.9775    343 node-epochs     3.7s
    x1: stopped after 218 of 600 epochs, best was 193
    x2: stopped after 82 of 600 epochs, best was 57
    x3: stopped after 43 of 600 epochs, best was 18
```

`flow.history` repeats the last entry of a node that stopped earlier, so the summed curve stays defined. `plot_training` marks the stop epochs and the best epochs.

```
plot_training(flow, stops=True)
plt.show()
```

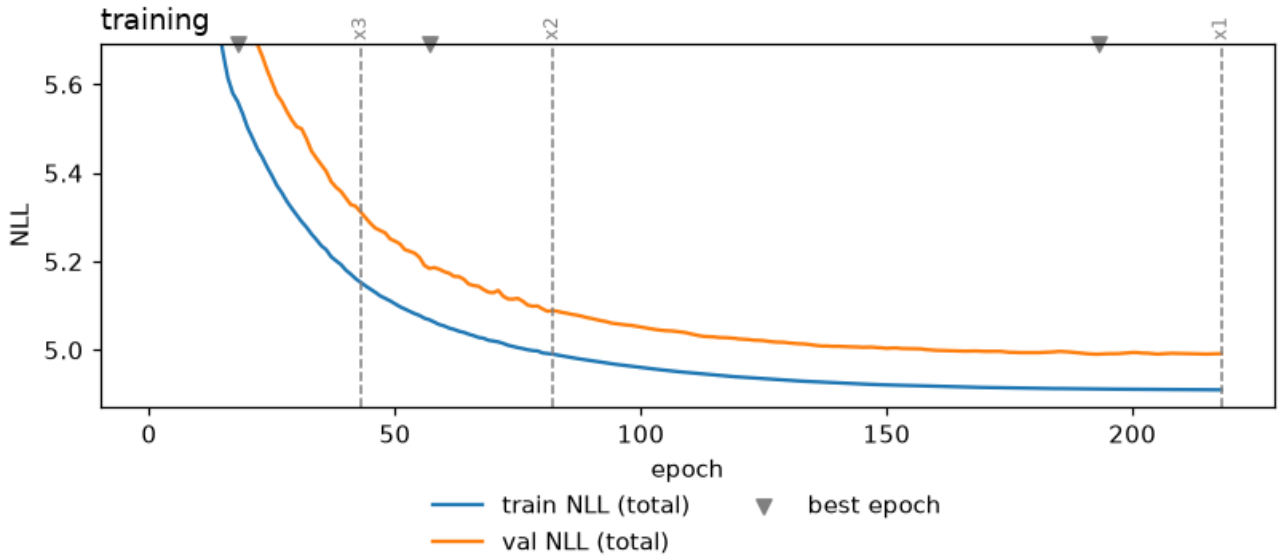


Figure 9: png

6. Writing your own

`docs/fitting.md` gives the callback hooks. `Plateau` builds torch's `ReduceLROnPlateau` on the optimizer of each node and steps it on `node.history["val"]`.

```
class Plateau(Callback):
    """One torch ReduceLROnPlateau per node, driven by fit's own score."""

    def __init__(self, factor=0.3, patience=10):
        self.factor, self.patience = factor, patience
        self.scheduler = None

    def on_fit_begin(self, node, optimizer):
        """Build the scheduler here: the node's optimizer exists only now."""
        self.scheduler = torch.optim.lr_scheduler.ReduceLROnPlateau(
            optimizer, factor=self.factor, patience=self.patience
        )

    def on_epoch_end(self, node, epoch, optimizer):
```

```
        """Step the scheduler, and stop once the rate has bottomed out."""
        self.scheduler.step(node.history"val")
        return optimizer.param_groups0 < 1e-5


flow = build()
t0 = time.perf_counter()
flow.fit(
    train,
    epochs=CEILING,
    batch_size=256,
    learning_rate=1e-2,
    validation_data=val,
    callbacks=Plateau(),
)
record("Plateau", flow, time.perf_counter() - t0)

for name, rates in flow.history"lr".items():
    print(f"    {name}: rate went 1.0e-02 -> {rates[0]:.1e}")
    assert rates[0] < 1e-2, f"the scheduler never decayed the rate of {name}"
```

```
Plateau                 val NLL 4.9907    500 node-epochs     5.4s
    x1: rate went 1.0e-02 -> 7.3e-06
    x2: rate went 1.0e-02 -> 7.3e-06
    x3: rate went 1.0e-02 -> 7.3e-06
```

A callable in a list is an `on_epoch_end` hook. This one records the validation NLL of each node after every epoch.

```
trace = []
build().fit(
    train,
    epochs=5,
    batch_size=256,
    callbacks=[lambda node, epoch, opt: trace.append((node.name, node.nll(val)))],
)
print(f"{len(trace)} entries, the first: {trace[0]}")
assert len(trace) == 5 * len(SPEC)
```

```
15 entries, the first: ('x1', 3.561253309249878)
```

`optimizer=` takes a factory `f(node)`. This one gives the networks weight decay in their own parameter group. `flow.history["lr"]` shows both groups of every node.

```
def adam_with_net_decay(node):
    """Adam with weight decay on the network parameters only."""
    groups = node.parameter_groups()
    rest = groups["intercept"] + groups["linear"]
    nets = {"params": groups["networks"], "weight_decay": 1e-4}
    groups = [{"params": rest}, nets]
    return torch.optim.Adam(groups, lr=1e-2)


flow = build()
flow.fit(train, epochs=20, batch_size=256, optimizer=adam_with_net_decay)
print("rates per group, last epoch:", flow.history"lr")
assert all(set(rates) == {0, 1} for rates in flow.history"lr".values())
```

```
rates per group, last epoch: {'x1': {0: 0.01, 1: 0.01}, 'x2': {0: 0.01, 1: 0.01}, 'x3': {0: 0.01, 1: 0.
```

7. One node alone

A node is a TRAM regression of one variable on its parents and fits without the DAG. It takes its name, its node spec and a parent schema, `{parent: "continuous" | n_levels}`.

```
x3 = Node("x3", SPEC["x3"], {"x1": "continuous", "x2": "continuous"})
x3.fit(
    train[["x1", "x2", "x3"]],
    epochs=CEILING,
    validation_data=val[["x1", "x2", "x3"]],
    callbacks=EarlyStopping(patience=25),
)
print(f"x3 alone: {len(x3.history['train'])} epochs, val NLL {x3.nll(val):.4f}")
assert abs(x3.nll(val) - min(x3.history["val"])) < 1e-4
```

```
x3 alone: 178 epochs, val NLL 1.3967
```

8. The scoreboard: a runtime comparison

Each row gives the wall-clock seconds, the node-epochs summed over the nodes, the validation NLL and its gap to the best recipe. A recipe that stops early wins on seconds only if its NLL stays near the best. The seconds apply to this machine only.

```
board = pd.DataFrame(scoreboard).set_index("strategy")
board["nll_gap"] = board["val_nll"] - board["val_nll"].min()
board["s_per_node_epoch"] = board["seconds"] / board["node_epochs"]
print(board.to_string(float_format=lambda v: f"{v:.4f}"))

fig, ax = plt.subplots(figsize=(7, 3.4))
ax.scatter(board["seconds"], board["nll_gap"])
for name, row in board.iterrows():
    ax.annotate(
        name,
        (row["seconds"], row["nll_gap"]),
        xytext=(4, 4),
        textcoords="offset points",
        fontsize=8,
    )
ax.margins(x=0.15)
ax.set_xlabel("wall-clock seconds")
ax.set_ylabel("validation NLL above the best recipe")
ax.set_title("cost against quality, one workload")
fig.tight_layout()
plt.show()

assert nll_best <= nll_plain + 1e-6, (
    f"EarlyStopping scored {nll_best:.4f} against plain Adam's {nll_plain:.4f}"
)
```

```
                   val_nll  node_epochs  seconds  nll_gap  s_per_node_epoch
strategy
plain Adam          5.0084         1800  20.0488   0.0618            0.0111
two phases          4.9707          900   9.9789   0.0241            0.0111
EarlyStopping       4.9465         1800  20.0551   0.0000            0.0111
EarlyStopping(25)   4.9775          343   3.7151   0.0309            0.0108
Plateau             4.9907          500   5.4025   0.0442            0.0108
```

The restored `EarlyStopping` score lies below the minimum of the plain Adam curve: each node

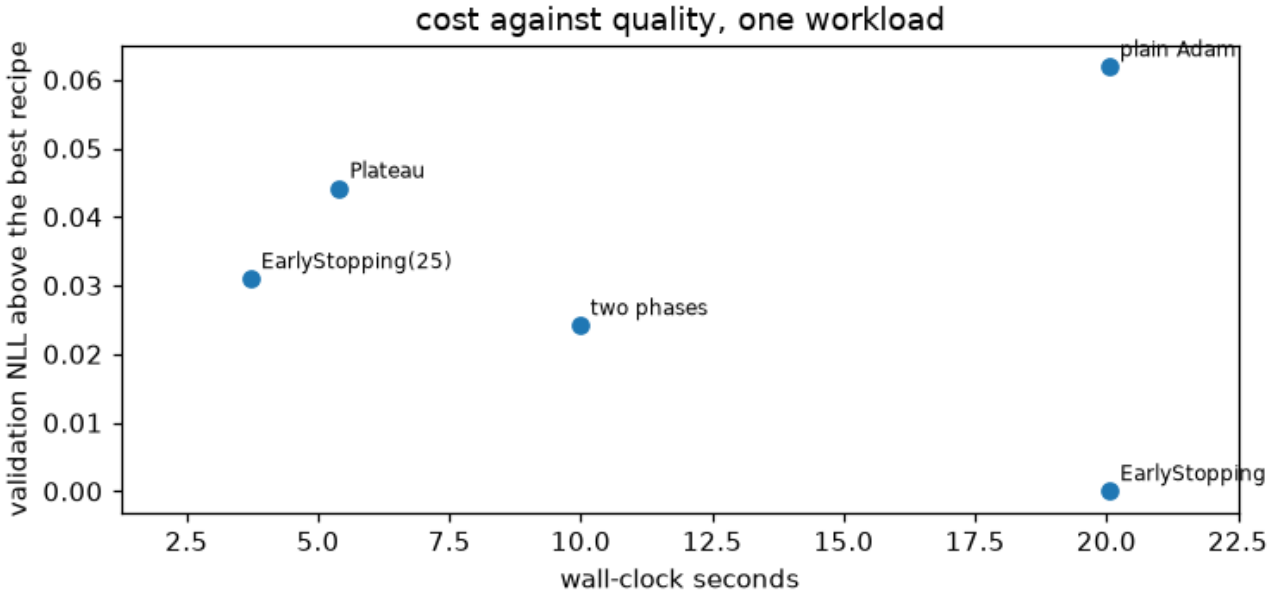


Figure 10: png

restores its own best epoch, while the summed curve has one epoch for all nodes.

```
assert nll_best <= min(history_plain) + 1e-6
fig, ax = plt.subplots(figsize=(7, 3.4))
ax.plot(np.arange(1, len(history_plain) + 1), history_plain, label="plain Adam")
ax.axhline(nll_best, ls="--", lw=1, color="C1", label="EarlyStopping, restored")
ax.set_xlabel("epoch")
ax.set_ylabel("validation NLL (total)")
ax.set_ylim(min(nll_best, min(history_plain)) - 0.02, min(history_plain) + 0.4)
ax.legend()
ax.set_title("keeping the best epoch against keeping the last")
fig.tight_layout()
plt.show()
```

Which one, when

The decision table lives in `docs/fitting.md`.

Heterogeneous treatment effects: the `VC` term

Most causal questions are not "does the treatment help?" but "**whom** does it help, and by how much?". A TRAM-DAG answers that with a **varying-coefficient** term, which gives one edge of the DAG a treatment effect that depends on covariates:

$$\text{shift} \;+\!=\; \beta(\text{modifiers}) \cdot x_t, \qquad \beta(x) = \beta_0 + b_\Theta(x)$$

`docs/varying-coefficients.md` states the model, its penalty and its centering; this notebook runs it.

The notebook scores the recovered `beta(x)` against a known effect function. It then shows what a plain flexible shift does not give: a read-out you can plot, and a **propensity-centered** variant that survives confounding.

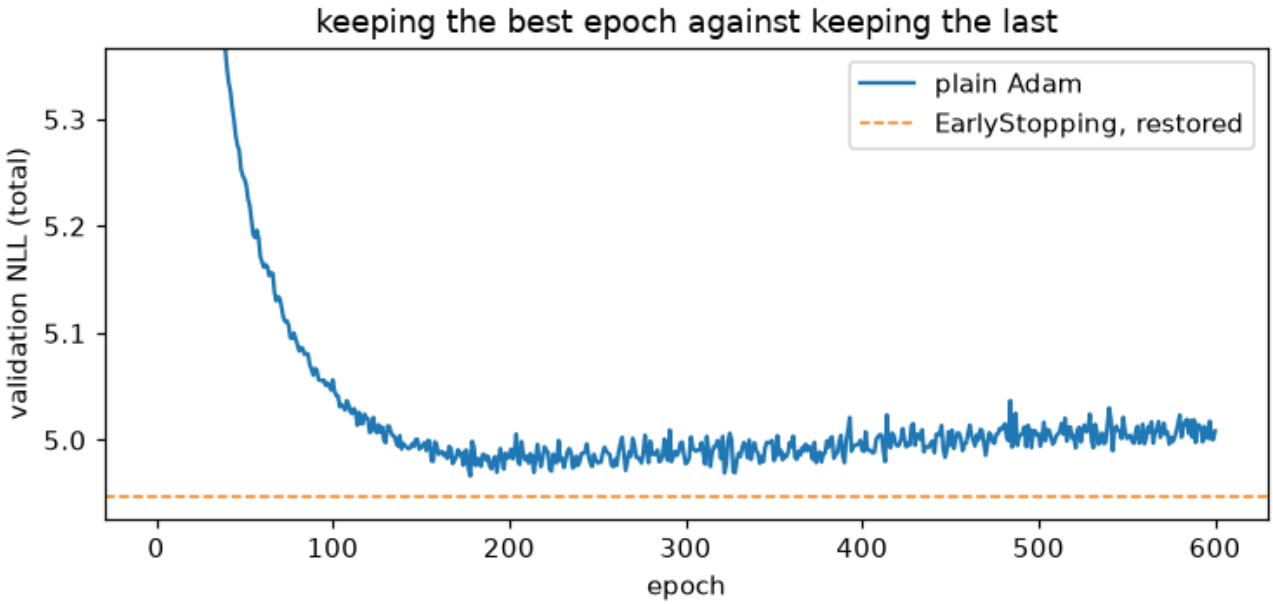


Figure 11: png

```
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd

from tramdag import CS, LS, SI, VC, CausalFlowDAG, ContinuousNode, OrdinalNode
from tramdag.callbacks import EarlyStopping
from tramdag.plots import plot_varying_coef

plt.rcParams["figure.dpi"] = 110
```

1. A DGP with a known effect function

Three covariates, a **confounded** treatment (`X1` and `X2` push assignment), and an outcome whose conditional is exactly a logistic-latent flow:

$$h(y) + g(x) + \beta(x)\, t = U, \qquad U \sim \text{Logistic}(0, 1)$$

with $h(y) = 2y$, a nonlinear prognostic part $g(x) = \frac{1}{2}X_1^2 + X_2 - \frac{1}{2}X_3$, and the effect function

$$\beta(x) = -1 + 0.8\, X_2 - 0.6\, X_3.$$

`X2` is deliberately *both* a confounder and an effect modifier. In this case a naive estimate fails hardest. The simulator writes `beta` as the same expression that `true_beta` reports.

```
B0, B2, B3 = -1.0, 0.8, -0.6


def true_beta(df):
    """The effect function the model has to recover, on the latent scale."""
    return B0 + B2 * df["X2"].to_numpy() + B3 * df["X3"].to_numpy()
```

```
def simulate(n, seed):
    rng = np.random.default_rng(seed)
    x1, x2, x3 = (rng.normal(size=n) for _ in range(3))
    t = (rng.logistic(size=n) > -(0.4 * x1 + 0.4 * x2)).astype(float)
    g = 0.5 * x1**2 + x2 - 0.5 * x3
    beta = B0 + B2 * x2 + B3 * x3
    y = (rng.logistic(size=n) - g - beta * t) / 2.0
    return pd.DataFrame({"X1": x1, "X2": x2, "X3": x3, "T": t, "Y": y})


train, val, test = simulate(4500, 1), simulate(500, 2), simulate(2000, 3)
print(f"treated fraction: {train['T'].mean():.2f}")
print(
    f"true effect ranges from {true_beta(test).min():+.2f} to {true_beta(test).max():+.2f}"
)
```

```
treated fraction: 0.48
true effect ranges from -4.41 to +2.29
```

2. Which covariates modify the effect? (`effect_modifier_scan`)

Before you choose the modifiers, ask the data. Fit the **simple all-LS model**, which takes seconds and is deterministic. Then look at the per-row scores of the treatment coefficient. With a constant effect, these scores show no pattern along any covariate. The scan measures that pattern per covariate and ranks the covariates.

```
def cheap_all_ls():
    return {
        "X1": ContinuousNode(),
        "X2": ContinuousNode(),
        "X3": ContinuousNode(),
        "T": OrdinalNode(2, LS("X1") + LS("X2")),
        "Y": ContinuousNode(LS("X1") + LS("X2") + LS("X3") + LS("T")),
    }


screen = CausalFlowDAG(cheap_all_ls(), seed=0)
screen.fit_classical(train)
scan = screen.effect_modifier_scan(train, "Y", t="T")
print(scan)
assert scan"flag"].all()
```

```
        stat       p_value  crit_5pct  flag
X2  4.164609  1.722864e-15     1.3581  True
X1  4.141297  2.537664e-15     1.3581  True
X3  3.161564  4.159703e-09     1.3581  True
```

`X2` and `X3` are the true modifiers, and both flag. `X1` flags too, but it does **not** enter $\beta(x)$. Here g contains $\frac{1}{2}X_1^2$, which the simple linear model cannot represent. `docs/scores.md` explains why a misspecified prognostic part flags too.

To show this, generate the data again with a *linear* prognostic X_1 and change nothing else. `X1` then does not flag, because the prognostic part is linear in `X1` and the simple model has the right form.

```
def simulate_linear_x1(n, seed):
    rng = np.random.default_rng(seed)
```

```
    x1, x2, x3 = (rng.normal(size=n) for _ in range(3))
    t = (rng.logistic(size=n) > -(0.4 * x1 + 0.4 * x2)).astype(float)
    g = 0.5 * x1 + x2 - 0.5 * x3
    y = (rng.logistic(size=n) - g - (B0 + B2 * x2 + B3 * x3) * t) / 2.0
    return pd.DataFrame({"X1": x1, "X2": x2, "X3": x3, "T": t, "Y": y})


well_specified = simulate_linear_x1(4500, 1)
screen2 = CausalFlowDAG(cheap_all_ls(), seed=0)
screen2.fit_classical(well_specified)
scan2 = screen2.effect_modifier_scan(well_specified, "Y", t="T")
print(scan2)
assert scan2"flag"].all()
assert not scan2"flag"
```

```
        stat       p_value  crit_5pct   flag
X2  4.608240  7.174333e-19     1.3581   True
X3  3.474392  6.545501e-11     1.3581   True
X1  0.553054  9.197166e-01     1.3581  False
```

The figure draws the scaled running sum B_j of the scan for each candidate (`docs/scores.md` explains it). The left panel uses the quadratic `X1`, the right panel the linear `X1`. `stat` is the largest distance of the path from zero, the dot on each line. If a path leaves the grey band, the scan flags its covariate.

```
CANDIDATES = {"X1": "#2a78d6", "X2": "#eb6834", "X3": "#1baf7a"}


def plot_cusum(ax, flow, df, title):
    """Draw each candidate's scaled running sum of the T scores, as the scan forms it."""
    psi = flow.scores(df, "Y")["T[1]"].to_numpy()
    n = len(psi)
    scan = flow.effect_modifier_scan(df, "Y", t="T")
    crit = scan["crit_5pct"].iloc[0]
    ax.axhspan(
        -crit, crit, color="0.93", lw=0, label=f"$|B_j| < {crit:.3f}$: no flag at 5%"
    )
    for c in (-crit, crit):
        ax.axhline(c, color="0.55", lw=1, ls="--")
    ax.axhline(0, color="0.75", lw=0.8)
    frac = np.arange(1, n + 1) / n
    for name, color in CANDIDATES.items():
        b = np.cumsum(psi[np.argsort(df[name].to_numpy(), kind="stable")])
        b /= psi.std() * np.sqrt(n)
        j = np.abs(b).argmax()
        ax.plot(frac, b, color=color, lw=2, label=name)
        ax.plot(frac[j], b[j], "o", ms=8, color=color, mec="white", mew=2)
        ax.annotate(
            f"{name}: stat {scan.loc[name, 'stat']:.2f}",
            (frac[j], b[j]),
            xytext=(0, 10 if b[j] > 0 else -16),
            textcoords="offset points",
            ha="center",
            color="0.2",
        )
    ax.set(
        title=title, xlabel="fraction of rows, sorted by the candidate", ylim=(-6, 5)
```

```
    )


fig, axes = plt.subplots(1, 2, figsize=(11, 4), sharey=True)
plot_cusum(axes[0], screen, train, "X1 quadratic in the outcome: X1 flags too")
plot_cusum(
    axes[1], screen2, well_specified, "X1 linear in the outcome: only X2, X3 flag"
)
axes[0].set_ylabel("scaled running sum $B_j$ of the T scores")
fig.legend(
    *axes[1].get_legend_handles_labels(), loc="lower center", ncol=4, frameon=False
)
fig.tight_layout(rect=(0, 0.07, 1, 1))
plt.show()
```

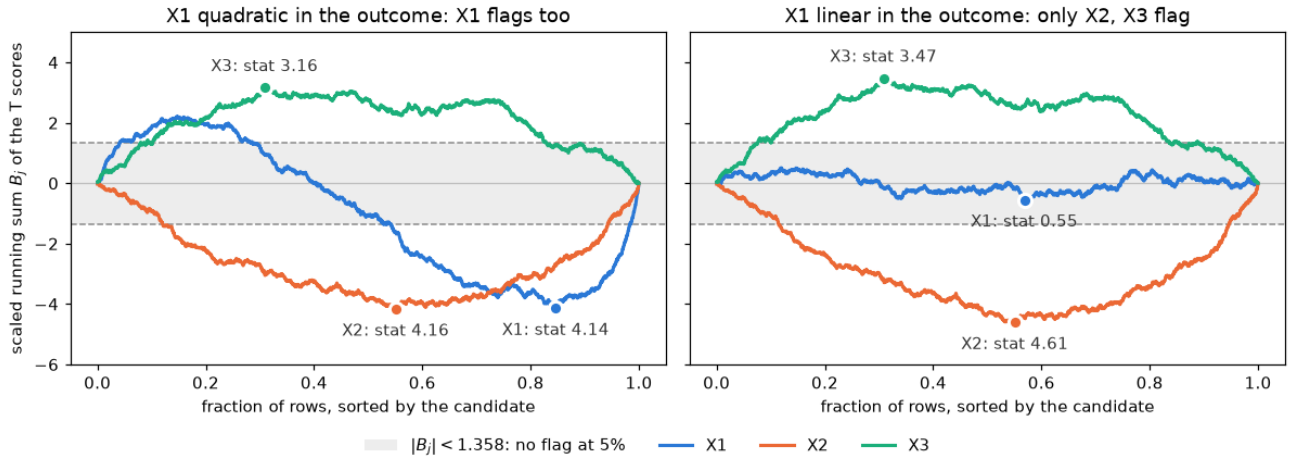


Figure 12: png

3. The spec: prognostic part and effect head are separate

`CS("X1", "X2", "X3")` absorbs the prognostic signal, as flexibly as necessary. `VC("X2", "X3", t="T")` carries the effect. `X2` and `X3` appear twice on purpose: prognostically through the shift, and as modifiers through the head. Only the treatment named by `t=` owns its edge.

```
spec = {
    "X1": ContinuousNode(),
    "X2": ContinuousNode(),
    "X3": ContinuousNode(),
    "T": OrdinalNode(2, LS("X1") + LS("X2")),
    "Y": ContinuousNode(CS("X1", "X2", "X3") + VC("X2", "X3", t="T")),
}
flow = CausalFlowDAG(spec, seed=0)
flow.fit(
    train,
    epochs=300,
    learning_rate=1e-2,
    batch_size=512,
    validation_data=val,
    seed=0,
    callbacks=EarlyStopping(),
```

```
)
print(flow.to_matrix())
```

```
   X1 X2 X3   T                          Y
X1           LS      CS['X1', 'X2', 'X3']
X2           LS  CS['X1', 'X2', 'X3']+VCm
X3               CS['X1', 'X2', 'X3']+VCm
T                                     VC
Y
```

In the meta-adjacency view (`docs/interpretation.md`) `VC` marks the treatment edge and `VCm` the modifiers, so `X2` visibly enters `Y` twice.

4. Reading the effect out

`varying_coef` gives $\beta(x)$ per row. It is **deterministic** and does not look at the outcome. It is a property of the fitted model, not of the rows' `Y` values.

```
beta_hat = flow.varying_coef(test, "Y")
beta_true = true_beta(test)
corr = float(np.corrcoef(beta_hat, beta_true)[0, 1])
beta0 = flow.nodes["Y"].shifts["T"].beta0.item()

print(f"corr(beta_hat, beta_true) = {corr:.3f}")
print(f"beta0 = {beta0:+.3f}   (true constant part {B0:+.1f})")
print(f"mean |error| = {np.abs(beta_hat - beta_true).mean():.3f}")
assert corr > 0.95, f"the fitted beta(x) correlates only {corr:.3f} with the truth"

fig, axes = plt.subplots(1, 2, figsize=(10, 3.8))
axes[0].scatter(beta_true, beta_hat, s=6, alpha=0.35)
lims = [min(beta_true.min(), beta_hat.min()), max(beta_true.max(), beta_hat.max())]
axes[0].plot(lims, lims, "k--", lw=1)
axes[0].set_xlabel(r"true $\beta(x)$")
axes[0].set_ylabel(r"fitted $\hat\beta(x)$")
axes[0].set_title(f"recovery, corr = {corr:.3f}")

plot_varying_coef(flow, test, "Y", by="X2", ax=axes[1])
x3_med = test["X3"].median()
grid = np.linspace(test["X2"].min(), test["X2"].max(), 2)
axes[1].plot(grid, B0 + B2 * grid + B3 * x3_med, "k--", lw=1)
along_x2 = test.assign(X3=x3_med)
gap_x2 = np.abs(flow.varying_coef(along_x2, "Y") - true_beta(along_x2)).max()
print(f"X3 at its median: max |beta_hat - beta_true| along X2 = {gap_x2:.3f}")
assert gap_x2 < 0.75, f"the fitted beta(X2) leaves the true line by {gap_x2:.3f}"
axes[1].legend(["fitted", r"$\beta = 0$", "true"])
axes[1].set_title(r"$\beta$ along $X_2$, $X_3$ at its median")
fig.tight_layout()
plt.show()
```

```
corr(beta_hat, beta_true) = 0.997
beta0 = -0.917   (true constant part -1.0)
mean |error| = 0.097
X3 at its median: max |beta_hat - beta_true| along X2 = 0.321
```

The right panel holds `X3` at its median, because the effect also depends on `X3`. Along `X2`, the fitted line follows the true line.

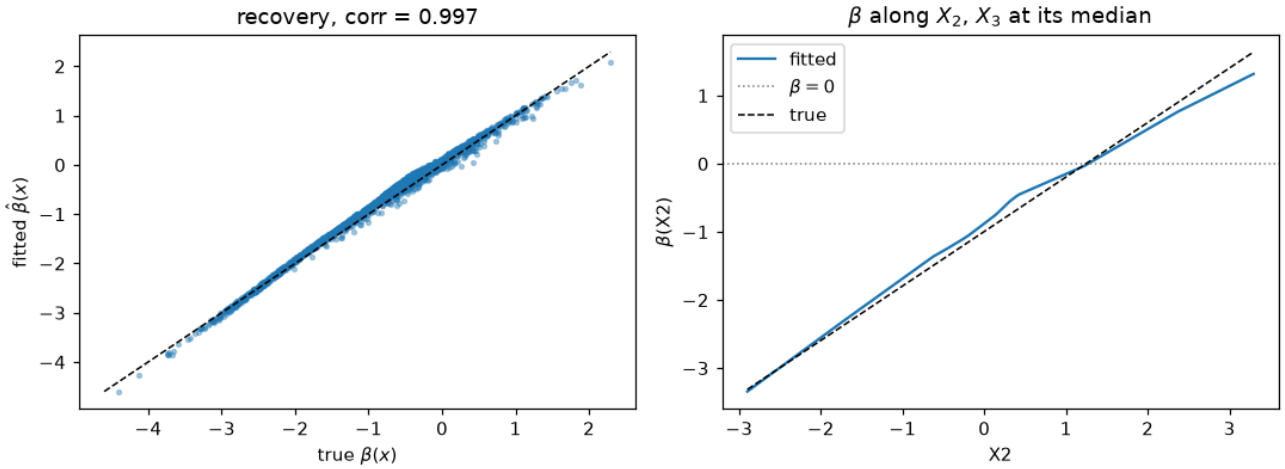


Figure 13: png

5. The read-out is an identity, not a summary

For a binary treatment, $\beta(x)$ *equals* the difference of the abducted latents between the two arms, with the outcome held fixed. The implementation must satisfy this definition to float precision:

```
u1 = flow.abduct(test.assign(T=1.0), seed=0)["Y"].to_numpy()
u0 = flow.abduct(test.assign(T=0.0), seed=0)["Y"].to_numpy()
gap = np.abs(beta_hat - (u1 - u0)).max()
print(f"max |beta(x) - (u(T=1) - u(T=0))| = {gap:.2e}")
assert gap < 1e-5, f"the read-out differs from the latent difference by {gap:.2e}"
```

```
max |beta(x) - (u(T=1) - u(T=0))| = 4.77e-07
```

6. Confounding: why `propensity=` exists

Everything above had a prognostic part flexible enough to absorb g. Real models are misspecified, and when the misfit correlates with **treatment assignment**, the effect head absorbs the confounding instead of the effect.

The configuration where this bites hardest is deliberately simple: one covariate, a strong propensity $e(x) = \sigma(2x)$, a constant true effect $\tau = -1$, and a quadratic prognostic part fitted with a linear term. `propensity="ps"` replaces t by $t - \hat{e}(x)$ with out-of-fold propensities from the training-frame column `ps`. `docs/varying-coefficients.md` says why they must be out of fold.

```
TAU = -1.0


def confounded(n, seed):
    rng = np.random.default_rng(seed)
    x = rng.normal(size=n)
    t = (rng.logistic(size=n) > -2.0 * x).astype(float)
    y = (rng.logistic(size=n) - 1.2 * x**2 - TAU * t) / 2.0
    return pd.DataFrame({"X": x, "T": t, "Y": y})


c_train, c_val, c_test = confounded(5400, 0), confounded(600, 7), confounded(3000, 1000)
```

Stage 1 of the centered design is the caller's job: the cross-fitted $P(T = 1 \mid X)$. A treatment model that never saw a fold predicts that fold, as double machine learning requires.

```
fold_id = np.random.default_rng(0).permutation(len(c_train)) % 5
e_oof = np.empty(len(c_train))
for j in range(5):
    t_spec = {
        "X": ContinuousNode(SI(transform="affine")),
        "T": OrdinalNode(2, LS("X")),
    }
    proxy = CausalFlowDAG(t_spec, seed=0)
    proxy.fit_classical(c_train.ilocfold_id != j])
    e_oof[fold_id == j] = proxy.pmf(c_train.iloc[fold_id == j], "T")[:, 1]

mae = {}
for propensity in (None, "ps"):
    spec_c = {
        "X": ContinuousNode(SI(transform="affine")),
        "T": OrdinalNode(2, LS("X")),
        "Y": ContinuousNode(LS("X") + VC("X", propensity=propensity, t="T")),
    }
    fc = CausalFlowDAG(spec_c, seed=0)
    fc.fit(
        c_train.assign(ps=e_oof) if propensity else c_train,
        epochs=250,
        learning_rate=1e-2,
        batch_size=512,
        validation_data=c_val,
        seed=0,
        callbacks=EarlyStopping(),
    )
    b = fc.varying_coef(c_test, "Y")
    centered = propensity is not None
    mae[centered] = float(np.abs(b - TAU).mean())
    print(
        f"centered={centered!s:5s}  mean |beta - tau| = {mae[centered]:.3f}"
        f"   mean beta = {b.mean():+.3f}  (true tau {TAU:+.1f})"
    )
ratio = mae[False] / mae[True]
print(f"centering cuts the bias by a factor of {ratio:.1f}")
assert ratio >= 2, f"centering should at least halve the bias: {ratio:.2f}"
```

```
centered=False  mean |beta - tau| = 1.303   mean beta = +0.044  (true tau -1.0)


centered=True   mean |beta - tau| = 0.279   mean beta = -0.722  (true tau -1.0)
centering cuts the bias by a factor of 4.7
```

Without centering the confounding swallows most of the effect; with centering the estimate returns near the truth. The printed factor is the measured bias reduction, and `tests/test_vc_centered.py` requires at least two.

What to take away

you want	use	read it with
one interpretable effect	`LS("T")`	`ls_coefficients()`
an effect that varies with covariates	`VC(*modifiers, t="T")`	`varying_coef(df, node)`

you want	use	read it with
the same under confounding + a misspecified prognostic part	`VC(..., propensity="ps")`	the same read-out
a shortlist of modifiers before you commit	`effect_modifier_scan` on a simple all-`LS` fit	its `flag` column, read as screening

Why not a joint `CS` over treatment and modifiers, and what the penalty and the centering mean: `docs/varying-coefficients.md`. The scan: `docs/scores.md`.

References

Athey, S., Tibshirani, J., & Wager, S. (2019). Generalized random forests. *The Annals of Statistics*, *47*(2). https://doi.org/10.1214/18-AOS1709

Chernozhukov, V., Chetverikov, D., Demirer, M., Duflo, E., Hansen, C., Newey, W., & Robins, J. (2018). Double/debiased machine learning for treatment and structural parameters. *The Econometrics Journal*, *21*(1), C1–C68. https://doi.org/10.1111/ectj.12097

Dandl, S., Haslinger, C., Hothorn, T., Seibold, H., Sverdrup, E., Wager, S., & Zeileis, A. (2024). What makes forest-based heterogeneous treatment effect estimators work? *The Annals of Applied Statistics*, *18*(1). https://doi.org/10.1214/23-AOAS1799

Javaloy, A., Sánchez-Martín, P., & Valera, I. (2023). Causal normalizing flows: From theory to practice. *Advances in Neural Information Processing Systems (NeurIPS)*. https://arxiv.org/abs/2306.05415

Khemakhem, I., Monti, R. P., Leech, R., & Hyvärinen, A. (2021). Causal autoregressive flows. *Proceedings of the 24th International Conference on Artificial Intelligence and Statistics (AISTATS), Proceedings of Machine Learning Research*, *130*, 3520–3528. https://arxiv.org/abs/2011.02268

Nie, X., & Wager, S. (2021). Quasi-oracle estimation of heterogeneous treatment effects. *Biometrika*, *108*(2), 299–319. https://doi.org/10.1093/biomet/asaa076

Pearl, J. (2009). *Causality: Models, reasoning, and inference* (2nd ed.). Cambridge University Press. https://doi.org/10.1017/CBO9780511803161

Robinson, P. M. (1988). Root-N-consistent semiparametric regression. *Econometrica*, *56*(4). https://doi.org/10.2307/1912705

Sánchez-Martín, P., Rateike, M., & Valera, I. (2022). VACA: Designing variational graph autoencoders for causal queries. *Proceedings of the AAAI Conference on Artificial Intelligence*, *36*, 8159–8168. https://doi.org/10.1609/aaai.v36i7.20789

Sick, B., & Dürr, O. (2025). Interpretable neural causal models with TRAM-DAGs. *Proceedings of the 4th Conference on Causal Learning and Reasoning (CLeaR), Proceedings of Machine Learning Research*, *275*, 606–630. https://arxiv.org/abs/2503.16206

Zeileis, A., & Hornik, K. (2007). Generalized M-fluctuation tests for parameter instability. *Statistica Neerlandica*, *61*(4), 488–508. https://doi.org/10.1111/j.1467-9574.2007.00371.x

Zeileis, A., Hothorn, T., & Hornik, K. (2008). Model-based recursive partitioning. *Journal of Computational and Graphical Statistics*, *17*(2), 492–514. https://doi.org/10.1198/106186008X319331