TECHNICAL REPORT | Software version 0.3.2 | 19 October 2026

torchtsetlin

GPU-native, PyTorch-idiomatic Tsetlin machines

Vajira Thambawita Simula Metropolitan Center for Digital Engineering, Oslo, Norway | vajira@simula.no

Abstract. The Tsetlin machine learns propositional rules with teams of Tsetlin automata driven by reward and penalty instead of gradients, which makes its models interpretable by construction and cheap to evaluate. Its software has grown up outside the deep-learning ecosystem, as C and CUDA kernels behind scikit-learn-style wrappers. `torchtsetlin` expresses the whole family as ordinary `torch.nn.Module` objects: vanilla, weighted, coalesced, regression, convolutional over sequences, images and volumes, and dense per-pixel segmentation. The learnable state is a tensor of automaton counters. A model therefore moves across devices with `.to(device)`, serialises with `state_dict()`, and learns from a mini-batch with one `model.update(x, y)` call. The central technical contribution is a batched formulation of Type I/II feedback in which per-example automaton transitions become event counts realised as binomial draws, so there is one commit per batch. On an RTX 3090 this reaches 2.2×10^5 training examples/s and up to 287× the same code on a 16-core CPU, while matching the sequential algorithm at moderate batch sizes. The package also ships an interactive, SQL-queryable catalog of 3,295 results on 128 datasets from 55 Tsetlin-machine papers.

1 Statement of need

A Tsetlin machine [1] represents each class with conjunctive clauses over Boolean literals and their negations. A clause such as `IF x0 AND NOT x1` votes for or against a class, and the vote count is the prediction. Its learned state is a matrix of small integers, one Tsetlin automaton per (clause, literal) pair, updated by counting reward and penalty events. The model is legible and its inference is bit-level cheap. That is why the family has grown convolutional [2], weighted [3], coalesced [4] and regression variants, and is used in clinical decision support and in hardware accelerators.

The reference software does not fit a modern research workflow. Each variant is a separate C or CUDA kernel behind a scikit-learn-style class [9]. Models do not move between devices like the rest of a PyTorch stack, they do not consume a `DataLoader` or serialise like other models, and extending one means writing a kernel. `torchtsetlin` starts from the opposite end: if the automaton counters are a `torch` buffer, every variant is a few tensor operations, and the PyTorch ecosystem [7] comes for free.

2 Design

State as tensors. The automata live in an integer buffer `ta_state` of shape $(C, 2F)$; a literal is included when its counter crosses the action boundary N. The buffers are not parameters (there are no gradients), so `to()`, `state_dict()` and `load_state_dict()` work unchanged. Learning is `update(x, y)`, the analogue of `loss.backward(); optimizer.step()`. It returns the pre-update vote sums, so a trainer logs batch metrics without a second forward pass.

One mechanism, many input structures. Concrete models supply only an output layer and a feedback policy through hooks on a shared base class. A different input structure overrides the encoding, the clause evaluation and the feedback counts. Convolution over 1-D sequences, 2-D images and 3-D volumes is one mixin; a clause fires if it matches any patch, with thermometer-encoded position literals. Dense segmentation is a flat machine with one example per pixel. Neither re-implements learning.

Batched feedback. Classically, every example triggers a stochastic pass over all $C \times 2F$ automata. For a batch of B examples, `torchtsetlin` instead forms the Type I and Type II selection matrices $S_{\mathrm{I}}, S_{\mathrm{II}} \in \{0,1\}^{B \times C}$ and the literal matrix $L \in \{0,1\}^{B \times 2F}$, and counts the events with two matrix products:

$$n^{\text{true}} = S_{\mathrm{I}}^{\top} L, \qquad n^{\text{false}} = S_{\mathrm{I}}^{\top} (1 - L), \qquad n^{\mathrm{II}} = S_{\mathrm{II}}^{\top} (1 - L),$$

plus a column sum n^{Ib} for non-firing clauses. The transition is then drawn in closed form,

$$\Delta = \text{Bin}\left(n^{\text{true}}, p_{\text{mem}}\right) - \text{Bin}\left(n^{\text{false}} + n^{\text{Ib}}, \tfrac{1}{s}\right) + \min\left(n^{\text{II}}, N - \theta\right),$$

and committed once (Figure 1). At $B = 1$ this is the classical algorithm term by term. For $B > 1$ it approximates it the way mini-batch SGD approximates SGD, and `feedback_mode="sequential"` keeps the exact per-example path. A clause-size budget is a hard cap [6]: newly included literals beyond it are reverted at commit.

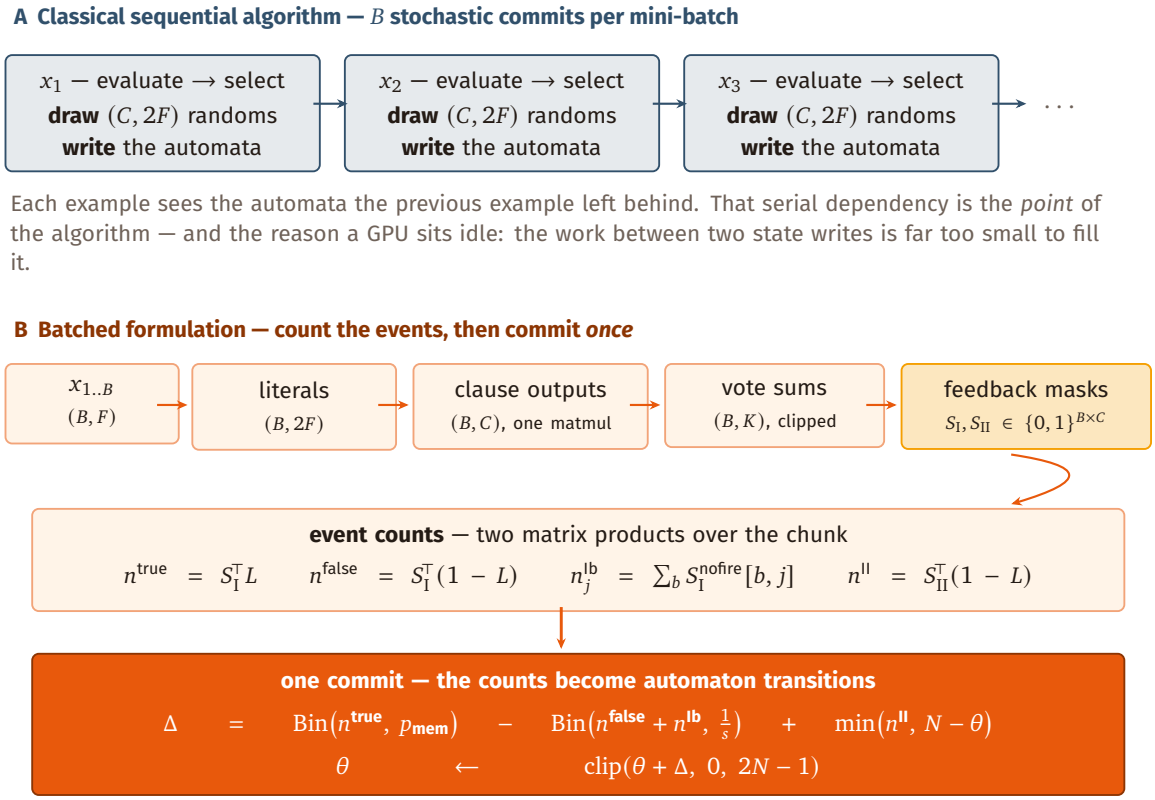


Figure 1. The sequential algorithm (A) makes one stochastic $(C, 2F)$ pass per example. The batched algorithm (B) accumulates the same events as counts and realises them in one binomial draw per batch, so the cost of an update is almost independent of B.

Table 1. What ships in `torchtsetlin` 0.3.2.

Models	`TsetlinMachine` (optional integer clause weights), `CoalescedTsetlinMachine` (shared clause pool, multi-class or multi-label), `RegressionTsetlinMachine`; convolutional 1-D, 2-D and 3-D variants of each; `SegmentationTsetlinMachine` and its coalesced form
Learning	vote margin T, specificity s, memory depth N, boosted true-positive feedback, hard clause-size budget, drop-clause/drop-literal, focused negative sampling, batched or exact sequential feedback
Workflow	Booleanization encoders (thermometer, one-hot, bit-plane, adaptive threshold, colour thermometer, hypervector); a Keras-style `Trainer` with callbacks; classification, regression and segmentation metrics; rule extraction and closed-form global/local importance [5]; plots
Footprint	pure Python on PyTorch ≥ 2.0 and NumPy; no compiler, CUDA toolkit or C extension; Python 3.9–3.12; MIT licence

3 Usage

Learning is a loop over `update`, or a Keras-style `Trainer` with callbacks. On Noisy XOR with 40% label noise, the model recovers the generating rule and prints it:

```
import torchtsetlin as tt

x_train, y_train = tt.data.make_noisy_xor(5000, noise=0.4)    # Boolean features, int labels
x_test,  y_test  = tt.data.make_noisy_xor(2000, noise=0.0, seed=1)
model = tt.TsetlinMachine(n_features=12, n_classes=2, n_clauses=20, T=15, s=3.9).to("cuda")

trainer = tt.Trainer(model, batch_size=10, callbacks=[tt.EarlyStopping(patience=20)])
trainer.fit((x_train, y_train), epochs=100, val_data=(x_test, y_test))
print(trainer.evaluate((x_test, y_test)))      # {'accuracy': 0.99}
print(model.rules()[:2])                       # ['IF x0 AND NOT x1 THEN 1', 'IF x1 AND NOT x0 THEN 1']
```

Images, sequences and volumes use the convolutional classes with a `patch_size`. Continuous columns pass through a thermometer encoder whose feature names flow into the rules, so a rule reads `worst perimeter>=105.9` rather than `x37`. `model.eval()` matters: an empty clause votes while learning and abstains when predicting.

4 Performance

An `update()` call touches every automaton, so its cost depends on the size of the machine far more than on the batch. On an RTX 3090, a 784-feature machine with 500 clauses per class trains at 427 examples/s at batch 1 and 220,028 at batch 4096 (Figure 2). At batch 256 it runs at 87,869 examples/s against 406 on a 16-core CPU, a factor of 216×, and inference reaches 1.06×10^6 examples/s. The advantage grows with the model, up to 287× for the clause budgets the image literature uses: a 113-second MNIST epoch becomes a 1.5-second one.

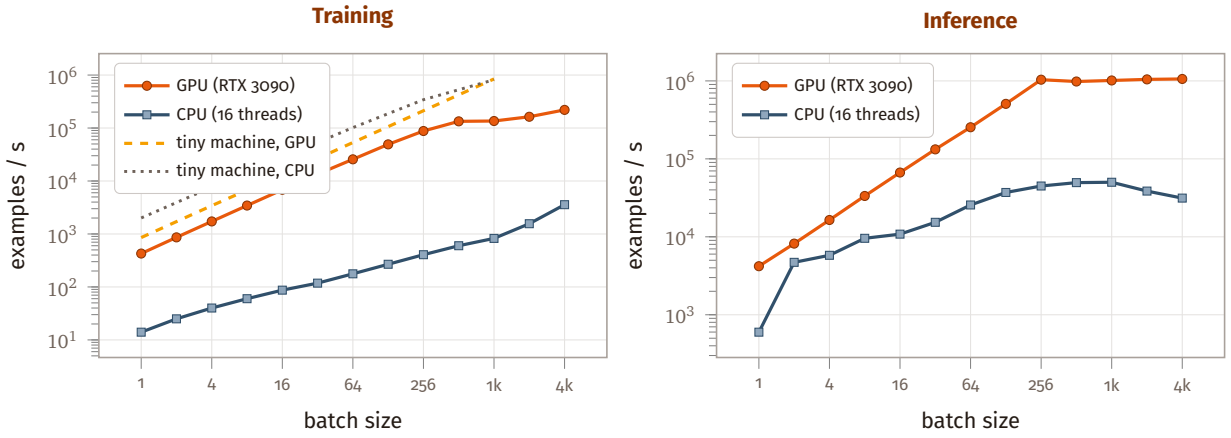


Figure 2. Throughput against batch size, GPU and CPU, for a 784-feature, 500-clause-per-class machine (log–log). The dashed line is a 12-feature, 10-clause machine: less work than a kernel launch, so the GPU only draws level at batch 1024.

Batching costs little accuracy when the batch is moderate. On Noisy XOR, the benchmark of the original paper, batches up to about 50 land in the same accuracy band as the exact sequential algorithm, and batch 200 degrades clearly (Figure 3).

The library therefore recommends batches of 16–64 for flat models and 8–32 for convolutional ones. Every number above comes from the benchmark driver in the repository, which regenerates the documentation's CPU-versus-GPU page from its own records, so it can be re-run on other hardware with one command.

5 An interactive review of the literature

The package ships a catalog of every dataset and reported number in 55 Tsetlin-machine papers (2018–2026): 128 datasets, 252 dataset usages with the splits as reported, and 3,295 results, covering both the proposed models and their baselines. One JSON record per paper is the source of truth. It is compiled into a 1,520 KiB

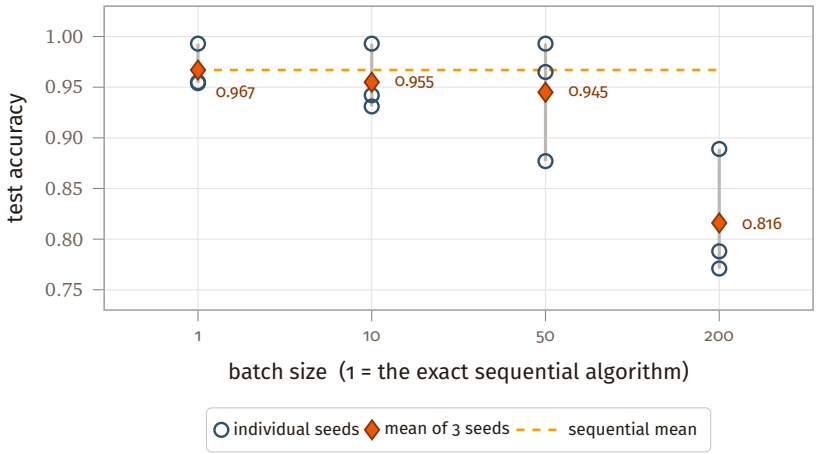


Figure 3. Noisy XOR test accuracy against batch size, 3 seeds; batch 1 is the exact sequential algorithm.

SQLite database, which the documentation opens in the browser with `sql.js`. Readers filter, group and rank results there, see the best value per group, compare the best Tsetlin-machine result with the best baseline per dataset, and run their own SQL. Values are ranked only within the same metric, unit and split, and a metric's direction decides what counts as best. Papers with a known reliability problem carry a caution flag (3 papers). Figure 4 shows two of its views.

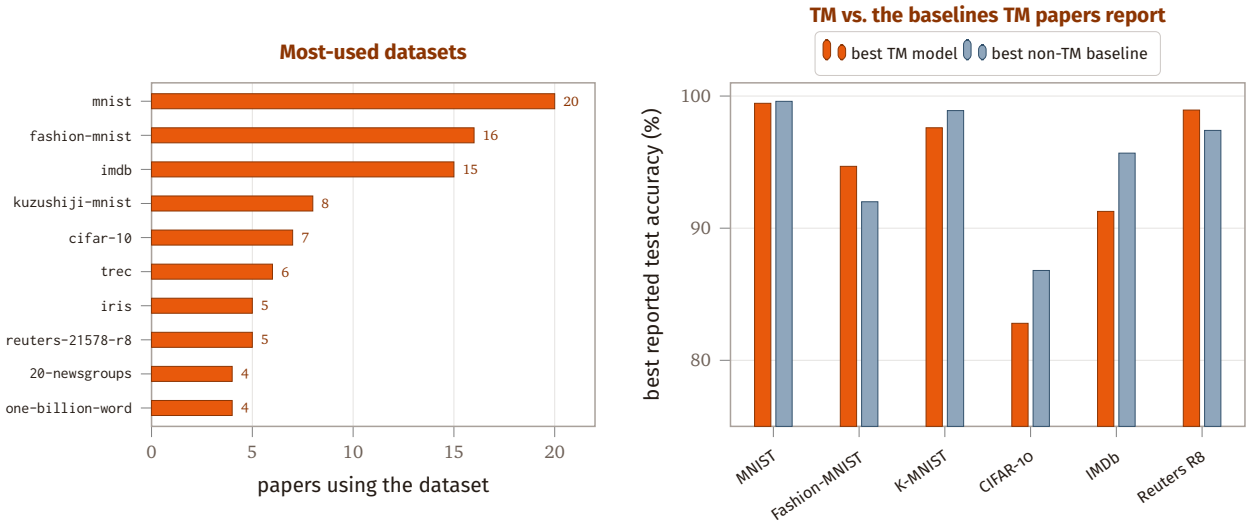


Figure 4. From the literature catalog. *Left:* the datasets used by the most papers. *Right:* the best reported Tsetlin-machine test accuracy next to the best non-TM baseline reported in the same papers, excluding flagged papers and federated per-client accuracies. All values are as reported by their authors, not reproduced.

6 Limitations

Batched feedback is an approximation. The default path trades some fidelity for throughput, and the characterisation is empirical: there is no proof that the batched update converges to the fixed point of the sequential one. The exact algorithm is one argument away.

Dense tensors, not bit-packed kernels. Literals are floats in a matrix product, not bits in a machine word, and clause sparsity is not exploited by indexing. On a small machine on a CPU, the reference C implementations are faster; the advantage of `torchtsetlin` appears with model size and a GPU.

Coverage of the literature. Graph Tsetlin machines, TM composites [8], contracting machines with absorbing automata and several other recent variants are not implemented yet. Training is single-device and single-process.

7 Availability and citation

Install	`pip install torchtsetlin` (version 0.3.2, 19 October 2026)
Source	`https://github.com/vlbthambawita/torchtsetlin` (MIT licence)
Documentation	`https://vajira.info/torchtsetlin/`, including the literature catalog and the generated benchmark pages
Quality gates	`ruff` and the full test suite on Python 3.9, 3.11 and 3.12 in CI; releases via PyPI Trusted Publishing after a clean-environment smoke import

If you use `torchtsetlin`, please cite this report, together with the Tsetlin-machine papers whose algorithms you rely on:

```bibtex
@techreport{thambawita2026torchtsetlin,
  author      = {Thambawita, Vajira},
  title       = {torchtsetlin: GPU-native, PyTorch-idiomatic Tsetlin machines},
  institution = {Simula Metropolitan Center for Digital Engineering},
  type        = {Technical report},
  year        = {2026},
  note        = {Software version 0.3.2},
  url         = {https://github.com/vlbthambawita/torchtsetlin}
}
```

References

[1] O.-C. Granmo. *The Tsetlin Machine — A Game Theoretic Bandit Driven Approach to Optimal Pattern Recognition with Propositional Logic*. arXiv:1804.01508, 2018.

[2] O.-C. Granmo, S. Glimsdal, L. Jiao, M. Goodwin, C. W. Omlin, G. T. Berge. *The Convolutional Tsetlin Machine*. arXiv:1905.09688, 2019.

[3] A. Phoulady, O.-C. Granmo, S. R. Gorji, H. A. Phoulady. *The Weighted Tsetlin Machine: Compressed Representations with Weighted Clauses*. arXiv:1911.12607, 2019.

[4] S. Glimsdal, O.-C. Granmo. *Coalesced Multi-Output Tsetlin Machines with Clause Sharing*. arXiv:2108.07594, 2021.

[5] C. D. Blakely, O.-C. Granmo. *Closed-Form Expressions for Global and Local Interpretation of Tsetlin Machines*. arXiv:2007.13885, 2020.

[6] K. D. Abeyrathna et al. *Building Concise Logical Patterns by Constraining Tsetlin Machine Clause Size*. IJCAI 2023, arXiv:2301.08190.

[7] A. Paszke et al. *PyTorch: An Imperative Style, High-Performance Deep Learning Library*. NeurIPS 2019.

[8] O.-C. Granmo. *TMComposites: Plug-and-Play Collaboration Between Specialized Tsetlin Machines*. arXiv:2309.04801, 2023.

[9] Centre for Artificial Intelligence Research (CAIR). *TMU: the Tsetlin Machine Unified library*. `https://github.com/cair/tmu`.