WHITE PAPER

Version 0.3.2 | 19 October 2026

torchtsetlin

A GPU-native, PyTorch-idiomatic framework for Tsetlin machines

Vajira Thambawita

Simula Metropolitan Center for Digital Engineering, Oslo, Norway

vajira@simula.no

Abstract. The Tsetlin machine is a propositional-logic learner: it builds classifiers out of conjunctive clauses over Boolean literals, updated not by gradient descent but by reward/penalty feedback to teams of Tsetlin automata. Its models are legible by construction and its inference is bit-level cheap, which is why it has become attractive for clinical decision support, edge inference and hardware accelerators. Its software, however, has grown up outside the deep-learning ecosystem: the reference implementations are C and CUDA kernels behind scikit-learn-shaped wrappers, each model family a separate package, with no shared tensor abstraction, no common training loop and no route into PyTorch tooling.

`torchtsetlin` closes that gap. It expresses the whole Tsetlin machine family — vanilla, weighted, coalesced, regression, convolutional over sequences, images and volumes, and dense per-pixel segmentation — as ordinary `torch.nn.Module` objects whose 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 a single `model.update(x, y)` call that plays the role of `loss.backward(); optimizer.step()`. The central technical contribution is a *batched* formulation of Type I/Type II feedback in which per-example stochastic automaton transitions are replaced by event counts accumulated with two matrix products and realised as binomial draws — one commit per batch instead of one per example. On an RTX 3090 this reaches 2.2×10^5 training examples/s and up to 287× the throughput of the same code on a 16-core CPU, while matching the sequential algorithm's accuracy at moderate batch sizes. Around the models the package ships Booleanization encoders, a Keras-style trainer with callbacks, metrics, closed-form global and local interpretability, and visualisation — everything a Tsetlin machine workflow needs, in one PyTorch-shaped package.

Software: `torchtsetlin` 0.3.2 | MIT licence | `pip install torchtsetlin`
Code: https://github.com/vlbthambawita/torchtsetlin | **Docs:** https://vajira.info/torchtsetlin/
Measurements in Section 8 were produced by the benchmark driver shipped with the repository (`benchmarks/run_benchmarks.sh`) and can be reproduced on other hardware with one command.

Contents

1 Executive summary 1

2 Motivation 2
- 2.1 Why a logic learner at all . . . 2
- 2.2 Three obstacles in practice . . . 3

3 Existing solutions and what they leave open 4
- 3.1 The landscape . . . 4
- 3.2 The gap . . . 4

4 The proposed library 6
- 4.1 Design principles . . . 6
- 4.2 What the library provides . . . 6

5 System architecture 8
- 5.1 Four layers . . . 8
- 5.2 The learning state . . . 9
- 5.3 How an update flows . . . 9
- 5.4 Extension points . . . 10

6 Technical design 11
- 6.1 Representation . . . 11
- 6.2 Inference is two matrix products . . . 11
- 6.3 The feedback policy . . . 12
- 6.4 Batched feedback: the core reformulation . . . 12
- 6.5 The clause budget is a hard cap . . . 13
- 6.6 Memory as an explicit budget . . . 15
- 6.7 Convolutional machines . . . 15
- 6.8 Volumes: 3D convolution . . . 16
- 6.9 Dense prediction: segmentation . . . 17
- 6.10 Booleanization . . . 18
- 6.11 Interpretation . . . 19
- 6.12 Correctness . . . 20

7 Usage 20
- 7.1 Thirty seconds . . . 20
- 7.2 Tabular data, rules in the original units . . . 21
- 7.3 Convolutional MNIST on a GPU . . . 21
- 7.4 Volumes and dense prediction . . . 22
- 7.5 Explaining a single prediction . . . 23
- 7.6 Building a new variant . . . 23

8 Benchmarks 23
- 8.1 Method . . . 24
- 8.2 Throughput against batch size . . . 24
- 8.3 Throughput against model size . . . 25
- 8.4 The model zoo . . . 25
- 8.5 Where an update actually goes . . . 26
- 8.6 CPU threads . . . 26
- 8.7 Where the data lives . . . 27
- 8.8 Segmentation and volumes . . . 27
- 8.9 End to end . . . 28
- 8.10 What batching costs in accuracy . . . 29

9 An interactive review of the literature 29
- 9.1 How it was built . . . 30
- 9.2 Using it . . . 30

10 Applications 31
- 10.1 What ships with the library . . . 31
- 10.2 Where a Tsetlin machine earns its place . . . 32

11 Limitations and future development 33
- 11.1 Known limitations . . . 33
- 11.2 Where it goes next . . . 35

12 Availability 36
- 12.1 Getting started . . . 36
- 12.2 Reproducing the measurements . . . 36

12.3 Quality gates 36
12.4 Documentation 37
12.5 Citation 37
12.6 Contributing 37
References **37**

1 Executive summary

How to read this document

Sections 1–4 state the problem and what the library offers; a reader who only wants to know *whether* to use `torchtsetlin` can stop after Section 4. Sections 5–7 are the engineering core: architecture, the batched learning algorithm, and the API as it is actually used. Section 8 reports measured performance, Section 9 the interactive catalog of the Tsetlin-machine literature, and Sections 10–12 cover applications, known limits and where to get the code.

`torchtsetlin` is an open-source Python package that implements the Tsetlin machine family as first-class PyTorch modules. A Tsetlin machine learns *propositional rules* — clauses such as `IF x0 AND NOT x1 THEN class 1` — using teams of Tsetlin automata driven by reward and penalty signals rather than gradients [1]. The resulting model is not merely explainable after the fact; the model *is* the explanation, and evaluating it costs a conjunction over included literals plus a vote count.

The problem `torchtsetlin` addresses is not the algorithm, which is well established, but its software. Tsetlin machine implementations have historically been standalone C or CUDA kernels wrapped in scikit-learn-shaped Python classes. That design serves a benchmark script well and serves a research workflow badly: models do not move between devices the way the rest of a practitioner's stack does, they do not serialise like other models, they do not consume a `DataLoader`, and extending them means writing a kernel rather than a few lines of tensor code.

`torchtsetlin` takes the opposite starting point. The learnable state — one integer counter per (clause, literal) pair — is a `torch` buffer inside an `nn.Module`. Everything else follows from that decision:

- **One line to learn.** `model.update(x, y)` is the analogue of `loss.backward(); optimizer.step()`. It returns the pre-update vote sums, so a training loop gets batch metrics without a second forward pass.
- **One line to move.** `model.to("cuda")` moves the automata; there is no separate device build, no kernel recompilation and no host round-trip through NumPy.
- **Everything else is PyTorch.** `state_dict()`, `load_state_dict()`, `DataLoader`, `Dataset`, hooks, `torch.compile` and profiling all work because there is nothing non-standard to work around.
- **Batched feedback.** The learning update is reformulated so that a mini-batch is consumed in a handful of tensor operations rather than one stochastic pass per example. This is what makes GPU execution worthwhile (Section 6.4).
- **One mechanism, many input structures.** Sequences, images and volumes are the same convolutional machine over a different patch set, and dense segmentation is a flat machine with one example per pixel. Neither re-implements learning (Sections 6.7–6.9).
- **A complete workflow.** Booleanization encoders, a Keras-style `Trainer` with callbacks, metrics, rule extraction, closed-form feature importance and plotting ship in the same package.

At a glance

Table 1 is the whole package in one view.

Table 1. `torchtsetlin` at a glance.

Models	`TsetlinMachine` (multi-class, optional integer clause weights), `CoalescedTsetlinMachine` (shared clause pool; multi-class or multi-label), `RegressionTsetlinMachine`, and convolutional variants `ConvTsetlinMachine`, `ConvCoalescedTsetlinMachine`, `ConvRegressionTsetlinMachine`, `Conv1dTsetlinMachine`; 3D variants `Conv3dTsetlinMachine`, `Conv3dCoalescedTsetlinMachine`, `Conv3dRegressionTsetlinMachine`; dense `SegmentationTsetlinMachine`, `CoalescedSegmentationTsetlinMachine`
Learning	vote margin T, specificity s, memory depth N, boosted true-positive feedback, clause-size budget (a hard cap), per-class feedback rates, drop-clause/drop-literal, focused negative sampling, batched or exact sequential feedback
Data	thermometer, one-hot, bit-plane, adaptive-threshold, colour-thermometer, hypervector and multi-scale pyramid encoders; synthetic images, volumes and scenes; torchvision and segmentation loaders
Training	`Trainer` over tensors/`Dataset`/`DataLoader`; early stopping, checkpoints, CSV logging, hyper-parameter schedules, `History`
Evaluation	classification, regression and multi-label metrics; pixel accuracy, IoU, Dice and boundary F1 for segmentation
Interpretation	rule extraction, clause activity and precision, closed-form global and local feature importance, per-example and per-pixel explanations, the image or volume region a convolutional clause is allowed to fire in
Literature	a searchable catalog of 128 datasets and 3,295 reported results from 55 TM papers, queried as SQL in the documentation (Section 9)
Peak measured	2.20×10^5 training examples/s and 1.06×10^6 inference examples/s (RTX 3090, 784 features, 500 clauses/class); up to 287× a 16-core CPU on the same code
Footprint	pure Python + PyTorch $\geq$ 2.0 and NumPy; no compiler, no CUDA toolkit, no C extension; Python 3.9–3.12; MIT licence

In short. A Tsetlin machine is a matrix of small integers updated by counting events. Once that is said in tensors instead of in C, the algorithm inherits the entire PyTorch ecosystem for free — and a GPU turns a 113-second MNIST epoch into a 1.5-second one.

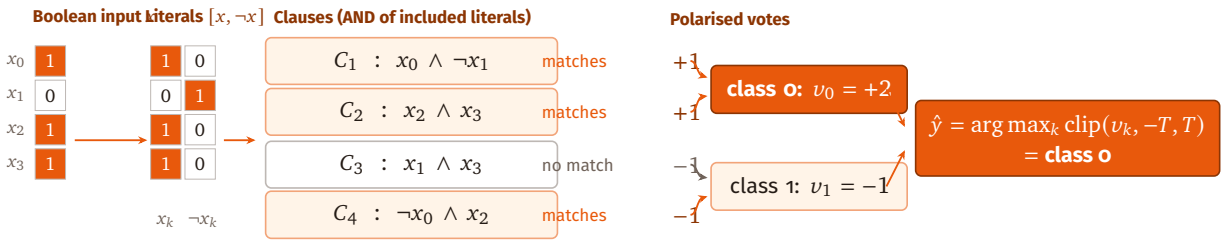
2 Motivation

2.1 Why a logic learner at all

Deep networks answer; they rarely account for their answers. In clinical decision support, regulated industry and on-device inference, the account matters as much as the answer: a cardiologist wants to know which measurement moved the prediction, an auditor wants a decision rule they can read, and a microcontroller wants an inference path that is a handful of bit operations rather than a matrix cascade.

The Tsetlin machine [1] occupies exactly that niche. It learns conjunctive clauses over Boolean literals; clauses vote for or against each class; the class with the largest vote sum wins (Figure 1). Three properties follow:

- **Legibility by construction.** A clause is a readable conjunction. There is no surrogate model, no saliency heuristic, and no gap between what is explained and what is executed. Global and local importance have closed-form expressions [4].



Every clause is a conjunction the reader can check by hand; the prediction is a signed vote count over the clauses that matched. The model is its own explanation, and inference is an AND plus an addition.

Figure 1. A Tsetlin machine classifier at inference time. Features are expanded into literals $[x, \neg x]$; each clause is a conjunction of the literals its automata have decided to *include*; matching clauses cast signed votes; the argmax over clipped vote sums is the prediction.

- **Bit-level economy.** Inputs are Booleans, the learned state is small integers, and inference is an AND over included literals plus a vote count — which is why a substantial hardware literature has grown around it, from FPGA and ASIC inference engines to in-memory computing [15,16,17].
- **Competitive accuracy** on the tasks where those properties are worth paying for: tabular and clinical data, keyword spotting, and small-image classification.

2.2 Three obstacles in practice

The motivation for `torchtsetlin` came from trying to use Tsetlin machines in a medical-imaging research group, where the surrounding stack is PyTorch and the data are large. Three obstacles showed up immediately, and none of them is about the algorithm.

1. The learning rule looks hostile to a GPU. The classical algorithm is defined per example: evaluate the clauses, draw feedback, *update the automata*, then move to the next example. The update is stochastic and touches every (clause, literal) counter, so a naive implementation performs one small, serial, random-number-heavy pass per training example. On a device with thousands of cores and a kernel-launch cost, that is the worst possible shape of work. A GPU implementation therefore requires reformulating the update, not merely porting it.

2. The tooling lives outside the ecosystem. Existing implementations are compiled kernels behind thin wrappers (Section 3). They are fast and faithful, but they sit beside a PyTorch pipeline rather than inside it. Moving a model to a GPU, checkpointing it mid-training, feeding it from a `DataLoader` that already augments on-device, or profiling it next to the CNN it is being compared against — each requires bespoke glue, and the data usually makes a round trip through host memory and NumPy on the way.

3. Everything around the model is re-invented per project. A Tsetlin machine consumes Booleans, so every project starts by writing a Booleanizer: thermometer thresholds for tabular columns, adaptive thresholding for images, bit planes for integers. Then a training loop, then early stopping, then a rule printer, then a clause visualiser. This is the unglamorous 80% of a Tsetlin machine project, it is rewritten constantly, and small differences in it (which thresholds, which threshold strategy) move accuracy more than the model hyper-parameters do.

The design question

Can the Tsetlin machine be expressed so that it is *ordinary* inside PyTorch — same device semantics, same checkpoints, same data plumbing — without giving up fidelity to the published algorithm? `torchtsetlin` is the answer to that question: yes, if the learning update is reformulated from per-example state mutation into per-batch *event counting* (Section 6.4), and if the exact per-example path is kept available for when fidelity matters more than speed.

3 Existing solutions and what they leave open

3.1 The landscape

Tsetlin machine software has been developed almost entirely by the research groups that developed the algorithms, and it reflects that origin: each implementation is organised around a kernel that is fast and faithful to one paper, with a thin Python surface on top.

Reference CPU implementations. The canonical Python packages (`pyTsetlinMachine` and successors) implement the machine in C and expose it through a wrapper offering `fit`/`predict`/`transform`, one class per published variant (multi-class, convolutional, regression, weighted). They are the semantic ground truth for the algorithm and remain the reference any other implementation should be checked against.

CUDA implementations. `PyTsetlinMachineCUDA` and its relatives move the same algorithm into hand-written CUDA kernels, typically compiled at import time through PyCUDA. They demonstrated that the Tsetlin machine *can* be run at scale on a GPU — but as a separate package with a separate class hierarchy, so a project that needs both CPU and GPU maintains two code paths.

Unified architectures. More recent work (the Tsetlin Machine Unified project, `tmu`) factors the machine into reusable components — clause banks, weight banks — so that new variants can be assembled rather than rewritten, with CPU and optional CUDA backends behind one interface. This is the closest in spirit to `torchtsetlin`: a shared substrate instead of a family of one-off programs. The substrate, however, is a custom C/CUDA layer rather than a tensor library, so the interoperability question below is unchanged.

Specialised and third-party work. Beyond these sit dedicated packages for particular variants (graph Tsetlin machines [14], sparse and coalesced CUDA machines), community ports in Rust, Julia, Go and JavaScript, and a growing hardware literature — FPGA and ASIC inference engines, asynchronous logic designs and in-memory computing architectures [15,16,17] — which target deployment rather than research iteration.

3.2 The gap

None of this is a criticism of the implementations on their own terms; they are fast, they are correct, and several of them predate the workflows described here. The gap is structural, and it has four parts.

1. **No shared tensor abstraction.** The learned state lives in a private C or CUDA buffer. It cannot be inspected, sliced, masked, moved or saved with the tools a PyTorch user already has; every

Table 2. Structural comparison of implementation approaches. ● available, ◐ partial or variant-specific, ○ not applicable to the design. The table compares *integration properties*, not accuracy or peak kernel speed; no head-to-head benchmark against other packages was run for this paper (Section 11).

Property	Reference C packages	CUDA packages	Unified C/CUDA	torchtsetlin
Learned state is a framework tensor	○	○	○	●
`.to(device)` moves the model	○	○	◐	●
Same code path on CPU and GPU	○	○	◐	●
`state_dict()` checkpointing	○	○	○	●
Consumes `Dataset`/`DataLoader`	○	○	○	●
Accepts device-resident input	○	◐	◐	●
New variant without compiled code	○	○	◐	●
Booleanization encoders included	◐	◐	●	●
Training loop with callbacks	○	○	○	●
Interpretability utilities	◐	◐	●	●
Plotting utilities	○	○	◐	●
Requires a C/CUDA toolchain	●	●	●	○
Hand-tuned bit-packed kernels	●	●	●	○

such operation needs an accessor written for it.

2. **Device transfer is a packaging decision, not a runtime one.** Which device a model runs on is chosen by which package was imported, so CPU debugging and GPU training are different code paths rather than `.to("cpu")` and `.to("cuda")`.

3. **The data path leaves the framework.** Booleanized batches are NumPy arrays on the host. A pipeline that already has its data on the GPU — the normal case when the Tsetlin machine is being compared against, or combined with, a neural network — pays a round trip in both directions.

4. **Extension means writing a kernel.** Trying a modified feedback rule, an alternative negative-sampling scheme or a new output layer requires C or CUDA, which puts the experiment out of reach of most of the people who want to run it.

Table 2 summarises this. The last row is deliberate and cuts the other way: a hand-written kernel can pack literals into machine words and exploit clause sparsity in ways a generic tensor library cannot, so for a small machine on a CPU the reference implementations remain the faster choice. `torchtsetlin` trades that per-operation efficiency for parallel width and for integration — a trade that pays exactly when the machine is large and a GPU is present, and that is quantified in Section 8.

In short. The algorithms are well served; the *workflow* is not. What has been missing is a Tsetlin machine that behaves like every other model in a modern Python stack — which means building it on a tensor library rather than beside one.

4 The proposed library

4.1 Design principles

torchtsetlin is built on five commitments, in priority order. Where they conflict, the earlier one wins.

1. **Be a `torch.nn.Module`, not a wrapper around one.** The automata state is a registered buffer. Every consequence of that — device moves, checkpoints, `repr`, hooks — is inherited rather than implemented.
2. **Be faithful to the published algorithm, and be explicit where you are not.** The exact per-example algorithm is always available (`feedback_mode="sequential"`). The default batched path is an approximation in the same sense as mini-batch SGD, its fidelity is characterised empirically (Section 8.10), and the trade is documented rather than hidden.
3. **Keep the learning mathematics stateless and separate.** All of it lives in one module, `functional.py`, as pure tensor functions. Models are thin, stateful compositions of those functions, which is what makes a new variant a subclass rather than a fork.
4. **Ship the whole workflow.** Booleanization, training, evaluation, interpretation and visualisation are part of the package, because in practice they are part of the job.
5. **Depend on as little as possible.** PyTorch and NumPy are the only hard requirements. Matplotlib, torchvision and scikit-learn are optional and import-guarded. There is no build step.

4.2 What the library provides

Models. Twelve concrete machines, all sharing one base class and one learning loop (Table 3). The flat, convolutional, volumetric and dense families are not parallel hierarchies. Each input structure is a mix-in that overrides input encoding, clause evaluation and feedback counting, and is combined with a flat model to produce its counterpart: convolution over images (Section 6.7), its 3D extension over volumes (Section 6.8), and dense per-pixel prediction (Section 6.9).

Learning options. The switches that the literature has shown to matter are constructor arguments rather than forks: vote margin T and specificity s; memory depth N; boosted true-positive feedback; integer clause weights; the clause-size constraint of [6], enforced as a hard cap (Section 6.5); drop-clause and drop-literal regularisation [8]; focused negative sampling; per-class feedback rates for imbalanced segmentation; and the choice between batched and exact sequential feedback.

Booleanization. Eight encoders — a plain binarizer, thermometer (quantile, uniform or unique-value thresholds), one-hot, bit plane, adaptive Gaussian/mean threshold, colour thermometer, sparse hypervector [13] and an OR-pooled multi-scale pyramid for dense prediction — all implemented as `nn.Module`s with the scikit-learn `fit`/`transform` protocol, composable with `Compose`, movable to a device and saved in a `state_dict`. Encoders also carry their inverse knowledge: `ThermometerEncoder.decode_range` turns included bits back into an interval in the original units, so a rule reads `IF age>=52.4 AND NOT age>=67.1` rather than `IF x37 AND NOT x41`.

Training. A `Trainer` with `fit`/`evaluate`/`predict` over tensors, `Dataset`s or `DataLoader`s; a `History` object; and callbacks for early stopping, checkpointing, CSV logging, hyper-parameter scheduling and automata-state logging. Nothing in the models requires the trainer — a hand-written loop over `model.update` is a first-class way to use the library.

Table 3. The model zoo. F is the number of Boolean features, C the total clause count, K the number of outputs. Every model stores its automata in one $(C, 2F)$ integer buffer.

Class	Task	Output layer and distinguishing feature
`TsetlinMachine`	multi-class	$C = K \cdot n$ clauses, half positive and half negative polarity per class; optional integer clause weights (weighted TM [3])
`CoalescedTsetlinMachine`	multi-class, multi-label	one shared clause pool with a signed integer weight per (clause, output); clauses may change sides during learning [5]
`RegressionTsetlinMachine`	regression	all clauses positive; vote sum in $[0, T]$ mapped linearly onto the target range [7]
`ConvTsetlinMachine`	image classification	clauses slide over $k_h \times k_w$ patches and fire if *any* patch matches; thermometer-encoded patch coordinates [2]
`ConvCoalescedTsetlinMachine`	image, multi-label	convolutional clauses over a shared pool
`ConvRegressionTsetlinMachine`	image regression	reads a continuous quantity off an image [7]
`Conv1dTsetlinMachine`	sequences, signals	$1 \times k$ windows over (B, Z, L) Boolean channels
`Conv3dTsetlinMachine`	volume classification	$k_d \times k_h \times k_w$ windows over (B, Z, D, H, W) volumes (MRI/CT stacks, or frames with time as depth); depth/row/column position literals
`Conv3dCoalescedTsetlinMachine`	volume, multi-label	3D convolutional clauses over a shared pool
`Conv3dRegressionTsetlinMachine`	volume regression	reads a continuous quantity off a volume
`SegmentationTsetlinMachine`	semantic segmentation	one example per pixel, classified from its $k_h \times k_w$ neighbourhood; returns a (B, H, W) label map and a (B, K, H, W) vote map
`CoalescedSegmentationTsetlinMachine`	segmentation, multi-label	dense head over a shared clause pool; one clause can serve every class it helps

Evaluation and interpretation. Accuracy, confusion matrix, per-class precision/recall/F1, regression and multi-label metrics, expected calibration error and trustworthiness curves; for segmentation, pixel accuracy, IoU, mean IoU, Dice and boundary F1, accumulated batch by batch so a test set never has to sit in memory as a vote map. Rule extraction, clause activity and precision, closed-form global and local feature importance [4], per-example explanations listing the clauses that fired and how they voted, and per-pixel explanations decoded into statements about named pixels of the input.

Visualisation. Training curves, confusion matrices, automata-state heat maps, clause memory plots, convolutional clause patches rendered as pixel stencils (3D clauses slice by slice), vote distributions, trustworthiness plots, and segmentation panels and per-class vote maps.

> **What `torchtsetlin` is not**
>
> It is not a reimplementation of every Tsetlin machine variant in the literature — graph machines, autoencoders, absorbing automata and TM composites are not implemented (Section 11). It is not a bit-packed kernel, and on a small machine on a CPU it will not beat one. It is the substrate on which those things can be built in tensor code instead of C.

5 System architecture

5.1 Four layers

torchtsetlin is organised as four layers with a strict one-way dependency (Figure 2). Each layer is useful on its own: a researcher testing a new feedback rule works in layer 1, someone adding a model variant works in layer 3, and an applied user only ever sees layer 4.

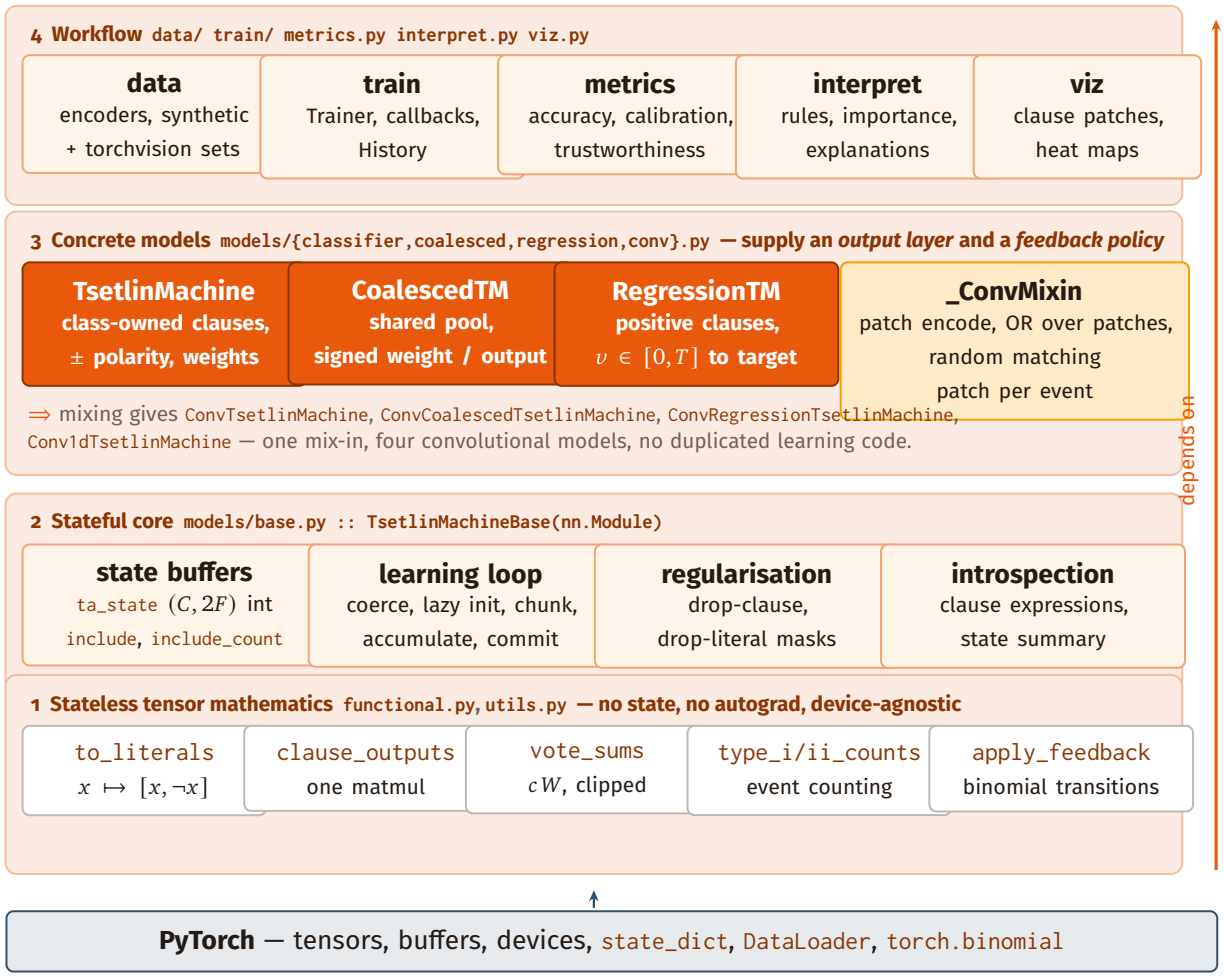


Figure 2. The four layers. Dependencies point downwards only: the learning mathematics does not know what a model is, and a model does not know what a trainer is.

Layer 1 — `functional.py`. Stateless tensor functions: literal expansion, clause evaluation, vote sums, feedback probabilities, negative-class sampling, Type I and Type II event counting, and the automaton transition itself. No state, no autograd, no device assumptions. All learning mathematics lives here, which means it can be unit-tested against a brute-force reference implementation and reused by code that does not use the model classes at all.

Layer 2 — `TsetlinMachineBase`. An `nn.Module` that owns the automata and runs the generic learning loop: input coercion, lazy shape initialisation, chunking, dropout masks, feedback accumulation and the commit. It knows nothing about classes, labels or images.

Layer 3 — concrete models. Each variant supplies only an *output layer* (how clause outputs become votes) and a *feedback policy* (which clauses receive which feedback for a labelled example).

Input structure is orthogonal to both and enters as a mix-in, so `class ConvTsetlinMachine(_ConvMixin, TsetlinMachine)` is the entire definition of the convolutional classifier's learning behaviour. Three mix-ins exist: `_ConvMixin` (images and sequences), `_Conv3dMixin`, which subclasses it and replaces only the geometry (volumes), and `_DenseMixin`, which drops the OR over patches and folds the patch axis into the batch axis (segmentation). Each combines with the flat classifier, the coalesced machine and, where it makes sense, the regression machine.

Layer 4 — workflow. Data preparation, training, metrics, interpretation and plotting. These depend on the models through their public API only, so a user's own model subclass gets the whole workflow layer for free.

5.2 The learning state

The learnable state is a **buffer, not a parameter**, because a Tsetlin machine has no gradients:

Buffer	Shape / dtype	Role
`ta_state`	$(C, 2F)$ `int32`	the automata; literal included iff state $\geq N$
`include`	$(C, 2F)$ float	cached 0/1 mask, derived, non-persistent
`include_count`	$(C,)$ long	literals per clause, derived, non-persistent
`weights`	$(C,)$ or (C, K) `int32`	clause weights (weighted/coalesced models)

Only `ta_state` and `weights` are persistent; the two caches are recomputed whenever the state changes, including after `load_state_dict`. Storing the derived mask costs $2CF$ floats and removes a comparison from the inner loop of every forward pass, which is the single hottest path in the library.

Two consequences are worth calling out. First, `model.parameters()` is deliberately empty and warns when it returns nothing, because silently returning an empty iterator to someone who wrote `optim.SGD(model.parameters())` would be worse than an error. Second, `n_features=None` enables lazy initialisation in the style of `torch.nn.LazyLinear`: the state is allocated on the first batch, and `_load_from_state_dict` reconstructs the shape from the checkpoint before loading so that a lazily built model can be restored without ever having seen data.

5.3 How an update flows

`update(x, y)` is the analogue of `loss.backward(); optimizer.step()`, and it returns the *pre-update* vote sums so that a training loop can log batch accuracy without a second forward pass. Figure 3 shows the flow.

The separation between *accumulate* and *commit* is the structural core of the library. Accumulation is pure counting and can be done in any number of chunks, in any order, without changing the result. The commit is the only place the automata are written, the only place randomness enters the state, and the only place the derived caches are refreshed. That split is what makes the batched algorithm expressible, what bounds memory usage, and what makes the convolutional mix-in possible — convolution changes what is counted and how, but not when or how it is committed.

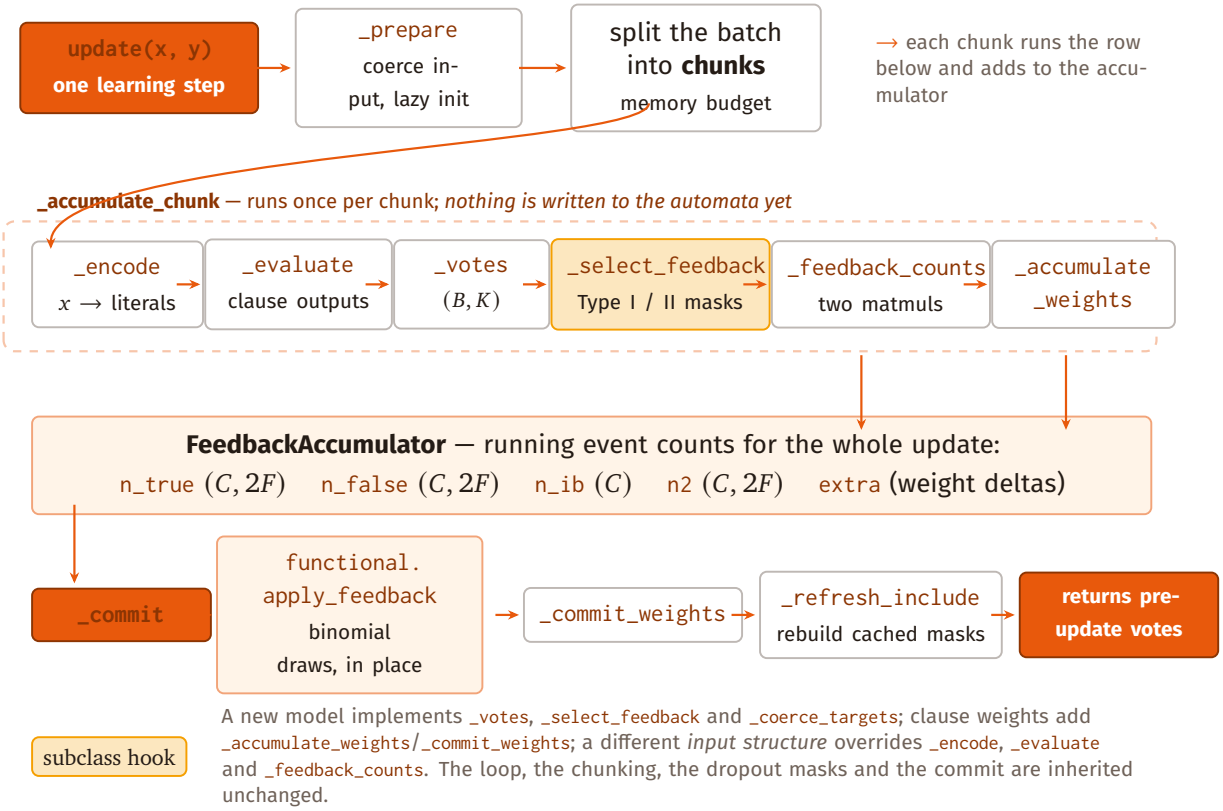


Figure 3. The `update()` pipeline. The batch is split into chunks that respect a memory budget; each chunk contributes *event counts* to a shared accumulator; the automata are written once, at the commit. Amber marks the hook a new model must implement.

5.4 Extension points

A new model is a subclass, not a fork. The required surface is three methods:

Method	Contract
`_votes(clause_out)`	(B, C) Boolean clause outputs → (B, K) vote sums
`_select_feedback(votes, y, clause_out)`	→ two (B, C) Boolean masks: who gets Type I, who gets Type II
`_coerce_targets(y, n, device)`	validate and convert the targets
optional	
`_accumulate_weights` / `_commit_weights`	clause weights
`_encode` / `_evaluate` / `_feedback_counts`	a different input structure (the convolutional, 3D and dense mix-ins)
`_chunk_elements_per_example`	the memory model for that structure
`rules()`	how the model prints itself

In short. Learning mathematics is stateless and lives in one file; state and the loop live in one base class; variants supply an output layer and a feedback policy. The entire convolutional family is one mix-in over that structure — roughly 360 lines, none of which re-implement learning — and its 3D extension adds about 290 more, all geometry.

6 Technical design

This section states the algorithm as `torchtsetlin` implements it. The notation throughout: B examples per batch, F Boolean features, $2F$ literals, C clauses, K outputs, N automaton states per action, s specificity, T vote margin.

6.1 Representation

An example is a Boolean vector $x \in \{0,1\}^F$. Its *literal vector* is $L(x) = [x, \neg x] \in \{0,1\}^{2F}$. Every (clause, literal) pair owns a Tsetlin automaton whose state $\theta_{jk} \in \{0, \dots, 2N-1\}$ decides membership:

$$\text{include}_{jk} = \mathbb{1}\left[\theta_{jk} \geq N\right], \qquad c_j(x) = \bigwedge_{k\,:\,\text{include}_{jk}} L_k(x).$$

The whole learned state is therefore one integer matrix $\theta \in \mathbb{Z}^{C \times 2F}$ — in `torchtsetlin`, the `int32` buffer `ta_state`. Two derived, non-persistent buffers cache the include mask as floats and the per-clause literal count; anything that mutates θ refreshes them, including `load_state_dict`.

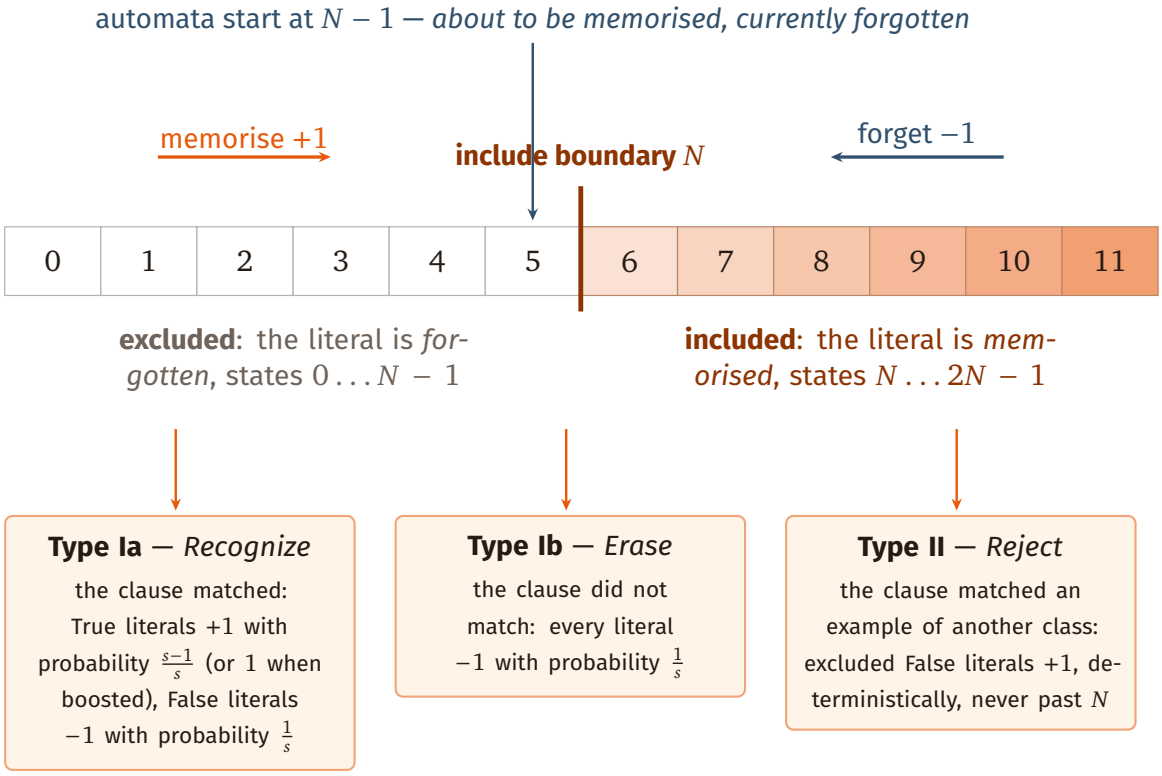


Figure 4. One Tsetlin automaton. The state line is split at N: below it the literal is forgotten, at or above it the literal is included in the clause. Deep states are hard to dislodge, states near the boundary flip easily — this is the memory that makes the machine robust to noisy examples.

6.2 Inference is two matrix products

A conjunction is awkward to vectorise directly, but its negation is a sum. A clause fails on an example exactly when it includes at least one literal that is False, so

$$\text{violations} = (1 - L)\,\text{include}^\top \in \mathbb{R}^{B \times C}, \qquad c_j(x_b) = \mathbb{1}\left[\text{violations}_{bj} = 0\right].$$

One dense matrix product evaluates every clause on every example. The vote sums are a second product, $v = c\,W$, clipped to $[-T, T]$, where $W \in \mathbb{Z}^{C\times K}$ holds the signed clause weights. Prediction is $\hat{y} = \arg\max_k v_k$.

> **Empty clauses are mode-dependent — and this is not a detail**
> A clause with no included literals has zero violations, so the formula above makes it match *everything*. That is correct while learning: a fresh clause must fire in order to receive the Type Ia feedback that gives it its first literal. It is wrong at prediction time, where an empty clause should vote for nothing. `torchtsetlin` keys this off `self.training`, exactly like `nn.Dropout`, so `model.eval()` is required before measuring accuracy — forgetting it silently changes predictions rather than raising. The behaviour is pinned by tests.

6.3 The feedback policy

For a labelled example the machine decides, per clause, whether to give Type I feedback (make the clause recognise this example), Type II feedback (make it stop matching), or nothing. The decision is probabilistic in the vote margin: the target class receives feedback with probability $(T - v_y)/2T$ and a sampled negative class with probability $(T + v_q)/2T$. Once a class already wins by T votes its clauses stop being updated, which frees them to specialise elsewhere — the resource-allocation mechanism at the heart of the algorithm [1].

Each model differs only in how votes are formed and who receives which feedback (Table 4). Everything below that line is shared.

Table 4. Output layer and feedback policy per model. This table *is* the difference between the model classes; the learning machinery underneath is identical.

Model	Vote sums	Who receives what
`TsetlinMachine`	$v_k = \sum_{j\in \mathrm{pos}(k)} w_j c_j - \sum_{j\in \mathrm{neg}(k)} w_j c_j$	target class: Type I to positive, Type II to negative clauses; one random negative class: the reverse
`CoalescedTsetlinMachine`	$v = c\,W$, $W \in \mathbb{Z}^{C\times K}$	per output, Type I when the weight sign agrees with the target and Type II when it does not; weights move by ± 1, so a clause can change sides
`RegressionTsetlinMachine`	$v = \sum_j w_j c_j$ clipped to $[0, T]$, mapped to $[y_{\min}, y_{\max}]$	Type I with probability $(v^\star - v)/T$ when the prediction is too low, Type II with $(v - v^\star)/T$ when too high

Two refinements are available on the classifiers. *Integer clause weights* (the weighted TM [3]) let one clause count for many, which is what makes large convolutional machines tractable: Type Ia increments a clause's weight, Type II decrements it. *Focused negative sampling* draws the negative class in proportion to how strongly it currently votes rather than uniformly, so the machine corrects the classes it actually confuses first — worth having when K is large.

6.4 Batched feedback: the core reformulation

The classical algorithm applies feedback one example at a time, re-evaluating the clauses in between. Each step draws random numbers for the whole $(C, 2F)$ state and writes it back. For a machine with 5 000 clauses over 784 features that is a 7.8-million-element stochastic pass *per training example* — on a GPU, a sequence of small kernels with nothing to parallelise across.

`torchtsetlin` reformulates the update around a simple observation: *every stochastic decision in Type I feedback is an independent Bernoulli trial with a state-independent probability*. Memorisation happens with probability $p_{\mathrm{mem}} = (s-1)/s$ (or 1 with boosted true positives), forgetting with

probability $1/s$. Whether a particular (clause, literal) automaton is offered such a trial depends only on which feedback events the batch produced — not on the automaton's own state. So the trials can be *counted* first and *sampled* once.

Let $S_{\mathrm{I}}, S_{\mathrm{II}} \in \{0,1\}^{B\times C}$ mark which (example, clause) pairs receive Type I and Type II feedback among the clauses that *matched*, and $S_{\mathrm{I}}^{\mathrm{nofire}}$ those that received Type I without matching. Then, with $L \in \{0,1\}^{B\times 2F}$ the batch literal matrix:

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

$$n_j^{\mathrm{Ib}} = \textstyle\sum_b S_{\mathrm{I}}^{\mathrm{nofire}}[b,j], \qquad n^{\mathrm{II}} = S_{\mathrm{II}}^{\top}(1-L). \tag{2}$$

Four count tensors, obtained from two matrix products and a column sum. The transition then follows in closed form:

$$\Delta = \underbrace{\mathrm{Bin}\left(n^{\mathrm{true}},\, p_{\mathrm{mem}}\right)}_{\text{Type Ia memorise}} - \underbrace{\mathrm{Bin}\left(n^{\mathrm{false}} + n^{\mathrm{Ib}},\, \tfrac{1}{s}\right)}_{\text{Type Ia forget + Type Ib}} + \underbrace{\min\left(n^{\mathrm{II}},\, N-\theta\right)}_{\text{Type II, bounded at the boundary}}, \tag{3}$$

and $\theta \leftarrow \mathrm{clip}(\theta + \Delta,\, 0,\, 2N-1)$. The two forgetting sources merge because both are Bernoulli$(1/s)$ trials and a sum of independent Bernoulli trials with equal probability is binomial. Type II is deterministic, and the min reproduces a property of the sequential algorithm exactly: an excluded False literal is pushed towards inclusion one step at a time, but once it crosses the boundary the clause no longer matches and receives no further Type II events in that update.

What is exact and what is not. At $B = 1$ Equation (3) reduces to the classical algorithm term by term. For $B > 1$ it is an approximation in precisely the sense mini-batch SGD approximates SGD: the examples within a batch do not see each other's updates. Two secondary effects follow — an automaton can move by more than one step in a commit, and the Type II bound is evaluated against the state at commit time rather than after each event. Section 8.10 measures what this costs. For work where fidelity matters more than throughput, `feedback_mode="sequential"` (or `update(..., sequential=True)`) runs the exact per-example algorithm, still vectorised across clauses and literals.

> **A design decision that measurement reversed**
>
> An obvious refinement is to *thin* the accumulated events — to cap how many Type Ia or Type II events one clause may contribute per batch, on the grounds that the sequential algorithm would have re-evaluated in between and delivered fewer. This was implemented and measured; it made learning distinctly worse. Aggregating all events is therefore deliberate, not an oversight, and is documented as such in the repository so that the next person to have the idea finds the answer before spending a week on it.

6.5 The clause budget is a hard cap

`max_included_literals` = b implements the clause-size constraint of [6]: a matching clause that already includes more than b literals receives no Type Ia feedback, and its Type I events all act as Type Ib (forgetting) until it shrinks. In the sequential algorithm that check runs before every example, so a clause can overshoot by at most one step. A batched commit is different. It aggregates many Type Ia and Type II events, and Type II is never gated by the budget, so a clause under budget at the start of a commit could end it far over. Measured before the fix, 41% of the extracted

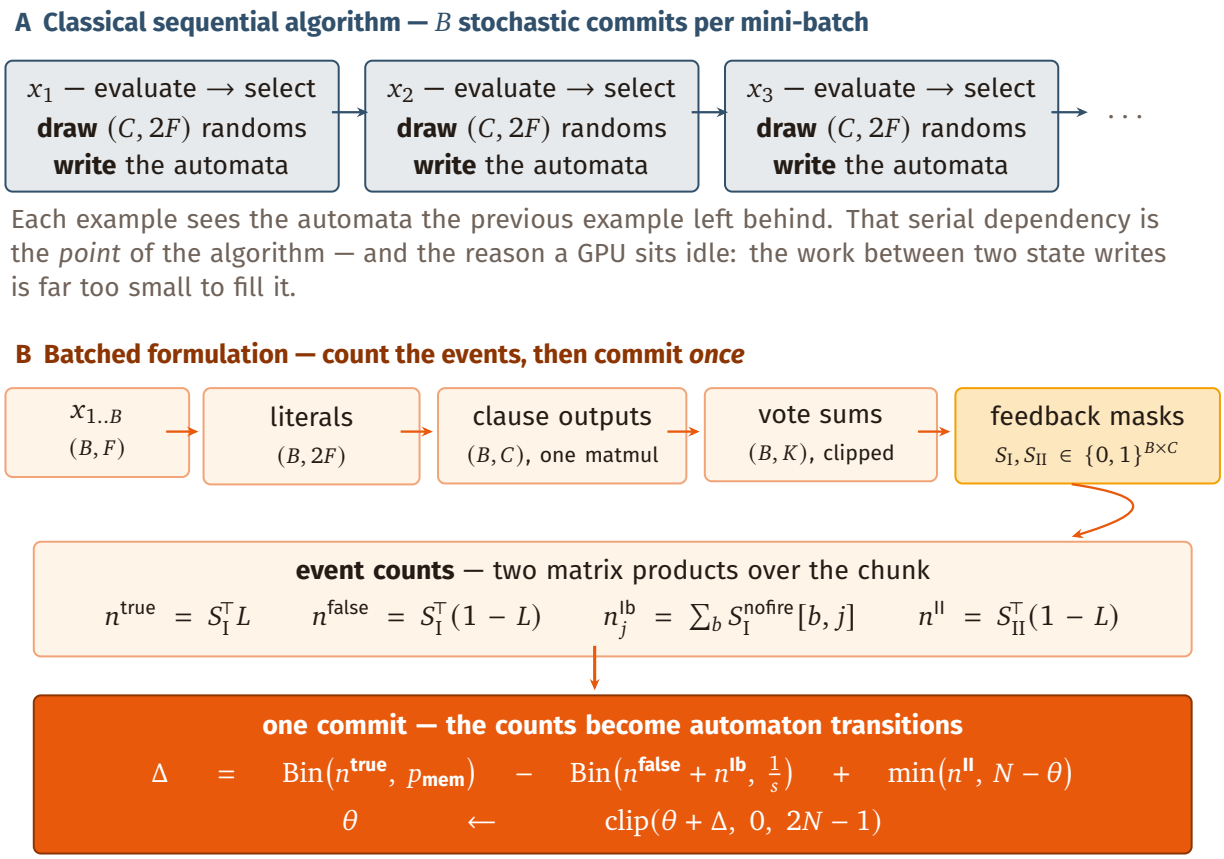


Figure 5. The reformulation. The sequential algorithm (A) performs one stochastic $(C, 2F)$ pass per example. The batched algorithm (B) accumulates the same events as counts and realises them in a single binomial draw per batch. Cost per update becomes almost independent of B, which is exactly the property a GPU rewards.

rules exceeded a budget of 6, and dense segmentation models averaged about 200 literals per clause against a budget of 32.

Since version 0.3.0 the commit enforces the budget directly. With $\mathcal{I}_j$ the literals clause j included before the commit and $\mathcal{I}'_j$ after it, the clause may keep

$$a_j = \max\bigl(0,\ b - |\mathcal{I}_j \cap \mathcal{I}'_j|\bigr)$$

of its newly included literals $\mathcal{I}'_j \setminus \mathcal{I}_j$. If it has more, a uniformly random subset of a_j is kept and the rest are reset to state $N - 1$, just below the include boundary, where one more event would re-include them. Forgetting is untouched, so an oversized clause can still only shrink. The selection is one `argsort` over random keys per offending row and runs only when some clause actually overshoots. The guarantee is the one a user expects from the parameter's name: no clause ends a commit with more than b literals unless it started with more.

A behaviour change, documented as one

Models trained with a budget learn differently under 0.3.0 than under 0.2.0, and results measured with a budget before 1 October 2026 are not directly comparable with later ones. Without a budget nothing changed.

6.6 Memory as an explicit budget

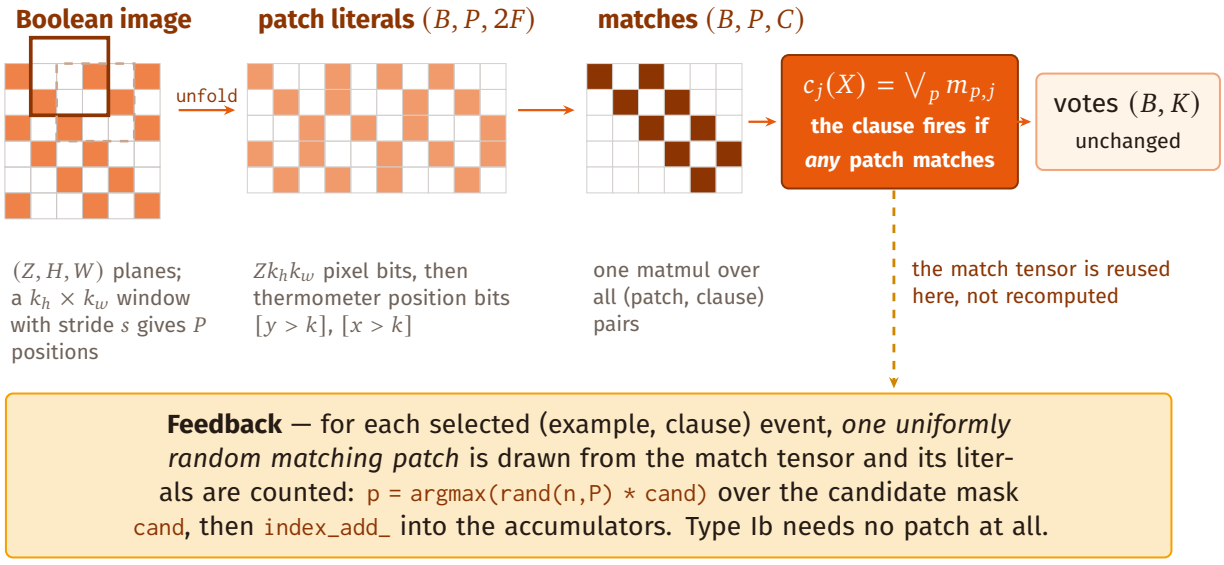
Intermediates, not the model, dominate memory. `torchtsetlin` never silently allocates a tensor proportional to B without a bound: each model declares how many elements one example costs,

$$\text{flat: } 4\,(C + 2F), \qquad \text{convolutional: } 3PC + 2C \cdot 2F + P \cdot 2F,$$

where P is the number of patches per image, and the base class splits the batch into chunks that keep the largest intermediate under `max_chunk_elements` (default 2^{27} elements). Chunking never changes the result of a batched update — accumulation is pure addition — so it is purely a memory-for-nothing trade. Getting the per-example estimate wrong is not benign, however: an under-estimate produces many tiny chunks per batch and a 4–20× slowdown, which is why the estimate is part of the subclass contract rather than a heuristic. The 3D models extend the budget along the other axis: when a single volume exceeds it, the patch axis itself is split into blocks of P_b positions, and the estimate becomes $3P_bC + PC/4 + 2C \cdot 2F + P \cdot 2F$, with the Boolean match tensor counted at one byte per entry (Section 6.8).

6.7 Convolutional machines

A convolutional Tsetlin machine [2] lets a clause look for a pattern *anywhere* in an image instead of memorising absolute positions. Each $k_h \times k_w$ window becomes a patch whose Boolean features are the Zk_hk_w pixels followed — when `position_encoding=True` — by thermometer-encoded window coordinates, $P_y - 1$ bits $[y > k]$ and $P_x - 1$ bits $[x > k]$. A clause is True for the image if it matches at least one patch, and a feedback event updates the clause from *one uniformly random matching patch*.



Drawing the patch here rather than during evaluation keeps *prediction* free of randomness — a (B, P, C) random tensor on every forward pass cost roughly 5× the runtime — and lets Type Ia and Type II draw independently, which matters for coalesced and multi-label models where one clause can receive both in the same update.

Figure 6. Convolutional evaluation. `unfold` produces the patch literals, one matmul produces all (patch, clause) matches, and an OR over patches gives the clause output. The random matching patch is drawn during *feedback counting*, over selected events only.

Implementation-wise this is `torch.nn.functional.unfold`, one matmul, and an `any` reduction. The

subtlety is *where* the random patch is drawn. Drawing it during evaluation is the obvious choice and it is wrong twice over: it costs a (B, P, C) random tensor and an `argmax` on *every* forward pass, including pure prediction where the draw is discarded — roughly 5× the runtime on MNIST-sized inputs — and it forces Type Ia and Type II to share one patch, which is incorrect for coalesced and multi-label models where a clause can receive both in the same update. `torchtsetlin` therefore returns the per-patch match tensor as an opaque *feedback context* from evaluation and draws inside the feedback counter, over the selected events only. Prediction is RNG-free, and a test pins that it stays so.

The random draw itself is a small trick worth noting: for candidate mask cand $\in \{0,1\}^{n\times P}$, `argmax(rand(n,P) * cand)` selects a uniformly random index among the ones, with no rejection loop and no gather over a variable-length list.

> **Position encoding is a modelling choice, not a default to leave alone**
>
> Position literals let a clause demand *where* it matches. On genuinely translation-invariant tasks they hurt — on the generated `make_shapes` benchmark, accuracy oscillates between 0.5 and 0.9 with them enabled and reaches 1.000 on every seed with them disabled. On MNIST, where digits are centred and position is informative, they are worth about a point of accuracy. Check this flag before suspecting the feedback code.

6.8 Volumes: 3D convolution

Cardiac MRI and CT come as stacks of slices, and cine sequences as stacks of frames. A 2D convolutional machine cannot see how a pattern changes from one slice to the next unless the slices are stacked as channels, which pins every pattern to a fixed depth. `Conv3dTsetlinMachine` lets the window slide in depth as well (Figure 7, right). For a Boolean volume $X \in \{0,1\}^{Z\times D\times H\times W}$, a $k_d \times k_h \times k_w$ window with stride (s_d, s_h, s_w) gives a grid of $P = P_d P_h P_w$ positions, and each patch has

$$F \;=\; Zk_dk_hk_w + (P_d - 1) + (P_h - 1) + (P_w - 1)$$

features: the voxels under the window, then three thermometer blocks $[d > k]$, $[y > k]$, $[x > k]$ for its position. The patches come from three chained `Tensor.unfold` calls and one reshape; there is no new kernel.

The design rests on one observation: the convolutional learning code never reads the geometry. Clause evaluation is an OR over the patch axis, and the feedback draw picks a uniformly random matching patch on that axis; neither cares whether the patches came from a line, a plane or a volume. The 3D mix-in therefore overrides only the encoder, the input coercion and the interpretation helpers, and inherits the learning path. Four results make that safe. Their proofs are in the repository (`plans/conv3d.md`) and each one is pinned by a test.

1. **Exact reduction to 2D.** With $D = k_d = 1$ the 3D literal tensor is bitwise identical to the 2D one: the patch order, the feature order and the position bits all coincide, and the depth thermometer has $P_d - 1 = 0$ bits. Because the learning code is shared, the whole training run is identical under the same seed. The test compares the automata after training, bit for bit.

2. **Position literals describe boxes.** A conjunction of position literals admits exactly an axis-aligned box of window positions, $[a_d, b_d] \times [a_h, b_h] \times [a_w, b_w]$, and every box costs at most six literals. A one-hot position code would need P features — 33,640 for a $12 \times 64 \times 64$ volume with a $3 \times 7 \times 7$ window — and could not express a box as one conjunction. `clause_region` decodes the box back.

3. **Depth as channels is the special case $k_d = D$, and it is strictly weaker.** Stacking the slices as channels of a 2D machine gives exactly the 3D machine with a full-depth window. On the

concept "some voxel is on" with D slices, the 3D machine with $k_d = 1$ needs one clause; any depth-as-channels machine needs at least D, because each single-voxel positive needs a clause that contains that voxel's positive literal and no other.

4. **Splitting the patch axis is exact.** When one volume exceeds the memory budget, the patches are processed in blocks. The OR over all patches is the OR of the per-block ORs, and the largest random key over a union is the largest of the per-block maxima, so a running "best key, best patch" carried across blocks draws from the same uniform distribution over matching patches. The draw is equal in distribution, not bit for bit; with a single block the code takes the 2D path, which keeps result 1 exact.

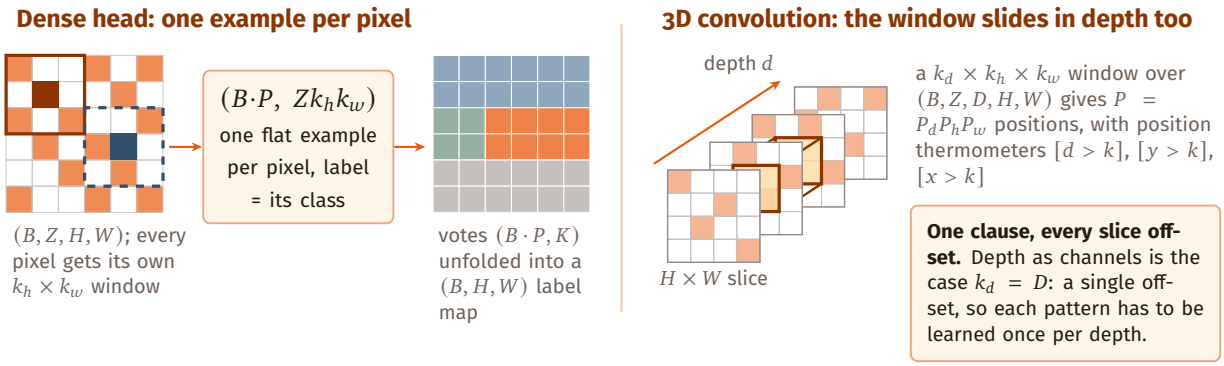


Figure 7. Two new input structures over the same learning code. Left: a dense head gives every pixel its own window and its own example, so the votes form a label map instead of being ORed into one answer. Right: a 3D window also slides in depth, so one clause covers a pattern at every slice offset, where depth-as-channels pins it to one.

Result 3 is visible in data. `data.make_3d_shapes` generates two shapes that differ only in the order of their slices — every individual slice is a plus or a dot in both classes — at a random position in an $8 \times 12 \times 12$ volume. With 20 clauses per class and the same hyper-parameters, the 3D machine with a $3 \times 3 \times 3$ window reaches 1.000 test accuracy on five of five seeds; the 2D machine with the slices stacked as channels reaches 0.83–0.98, and a flat machine 0.73–0.80. Position literals cost the 3D machine what they cost in 2D on a translation-invariant task (accuracy falls to 0.48), so they are off in this comparison. Throughput and memory are in Section 8.8.

6.9 Dense prediction: segmentation

A convolutional machine cannot segment. The OR over patches that gives it translation tolerance also erases *where* a pattern matched: afterwards it knows that a building is in the picture, not which pixels are building. `SegmentationTsetlinMachine` keeps the patch axis instead and folds it into the batch axis (Figure 7, left). With `padding="same"` and stride 1 every pixel has one $k_h \times k_w$ neighbourhood, and that neighbourhood is an ordinary example of a flat machine whose label is the pixel's class:

$$\underbrace{(B, Z, H, W)}_{\text{planes}} \xrightarrow{\text{unfold}} \underbrace{(B \cdot P,\ Zk_hk_w)}_{\text{one example per pixel}} \to \underbrace{(B \cdot P,\ C)}_{\text{clauses}} \to \underbrace{(B \cdot P,\ K)}_{\text{votes}} \to (B, K, H, W).$$

With $B' = B \cdot P$, the vote sums, Type I/Type II selection, event counting and the commit all apply unchanged. `predict` returns a (B, H, W) label map and `vote_map` a (B, K, H, W) vote surface, while `forward` returns the folded $(B \cdot P, K)$ votes, which is what keeps the trainer, the metrics and the interpretation tools working unchanged.

The commit size is a real hyper-parameter. One batch of eight 32×32 images is 8,192 pixel-examples, and committing all their feedback at once is a mini-batch of 8,192 — far past the point where batched feedback stops matching the classical algorithm (Section 8.10). `patches_per_commit` (default 128) bounds it. Measured over three seeds on a toy where the classes are only separable from the neighbourhood, mean IoU is flat up to 128 and then falls away, and the seed-to-seed spread grows with it:

`patches_per_commit`	64	**128**	256	512	1024	4096
mean IoU	0.741	**0.749**	0.659	0.620	0.591	0.527
seed s.d.	0.002	0.009	0.056	0.097	0.029	0.073
relative time	21×	11×	5.7×	3.0×	1.7×	1×

Context without gradients. A 3×3 patch of building wall and a 3×3 patch of road look alike once lighting varies; telling them apart needs to know where the horizon is, which is not inside the window. Two Tsetlin-native mechanisms supply it. Thermometer-encoded *position literals* let a clause say "only in these rows" (an integer `position_encoding` bounds the feature count on large images). A *multi-scale pyramid*, `PyramidEncoder`, OR-pools the Boolean planes, takes the same window at each level and resamples every level back to full resolution, so a clause reads both the neighbourhood and the coarse layout around it. On generated street scenes (32×32, four classes, 300 clauses per class), building IoU goes from 0.505 with neither, to 0.699 with the pyramid, 0.894 with position literals and 0.921 with both, and mean IoU from 0.798 to 0.959. There is no float parameter anywhere.

> **Report the floor**
> On the same scenes, a baseline that never sees a pixel — it labels every pixel with the most common class of its row — already scores 0.876 pixel accuracy, because the scenes have a stable vertical layout. It scores 0.410 mean IoU and zero on two of the four classes. Any dataset with a consistent layout has a floor like this, which is why the library's examples report mean IoU and the floor alongside pixel accuracy.

Imbalance. Background dominates real segmentation data, and a Tsetlin machine has no loss to reweight. The lever is which pixels produce feedback: `class_feedback_p` sets a per-class probability that a pixel takes part in an update, and `data.balanced_class_probabilities` computes the equalising choice. It is a rescue, not a default. It lifts a pyramid-only run from 0.874 to 0.927 mean IoU, and costs 0.05 once position literals are on, because by then the classes are separable anyway and discarding most of the road pixels is pure loss.

6.10 Booleanization

The literals are the vocabulary of every rule the machine can learn, so Booleanization is the most consequential modelling decision in a Tsetlin machine project — and in practice it moves accuracy more than the model hyper-parameters do. `torchtsetlin` treats encoders as first-class modules (Table 5): they are `nn.Module`s with a `fit`/`transform` protocol, they move to a device, they are saved in a `state_dict`, and they compose.

Encoders also carry the inverse mapping, which is what keeps rules readable. A thermometer encoder's `decode_range` turns a clause's included bits back into an interval in the original units, and `aggregate_literal_importance` sums literal importances back over the bits of one original column.

Table 5. Booleanization encoders. F input columns, b bits per column.

Encoder	Output	Use and why
`Binarizer`	F	a fixed threshold per element, the original MNIST recipe (`pixel > 0.3`)
`ThermometerEncoder`	$F \cdot b$	ordinal bits $[v \geq t_1, \ldots, v \geq t_b]$; two literals then express an interval (`age>=30 AND NOT age>=50`). Thresholds by quantile, uniform width or unique value
`OneHotEncoder`	$\sum_i n_i$	one bit per category; rules read as `colour=red`
`BitPlaneEncoder`	$F \cdot b$	place-value bits of integers; compact codes
`AdaptiveThresholdEncoder`	F	local Gaussian/mean thresholding, the standard recipe for MNIST-like images in the convolutional TM literature
`ColorThermometerEncoder`	$C \cdot b$ planes	per-channel thermometer for RGB
`HypervectorEncoder`	d	sparse random hypervectors bundled with OR, optionally binding position by cyclic shift — a compact encoding for large vocabularies [13]

A GPU bug worth documenting

Encoders keep their thresholds on the CPU, so handing one a CUDA tensor triggers a device-to-host copy. Issued with `non_blocking=True` into pageable memory, such a copy returns *before* the data lands, and the encoder thresholds stale memory — about 40% wrong bits, no error, just a quietly worse model. All coercion in `torchtsetlin` routes through a helper that sets `non_blocking` only for CUDA destinations. The regression test has to allocate on the device (`torch.rand(..., device=dev)`) rather than moving a host tensor across, or the race never occurs and the test passes vacuously.

6.11 Interpretation

Three levels of explanation are available, all closed-form and all computed from the state rather than by probing the model.

Rules. `model.rules()` prints each non-empty clause as `IF <conjunction> THEN <class>`, with its weight where weights exist. Feature names flow through from the encoder, so the conjunction is in the user's vocabulary.

Global importance. Following [4], the strength of literal l for output k is the weighted fraction of positive clauses of k that include it,

$$g_k[l] = \frac{1}{m_k} \sum_{j\,:\,W_{jk}>0} W_{jk}\,\text{include}_{jl}, \qquad m_k = \left|\{\, j : W_{jk} > 0 \,\}\right|.$$

Local importance and explanations. For one example, the same quantity restricted to the clauses that actually matched and to the literals that are True gives a per-example attribution; `interpret.explain` returns the matching clauses ranked by weight, which is an account of the prediction in the model's own terms rather than a surrogate.

Spatial explanations. For convolutional models, `clause_patch` returns the pattern a clause looks for as a stencil over the window — (Z, k_d, k_h, k_w) for volumes — and `clause_region` decodes its position literals into the box of window positions it may fire in. For segmentation,

`interpret.explain_pixel` lists the clauses that decided one pixel, decoded over *absolute* image coordinates (`p[1,21,13]` is "plane 1 of pixel (21, 13)"), so a dense prediction can be audited pixel by pixel rather than attributed by a saliency heuristic.

6.12 Correctness

The library is tested with 110 tests, parametrised over a `device` fixture so that every device-marked test runs twice on a GPU machine (174 runs in all). The tests that matter most are not the smoke tests but the semantic pins: brute-force reference implementations of clause evaluation and of convolutional patch semantics; the empty-clause train/eval asymmetry; the RNG-free prediction path for convolutional models; the hard clause budget under batched commits; bitwise equality of the 3D and 2D models at depth one; a χ^2 test that the random matching patch is uniform, with and without patch blocking; the encoder device-transfer race described above; and a packaging test asserting that the version agrees between `pyproject.toml` and `__init__.py` and that every subpackage ships in the wheel — which exists because a `.gitignore` rule once silently excluded `torchtsetlin/data` from a release.

> **In short.** Clause evaluation is a matmul against the include mask; learning is event counting plus one binomial draw per commit. Those two identities are the whole library: everything else — models, convolution, chunking, the trainer — is organisation around them.

7 Usage

The API has two levels: a bare loop over `model.update` for people who want control, and a `Trainer` for people who want Keras. Both are shown below; every listing is runnable as written, and all of them have full counterparts in the repository's `examples/` directory.

7.1 Thirty seconds

```python
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")

# update() is the analogue of loss.backward(); optimizer.step()
for epoch in range(50):
    model.train()
    for i in range(0, 5000, 10):
        model.update(x_train[i:i+10].cuda(), y_train[i:i+10].cuda())

model.eval()                                  # empty clauses now vote for nothing
acc = (model(x_test.cuda()).argmax(1) == y_test.cuda()).float().mean()
print(model.rules()[:2])
# ['IF x0 AND NOT x1 THEN 1', 'IF x1 AND NOT x0 THEN 1']
```

The last line is the point of the exercise: the model has recovered the generating rule of the Noisy XOR benchmark from data with 40% label noise, and printed it.

7.2 Tabular data, rules in the original units

Continuous columns become Boolean through a thermometer encoder. Passing the encoder's feature names to the model is what turns `x37` back into `worst perimeter>=105.9`, and `max_included_literals` keeps the printed rules short enough to read.

```
from sklearn.datasets import load_breast_cancer
from sklearn.model_selection import train_test_split
import torch, torchtsetlin as tt

tt.seed_everything(0)
data = load_breast_cancer()
x_tr, x_te, y_tr, y_te = train_test_split(data.data, data.target, test_size=0.3, random_state=0)
x_tr, x_te = torch.tensor(x_tr).float(), torch.tensor(x_te).float()

enc = tt.data.ThermometerEncoder(n_bits=8, strategy="quantile").fit(x_tr)
xb_tr, xb_te = enc(x_tr), enc(x_te)

model = tt.TsetlinMachine(xb_tr.shape[1], 2, n_clauses=100, T=20, s=4.0,
                          max_included_literals=6)
model.feature_names = enc.feature_names(list(data.feature_names))   # readable literals
model.class_names   = list(data.target_names)

trainer = tt.Trainer(model, batch_size=8, callbacks=[tt.EarlyStopping(patience=30)])
trainer.fit((xb_tr, y_tr), epochs=200, val_data=(xb_te, y_te))
print(trainer.test((xb_te, y_te)))                # {'test_accuracy': 0.96...}

for rule in model.rules(polarity=1)[:5]:          # clauses voting *for* a class
    print(rule)

# Which original columns carry the model, with the thermometer bits summed back together
imp = tt.interpret.global_feature_importance(model, class_index=0)
per_feature = tt.interpret.aggregate_literal_importance(
    imp, [i // 8 for i in range(xb_tr.shape[1])])
```

7.3 Convolutional MNIST on a GPU

The convolutional path differs in three places only: the encoder (adaptive thresholding rather than a fixed cut), the model class, and the patch geometry. Keeping the encoder on the compute device means the images never travel back to the host.

```
import torch, torchtsetlin as tt
dev = "cuda"

enc = tt.data.AdaptiveThresholdEncoder(block_size=11, C=2, method="gaussian").to(dev)
booleanize = lambda x: enc(x * 255.0)

x_train, y_train = tt.data.load_mnist_boolean("./data", train=True,
                                              encoder=booleanize, device=dev)
x_test,  y_test  = tt.data.load_mnist_boolean("./data", train=False,
                                              encoder=booleanize, device=dev)

model = tt.ConvTsetlinMachine(
    n_classes=10, n_clauses=200, T=100, s=10.0,
    patch_size=10, stride=1, position_encoding=True, weighted=True,
    input_shape=tuple(x_train.shape[1:]),          # (1, 28, 28)
)

trainer = tt.Trainer(model, device=dev, batch_size=32, callbacks=[
    tt.EarlyStopping(monitor="val_accuracy", patience=5),
    tt.ModelCheckpoint("mnist_conv.pt", monitor="val_accuracy"),
```

```
    tt.CSVLogger("mnist_conv.csv"),
    tt.StateSummaryLogger(),                        # mean literals/clause, empty clauses
])
trainer.fit((x_train, y_train), epochs=10, val_data=(x_test, y_test))

votes = trainer.predict((x_test, y_test), return_votes=True)
print(tt.metrics.classification_report(votes, y_test))

tt.viz.plot_conv_clauses(model, range(16))          # what each clause looks for
```

Hyper-parameter scheduling

Tsetlin hyper-parameters are ordinary Python attributes, so annealing one is a callback and not a framework feature: `tt.HyperparameterSchedule("s", lambda epoch: 10.0 - 5.0 * epoch / 30)`, or the same for `drop_clause_p`. `model.state_summary()` reports mean included literals per clause and the number of empty clauses — the two numbers that tell you whether s is in the right range.

7.4 Volumes and dense prediction

The 3D and segmentation models change the input shape and the output, not the workflow. A volume classifier takes (B, Z, D, H, W) Booleans; a single-plane volume may be passed as (B, D, H, W).

```
x_train, y_train = tt.data.make_3d_shapes(2000, size=(8, 12, 12))   # (n, 1, 8, 12, 12)
x_test,  y_test  = tt.data.make_3d_shapes(1000, size=(8, 12, 12), seed=1)

model = tt.Conv3dTsetlinMachine(
    n_classes=2, n_clauses=20, T=15, s=10.0,
    patch_size=3,                # (kd, kh, kw); an int means a cube
    stride=(1, 2, 2),            # in-plane stride cuts the patch count fourfold
    position_encoding=False,     # translation-invariant task: leave position out
)
tt.Trainer(model, device="cuda", batch_size=16).fit((x_train, y_train), epochs=10)

model.clause_patch(0)            # (Z, kd, kh, kw) stencil: +1 on, -1 off, 0 free
model.clause_region(0)           # {"d": (lo, hi), "y": (lo, hi), "x": (lo, hi)}
tt.viz.plot_conv_clauses(model, range(8))     # 3D clauses drawn slice by slice
```

A segmentation model takes (B, Z, H, W) planes and (B, H, W) label maps, and the trainer reports pixel accuracy, mean IoU, mean Dice and per-class IoU on its own.

```
rgb, labels = tt.data.make_scenes(300, size=32)          # (n, 3, 32, 32) RGB, (n, 32, 32)
enc = tt.data.ColorThermometerEncoder(n_bits=4, value_range=(0.0, 1.0))
pyramid = tt.data.PyramidEncoder(scales=(1, 2, 4), patch=3)  # neighbourhood + coarse layout
x = pyramid(enc(rgb).float()).float()                        # Boolean planes per pixel
x_tr, y_tr, x_te, y_te = x[:220], labels[:220], x[220:], labels[220:]

model = tt.SegmentationTsetlinMachine(
    n_classes=4, n_clauses=300, T=60, s=10.0, weighted=True,
    patch_size=1,                # the pyramid already gathered the context
    position_encoding=8,         # 8 thermometer bits per axis
    patches_per_commit=128,      # commit size in pixels: the fidelity knob
)
trainer = tt.Trainer(model, device="cuda", batch_size=8)
trainer.fit((x_tr, y_tr), epochs=8)
print(trainer.test((x_te, y_te)))           # pixel accuracy, mean IoU, Dice, IoU per class

model.eval()
label_map = model.predict(x_te.cuda())                   # (80, 32, 32)
print(tt.interpret.explain_pixel(model, x_te[0].cuda(), 12, 20))   # why this pixel?
```

7.5 Explaining a single prediction

```
expl = tt.interpret.explain(model, x_test[0], feature_names=model.feature_names)
print(expl)                    # shape of the output; the numbers depend on the run
# prediction: 1  votes: [-14.0, 17.0]
#   [+3 -> output 1] clause 42: IF worst perimeter>=105.9 AND NOT mean texture>=21.8
#   [+2 -> output 1] clause 77: IF worst concave points>=0.15
#   ...

torch.save(model.state_dict(), "model.pt")          # ordinary PyTorch checkpointing
reloaded = tt.TsetlinMachine(None, 2, n_clauses=100, T=20, s=4.0)   # lazy: shape from ckpt
reloaded.load_state_dict(torch.load("model.pt"))
```

The explanation is not a surrogate: those clauses are the model, and the listed votes add up to the reported vote sums.

7.6 Building a new variant

A new model supplies an output layer and a feedback policy; the base class provides everything else, including chunking, dropout, the batched commit and the whole workflow layer.

```
class MyTsetlinMachine(tt.TsetlinMachineBase):
    def __init__(self, n_features, n_clauses, T, **kw):
        self.T = T
        super().__init__(n_features, n_clauses, **kw)

    def _votes(self, clause_out, clamp=True):        # (B, C) bool -> (B, n_outputs)
        ...

    def _coerce_targets(self, y, n, device):         # validate / convert targets
        ...

    def _select_feedback(self, votes, y, clause_out):  # -> (type_i, type_ii, aux) masks
        ...
```

Or skip the class hierarchy entirely and compose the functional layer directly — useful when prototyping a different feedback rule:

```
from torchtsetlin import functional as F

lit     = F.to_literals(x)                                  # (B, 2F)
include = (state >= N).float()
c       = F.clause_outputs(lit, include, empty_value=True)  # (B, C)
v       = F.vote_sums(c, weights, T)
n_true, n_false, n_ib = F.type_i_counts(lit, sel_fire_i, sel_nofire_i)
n2      = F.type_ii_counts(lit, sel_fire_ii)
F.apply_feedback(state, n_true, n_false, n_ib, n2, n_states=N, s=s)
```

It is plain tensor code, so `torch.compile`, a custom CUDA kernel or an alternative sampling scheme can be dropped in without touching anything else.

8 Benchmarks

8.1 Method

Every number in this section comes from one run of the benchmark driver shipped with the repository, `benchmarks/run_benchmarks.sh`, on the machine in Table 6. Each point is the median of three timed blocks of steady-state calls after a warm-up, with the device synchronised once per block. The micro-benchmarks use synthetic Boolean data at 20% density, resident on the device unless stated otherwise. The driver writes a JSON record per measurement and regenerates the figures and tables from it, so the whole section can be reproduced — or replaced with another machine's numbers — with one command.

Table 6. Measurement environment.

CPU	AMD Ryzen 9 3950X, 16 cores / 32 threads (PyTorch using 16 threads)
GPU	NVIDIA GeForce RTX 3090 (24 GB)
Software	PyTorch 2.10.0+cu128 / CUDA 12.8, Python 3.10.11, `torchtsetlin` 0.3.1
Measured	12 October 2026
Reference config	784 Boolean features, 500 clauses per class, 10 classes, unless stated otherwise

What this section does and does not measure

It measures *this* library on two devices: the same Python code, the same algorithm, one run on a CPU and one on a GPU. It is not a comparison against other Tsetlin machine implementations — no such comparison was run — and it is not a state-of-the-art accuracy claim. Where accuracy appears it is there to show that the batched update learns what it should, not to compete with 400-epoch results from the literature.

8.2 Throughput against batch size

One `update()` call touches every automaton in the machine, so its cost depends far more on the *size of the machine* than on how many examples are in the batch. That is the whole point of the reformulation in Section 6.4, and Figure 8 is what it buys: on the GPU, training throughput rises from 427 examples/s at batch 1 to 220,028 at batch 4096 — a factor of 515 for feeding the same kernel more work.

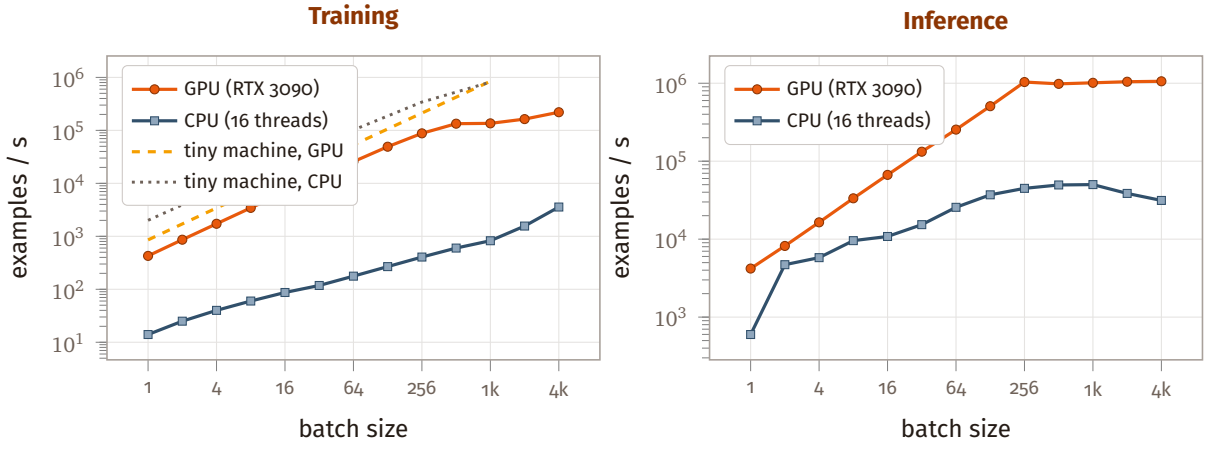


Figure 8. Throughput against batch size for a 784-feature, 500-clause-per-class machine (log–log). Training on the GPU scales almost linearly with batch size until the device saturates. The dashed line is the counter-example: a 12-feature, 10-clause machine is less work than a kernel launch, and only draws level with the CPU at batch 1024.

At batch 256 the same model trains at 87,869 examples/s on the GPU against 406 on the CPU — 216×. Inference behaves differently: the CPU forward pass is already cheap, so the gap there tops out around 34×, reaching 1.06×10^6 examples/s on the GPU.

The dashed series is the honest counterpoint. A Noisy-XOR-scale machine (12 features, 10 clauses per class) is *slower* on the GPU than on the CPU for every batch size below 1024. Small machines belong on a CPU, and `torchtsetlin` makes that a one-word change rather than a different package.

8.3 Throughput against model size

Both axes of model size cost the same thing — an update is $O(C \cdot 2F)$ element-wise work — so both throughput curves fall roughly as $1/x$. The interesting quantity is the *ratio*, in Figure 9: the CPU falls off immediately while the GPU stays flat until the machine is large enough to fill it, so the advantage *widens* with model size, from 50× at 50 clauses per class to 273× at 4000, and from 69× at 128 features to 287× at 8192.



Figure 9. The GPU advantage grows with the model. Big machines — the ones the Tsetlin machine literature actually uses for images — are exactly the ones worth moving to a GPU.

This matters for a practical reason. The published configurations that reach competitive accuracy on images use thousands of clauses per class; on a CPU those are overnight runs, and the resulting iteration speed is what keeps people away. At 2000 clauses per class the GPU trains at 25,917 examples/s against 96 on the CPU.

8.4 The model zoo

Weighted and coalesced variants cost about what the plain machine costs — they change the output layer and the feedback policy, not the per-clause work that dominates (Table 7). The convolutional models are the outlier at roughly 3× the cost per example on the GPU, which is arithmetic rather than overhead: a 10×10 window over a 28×28 image is 361 patches, and every patch is evaluated against every clause. Their *inference* cost is the striking one — 24,822 examples/s against 490,091 for the flat machine — because prediction is where the patch dimension has no feedback work to hide behind.

GPU peak memory during training is 262 MB for the flat models and 182–225 MB for the convolutional ones at these configurations — small enough that clause budget, not memory, is the binding constraint on most cards.

Table 7. All models on a 28×28 Boolean input, batch 128. Throughput in examples/s. Each model is measured at its natural configuration: 5000 clauses over 784 flat features, against 2000 clauses over 136 per-patch features for the convolutional ones. The CPU column therefore compares two different amounts of work, not two implementations of the same one.

	Training			Inference		
Model	**CPU**	**GPU**	**speedup**	**CPU**	**GPU**	**speedup**
`TsetlinMachine`	269	48,433	180×	31,087	490,091	16×
`TsetlinMachine` (weighted)	268	48,228	180×	32,101	451,039	14×
`CoalescedTsetlinMachine`	178	42,684	240×	34,131	493,050	14×
`RegressionTsetlinMachine`	215	56,935	265×	42,894	541,666	13×
`ConvTsetlinMachine`	559	14,722	26×	710	24,822	35×
`ConvCoalescedTsetlinMachine`	393	15,116	38×	702	24,627	35×

8.5 Where an update actually goes

Figure 10 breaks one `update()` call into stages. The result is unambiguous: `apply_feedback` — the stochastic increment and decrement of the whole $(C, 2F)$ state — is 92% of the 624 ms CPU call and 42% of the 2.8 ms GPU call. A standalone `torch.binomial` draw of the same shape costs 266.5 ms on the CPU against 0.22 ms on the GPU, making it the single most expensive operation in the commit.

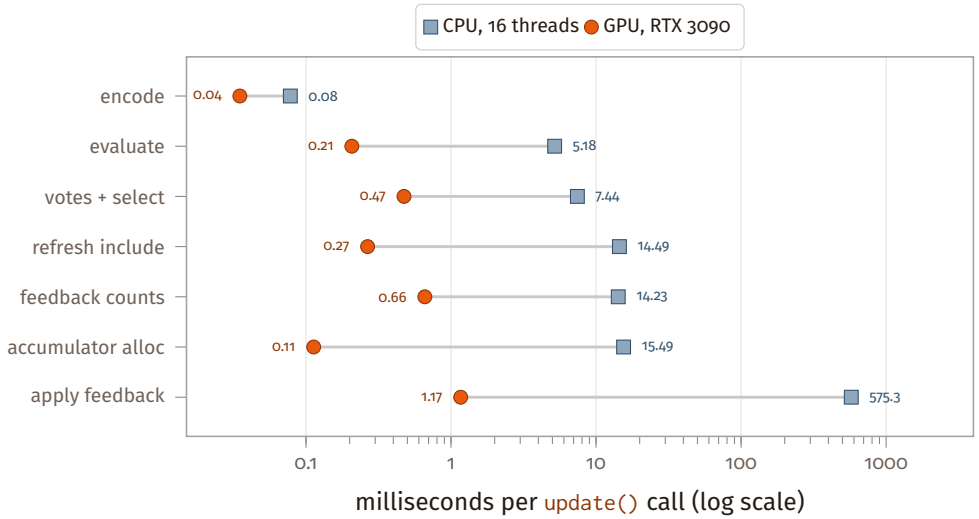


Figure 10. Stage timings for one update (5000 clauses, 784 features, batch 256). Note the log scale: the CPU total is 624.1 ms and the GPU total 2.85 ms. Everything before the commit — encoding, clause evaluation, votes and the feedback-count matmuls — is comparatively free.

This is also the quantitative argument for batching. A batched update draws once per commit; a sequential one draws once per example. Whatever the batch size, the expensive part of the call happens exactly once.

8.6 CPU threads

More CPU threads buy surprisingly little in training — 1.3× from 1 to 32 threads (Figure 11). The batched update is a chain of large element-wise operations over the state, and the binomial draw that dominates it does not thread. Inference scales much better, 10.8× over the same range, because the forward pass is one large matmul-shaped reduction. If a CPU run is too slow, the lever is the clause budget or the batch size, not `torch.set_num_threads`.

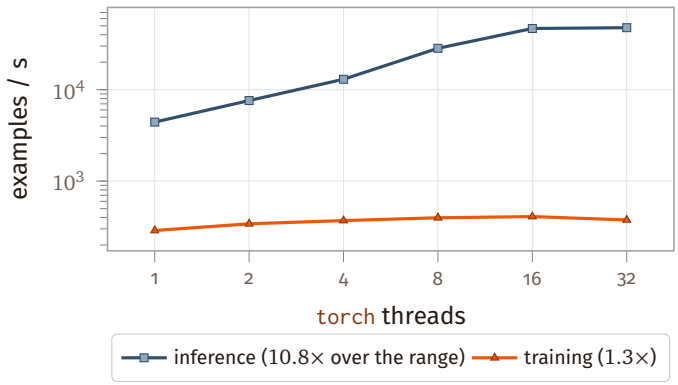


Figure 11. CPU thread scaling (5000 clauses, 784 features, batch 256).

8.7 Where the data lives

Keeping the Boolean dataset in host memory and copying each mini-batch across costs about 1% of throughput on this machine — the update is long enough to hide the transfer. Streaming a dataset that does not fit in VRAM is therefore free in practice. The copy that *does* bite is the one in the other direction, into a CPU-resident encoder, which is the hazard described in Section 6.

8.8 Segmentation and volumes

The two newer input structures multiply the work per input by the number of window positions, so their throughput is quoted per pixel and per volume.

Dense segmentation. A 3×3 dense model on 32×32 inputs (four planes, four classes, 100 clauses per class, eight images per batch) trains at 34,942 pixels/s on the CPU and 117,649 on the GPU at the default commit size of 128 pixels, and predicts at 1.09×10^6 and 2.34×10^7 pixels/s. The training speedup is small, 3×, and Figure 12 shows why: at 128 pixels per commit the GPU spends its time on commits rather than on pixels. Committing once per batch is 49× faster on the GPU (5.8 million pixels/s), but commits that large are the ones Section 6.9 shows costing almost a third of the mean IoU (0.749 at 128 pixels, 0.527 at 4,096). The commit size is a fidelity knob that happens to also be the speed knob, and the default sides with fidelity.

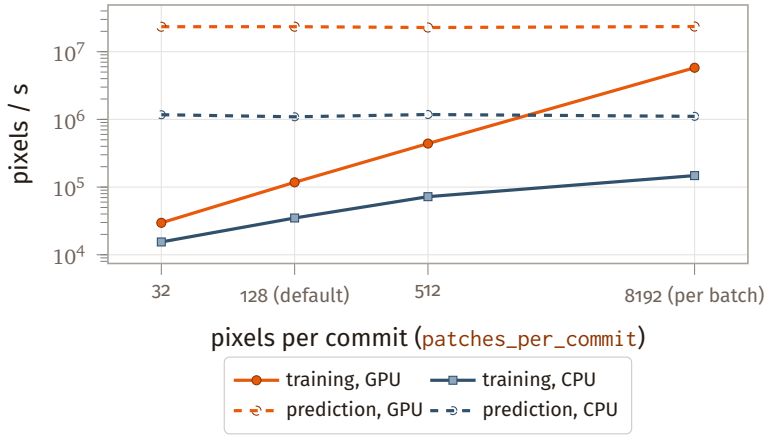


Figure 12. Dense segmentation throughput against commit size (3×3 windows, 32×32 images, batch of eight; log–log). Prediction does not depend on the commit size. The rightmost point is one commit per batch (8,192 pixels).

3D convolution. Table 8 covers volumes from $8 \times 32 \times 32$ to a cardiac-sized $12 \times 64 \times 64$. On the GPU the large volume with a $3 \times 7 \times 7$ window has 33,640 window positions and trains at 656 volumes/s; an in-plane stride of 2 cuts the positions fourfold and runs 3.0× faster. The GPU advantage is large for the same reason it is large for big flat machines: every volume is tens of thousands of patch evaluations.

Table 8. 3D convolution, one Boolean plane. The first four rows use 2 classes × 100 clauses at batch 4; the last three 10 classes × 200 clauses (2000 in all) at batch 2. Throughput in volumes/s; memory is GPU peak during training.

		Training			Inference	
Volume, window	**patches**	**CPU**	**GPU**	**speedup**	**GPU**	**memory**
$8 \times 32 \times 32$, $3 \times 5 \times 5$	4,704	166.1	2,451.8	15×	10,209	67 MB
$16 \times 32 \times 32$, $3 \times 5 \times 5$	10,976	64.1	2,054.6	32×	4,761	147 MB
$12 \times 64 \times 64$, $3 \times 7 \times 7$, stride $(1, 2, 2)$	8,410	61.6	1,991.5	32×	4,423	153 MB
$12 \times 64 \times 64$, $3 \times 7 \times 7$	33,640	12.4	655.7	53×	898	533 MB
2000 clauses, one block	33,640	4.5	185.4	42×	206	450 MB
2000 clauses, auto blocks (18,982)	33,640	4.1	170.6	41×	198	369 MB
2000 clauses, blocks of 2,048	33,640	5.3	134.6	25×	177	365 MB

The last three rows test patch blocking (Section 6.8), and the result is more modest than the design analysis predicted. With 2000 clauses one volume no longer fits the memory budget, and automatic blocking lowers peak memory from 450 to 369 MB — 18% — for an 8% loss of throughput; much smaller blocks save nothing further and cost a quarter of the speed. With 200 clauses blocking does not engage at all, because ordinary batch chunking already keeps the intermediates inside the budget. The design analysis had estimated about 1 GB of intermediates per volume; that was a worst case, not what the chunked code allocates. In practice, the in-plane stride is the cheaper lever and blocking is the safety net for very large clause banks.

8.9 End to end

Micro-benchmarks flatter whichever device wins the micro-benchmark, so Table 9 reports whole training runs with `Trainer.fit` included — data loading, epoch loop, validation and all.

Table 9. End-to-end MNIST runs with `Trainer.fit` included. The flat model's accuracy matches across devices to within 0.0006, as it should — only the wall clock moves. The single-epoch convolutional run is noisier (0.847 against 0.804): that is the variance of one pass over 10 000 examples, not a device difference.

Model	**Device**	**train ex.**	**epochs**	**s / epoch**	**test acc.**
`TsetlinMachine` 500/class	CPU	60,000	3	113.45	0.9542
	GPU	60,000	3	**1.54**	0.9548
`ConvTsetlinMachine` 200/class, 10×10	CPU	10,000	1	15.49	0.8470
	GPU	10,000	1	**0.70**	0.8035

A 113-second epoch becomes a 1.5-second one (74×); the convolutional run goes from 15.5 seconds to 0.7 (22×). That is the difference between a hyper-parameter sweep you plan and one you simply run.

On accuracy: the flat 500-clause machine reaches 95.8% after five epochs, and the convolutional model 97.2% after three epochs at batch 32 (200 clauses per class, $T = 100$, $s = 10$, weighted). The published figures for much larger budgets — around 98% for 2000 clauses per class after 400 epochs, and around 99% for convolutional machines with 8000 clauses per class — are what the

same code should reach given the same budget and patience; they are not claimed here.

8.10 What batching costs in accuracy

The batched update is an approximation, so the honest question is how much fidelity it gives up. Figure 13 answers it on Noisy XOR — the benchmark of the original paper, deliberately chosen because its generating rule is known, so a failure to learn is unmistakable. Twenty clauses, $T = 15$, $s = 3.9$, 100 states, 2000 training examples with 40% label noise, three seeds per configuration, with the number of epochs raised as the batch grows so that every configuration sees the data a comparable number of times.

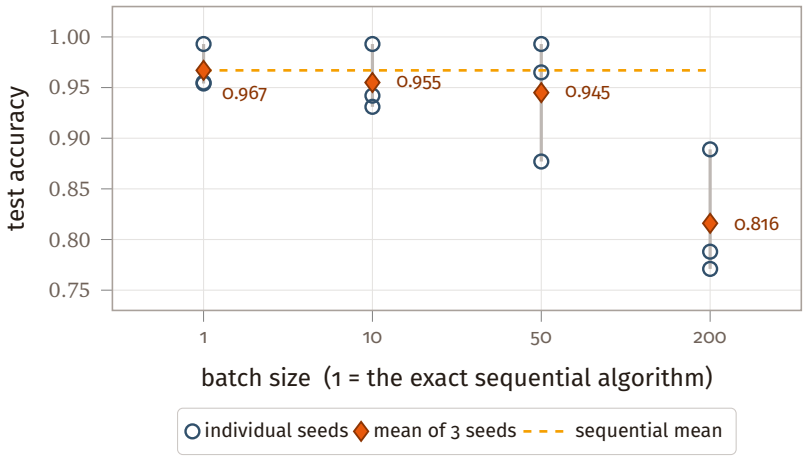


Figure 13. Noisy XOR accuracy against batch size (3 seeds). Batches up to about 50 land in the same band as the exact sequential algorithm; 200 degrades clearly. Seed-to-seed spread is wide on this task — which is itself the reason to read the individual points rather than only the means.

Batch sizes of roughly 8–64 keep the accuracy of the sequential algorithm while running one to two orders of magnitude faster on a GPU. At 200 the degradation is unambiguous. The practical recommendation the library documents follows directly: start at 16–64 for flat models and 8–32 for convolutional ones, and if accuracy lags, halve the batch or add epochs before touching T and s.

The cost of exactness is measurable too. At batch 32, `feedback_mode="sequential"` runs 7–29× slower than the batched update depending on model scale and device, because it performs 32 commits where the batched path performs one. That is the trade, stated in numbers: *batching buys one to two orders of magnitude and costs accuracy only once the batch grows past a few tens of examples.*

In short. Reformulating the update as event counting plus one binomial draw makes the cost of an update nearly independent of batch size — which is what turns a 113-second MNIST epoch into a 1.5-second one, and what makes a clause budget of thousands a routine choice rather than an overnight one.

9 An interactive review of the literature

Choosing hyper-parameters, datasets and baselines for a Tsetlin machine experiment means reading the literature for numbers it rarely tabulates in one place. The package therefore ships a catalog of every dataset and every reported number from 55 Tsetlin-machine papers published between 2018 and 2026. The papers span the foundational variants, image classification, segmentation, medical and clinical applications, hardware accelerators, generative and sequence models, and

privacy-preserving learning. The catalog covers both the methods each paper proposed and the baselines it compared against.

Table 10. What the literature catalog records.

Table	Rows	One row per…
papers	55	paper: venue, arXiv/DOI, code, the Tsetlin-machine variants tested, the feature it introduces, and a caution flag where a reported number has a specific reliability problem (3 papers)
datasets	128	dataset: data type, task, classes, input shape, download link (95 public, each one checked) and the variants that used it
dataset_usages	252	(paper, dataset) pair: the train/val/test sizes, the split exactly as reported, and how the inputs were Booleanized
results	3,295	reported number: model, metric (204 distinct), value, unit, split, configuration, and the table or figure it came from; 1,974 for Tsetlin-machine models, 1,321 for baselines

9.1 How it was built

Each paper has one JSON record in benchmarks/literature/papers/, and those records are the source of truth. Numbers were transcribed from the paper's tables and text. Where an extracted table was garbled, they were checked against the rendered PDF page. Values that appear only in plots were left out rather than estimated. Dataset names are unified to one identifier (kmnist and kuzushiji-mnist are the same dataset), and a field a paper does not state stays empty. build_catalog.py turns the records into the SQLite file docs/assets/literature/literature.db (1,520 KiB) and a flat CSV.

9.2 Using it

The documentation's *Literature results* page opens the database in the browser with sql.js, so filtering, grouping and ranking run as SQL on the reader's machine, with no server. Its tabs are the results themselves, a best-per-dataset leaderboard, the datasets with their download links and per-paper splits, the papers, and a SQL console. Within any grouping the page stars the best value. Values are compared only when they share the metric, unit and split, and a metric's direction decides what counts as best: higher for accuracy, F_1 or throughput, lower for error, latency, energy or area. The same database backs a Hugging Face Space[1], and the file can be opened with any SQLite client:

```
SELECT p.year, p.title, r.model, MAX(r.value) AS accuracy
FROM results r JOIN papers p USING (paper_id)
WHERE r.dataset_id = 'cifar-10' AND r.metric = 'accuracy' AND r.is_tm = 1
GROUP BY r.paper_id ORDER BY accuracy DESC;
```

Figure 14 shows what the catalog makes easy to see. Tsetlin machines are ahead of the baselines they are reported against on Fashion-MNIST and Reuters R8, close on MNIST and Kuzushiji-MNIST, and behind on CIFAR-10 and IMDb.

> **Reading the numbers**
>
> Every value is *as reported by its authors*; none was reproduced for the catalog. Papers differ in splits, Booleanization, clause budgets and epochs. Some report the peak test accuracy over training, and results measured before 0.3.0 made the clause budget a hard cap are not directly comparable with torchtsetlin runs. Every row carries its configuration, source location and notes, and the page shows them on click.

[1] https://huggingface.co/spaces/vlbthambawita/TsetlinMachines

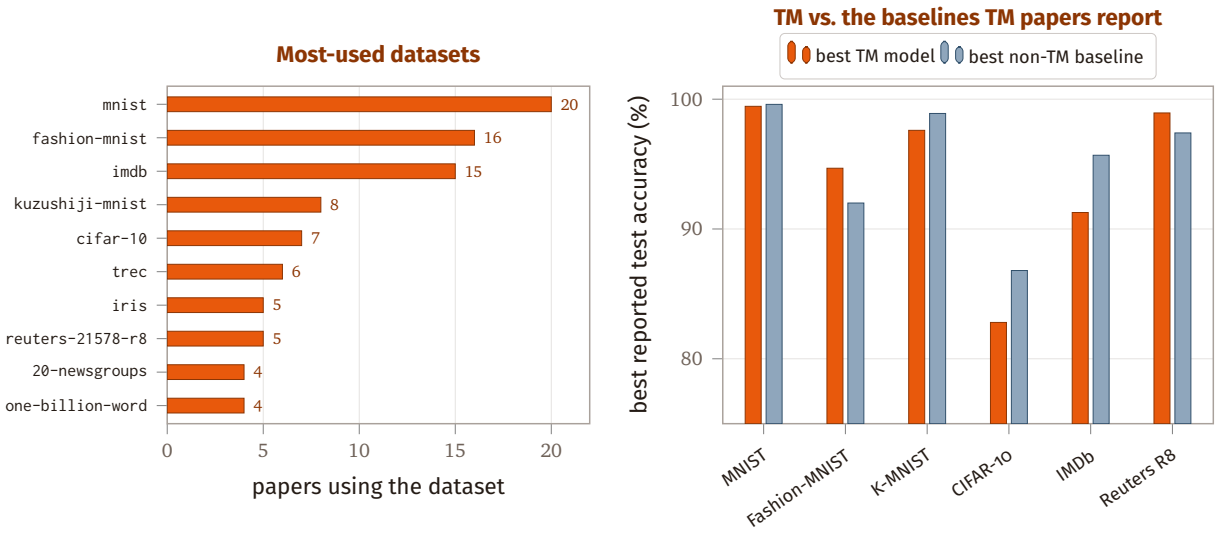


Figure 14. The literature catalog. *Left:* the datasets used by the most papers. *Right:* the best test accuracy reported for a Tsetlin-machine model next to the best non-TM baseline reported in the same body of papers. Flagged papers and federated (per-client) accuracies are excluded. The baselines are those TM papers chose to compare against, not the state of the art in each field.

10 Applications

10.1 What ships with the library

The repository includes ten runnable example scripts and five worked notebooks, chosen so that each one demonstrates a different part of the design rather than a different dataset.

The two segmentation notebooks are worth singling out. They reproduce the structure of published Tsetlin-machine segmentation work — convolutional regression [18] and a U-Net hybrid [19] — and the second builds a dense per-pixel classification head on top of `TsetlinMachine` without any new model code. That notebook is where `SegmentationTsetlinMachine` came from: once the folding trick had been shown to work as a composition, making it a model was a mix-in of the same kind as convolution, plus the metrics, commit-size control and per-pixel explanations a notebook leaves out. The 3D models followed the same path in reverse, from a written analysis with proofs to a mix-in that changes only geometry — which is the clearest evidence that the abstraction is at the right level.

Table 11. Worked applications shipped with `torchtsetlin`. Timings are for the default arguments on an RTX 3090.

Example	Model	What it shows
`noisy_xor.py`	`TsetlinMachine`	the benchmark of the original paper; recovers the generating rule under 40% label noise
`tabular_breast_cancer.py`	`TsetlinMachine`	thermometer encoding of clinical measurements, rules printed in the original units
`regression_sine.py`	`RegressionTsetlinMachine`	a Tsetlin machine that emits a number
`mnist_flat.py`	`TsetlinMachine`	784 pixels as flat features, checkpoints, classification report
`shapes_conv.py`	`ConvTsetlinMachine`	1.000 test accuracy on every seed tried against 0.55–0.91 for a flat machine with the same clause budget — and what `position_encoding` costs on a translation-invariant task
`conv1d_ramps.py`	`Conv1dTsetlinMachine`	0.996 on noisy 1-D signals; the clauses print as the ramp they detect
`shapes_conv3d.py`	`Conv3dTsetlinMachine`	shapes that differ only along depth: 1.000 on every seed tried, against 0.83–0.98 for slices stacked as channels and 0.73–0.80 for a flat machine; clauses print slice by slice
`conv_regression_blobs.py`	`ConvRegressionTsetlinMachine`	reading a continuous quantity (a square's side length) off an image: MAE 0.20, R^2 0.96
`mnist_conv.py`	`ConvTsetlinMachine`, `ConvCoalescedTsetlinMachine`	0.979 with 200 clauses per class; 0.985 with `--coalesced --clauses 1000 --T 500`, in less time
`segmentation_scenes.py`	`SegmentationTsetlinMachine`	generated street scenes: 0.991 pixel accuracy and 0.946 mean IoU with position literals, 0.959 adding a pyramid; prints the row-prior floor and explains single pixels
notebooks		
Iris	`TsetlinMachine`	a rule set small enough to read in full
MNIST, CIFAR-10	convolutional	clause patches, colour Booleanization, GPU throughput
Convolutional regression	`ConvRegressionTsetlinMachine`	working through the C-RTM segmentation literature [18]
CTM-UNet	`TsetlinMachine`	a dense per-pixel Tsetlin segmentation head, after [19]

10.2 Where a Tsetlin machine earns its place

Clinical decision support. This is the application domain the Tsetlin machine literature has pushed hardest, and for a reason: a clinician can audit a conjunction and a regulator can read it. Published work covers interpretable rule learning from medical text [22], premature ventricular contraction identification in ECG [21], organism prediction in peritoneal dialysis, bladder-cancer recurrence rule discovery and cervical-cancer classification with convolutional machines. What these share is a data regime — a few thousand examples, features that are already measurements with meaning — in which a thermometer-encoded Tsetlin machine is both competitive and legible, and a deep network is neither reliably better nor auditable.

`torchtsetlin` contributes three things here: the encoders that turn clinical measurements into literals with names attached, so a rule comes out as `IF worst perimeter>=105.9 AND NOT mean`

texture>=21.8; per-example explanations that are the model rather than a surrogate; and the ability to run the model inside the same PyTorch pipeline as the deep baseline it must be compared against.

Medical imaging. Convolutional and coalesced machines extend the same argument to images, and segmentation work [18,19] shows the family reaching per-pixel tasks. This is the setting that motivated torchtsetlin: in a cardiac-MRI research group the surrounding stack is PyTorch, the volumes are large, and an interpretable model that cannot consume a DataLoader or run on the GPU next to the U-Net baseline is not a practical option. Making a convolutional Tsetlin machine an nn.Module is what makes that comparison a one-file experiment rather than a project. The library now covers the shapes of that data directly: Conv3dTsetlinMachine reads a stack of short-axis slices — or a cine sequence with time as depth — as one volume, and SegmentationTsetlinMachine produces a label map whose every pixel can be explained in terms of named input pixels. A cardiac-sized $12 \times 64 \times 64$ volume trains at several hundred volumes per second on one GPU (Section 8.8).

Edge and hardware inference. Inference is an AND over included literals and a vote count, so a trained machine maps naturally onto digital logic. There is now a substantial literature on that mapping — asynchronous logic for edge inference [15], in-memory computing on Y-Flash cells [16], and 65 nm keyword-spotting accelerators [17]. torchtsetlin is a *training* framework, not a deployment target, but the learned state it exports is exactly the artefact those pipelines consume: an integer matrix and a set of weights, in a state_dict, with a documented layout.

Signals and time series. Conv1dTsetlinMachine applies the same patch mechanism to sequences, where a clause becomes a motif detector that fires wherever the motif occurs. Combined with thermometer encoding of the signal amplitude, the learned clauses are legible as pictures of the waveform they detect — which is how the shipped conv1d_ramps.py example prints them.

Tabular and scientific data. The oldest and least glamorous case, and still the strongest: datasets with tens to hundreds of meaningful columns, where the question is often *which combinations of measurements matter* rather than what the accuracy is. Global feature importance aggregated back over thermometer bits answers that question in closed form, with no permutation testing and no surrogate.

In short. The applications that suit a Tsetlin machine are the ones where somebody has to read the model. torchtsetlin's contribution to them is prosaic and decisive: the model now lives in the same framework as everything else the project already uses.

11 Limitations and future development

11.1 Known limitations

Batched feedback is an approximation. The default path trades fidelity for throughput, and the trade is real: at batch 200 on Noisy XOR the degradation is unambiguous (Section 8.10). The exact algorithm is always one argument away, and the characterisation here is empirical — there is no proof that the batched update converges to the same fixed point as the sequential one, and establishing the conditions under which it does is open work.

Dense tensors, not bit-packed kernels. Literals are floats in a matmul, not bits in a machine word. A hand-written kernel can pack 64 literals per word and exploit clause sparsity through indexing [12]; `torchtsetlin` does neither. The consequence is measurable and one-directional: on a *small* machine on a CPU, the reference C implementations are faster, and `torchtsetlin`'s advantage only appears with model size and a GPU. The cached include mask also costs $2CF$ floats of memory that a packed representation would not need.

The commit is dominated by one operation. `torch.binomial` over the $(C, 2F)$ state is 43% of the CPU update and the single largest GPU kernel in the commit. It is a generic sampler being asked to do a very specific job; a fused kernel that samples and updates in one pass, or an approximation valid in the large-count regime, would move the whole library.

Variants not implemented. The Tsetlin machine literature has grown faster than any one package can follow. Absent from `torchtsetlin` today: graph Tsetlin machines [14], TM composites [20], contracting machines with absorbing automata [24], the sparse machine with active literals [25], probabilistic and uncertainty-quantifying variants [26], and autoencoders. Each of these is expressible in the current architecture — most are an output layer and a feedback policy — but none is written.

No comparison against other implementations. Section 8 compares this library on a CPU against itself on a GPU. It does not establish how `torchtsetlin` compares against `tmu` or the CUDA packages at matched configurations, on either speed or accuracy. That comparison is worth doing carefully, with matched Booleanization and matched hyper-parameters, and it has not been done here.

Interpretability coverage is uneven. Rule extraction, clause activity, and global importance work for every model. Local feature importance is weighted by literal values only for flat models; for convolutional models the per-example attribution is computed over clause weights and inclusion without the per-patch literal gating, which makes it coarser than its flat counterpart.

Dense and volumetric models multiply the work. A segmentation model evaluates every clause at every pixel, and a 3D model at every window position of a volume, so both cost one to two orders of magnitude more per input than a classifier. Dense training is also bounded by the commit size: the default of 128 pixels keeps fidelity and leaves most of the GPU's speed unused (Section 8.8). Blocking the patch axis keeps a single large volume inside the memory budget, but it saved only 18% of peak memory in the measured case; for big volumes the stride is the effective lever. There is no 3D segmentation model yet, and its fold $B' = B \cdot P$ would make the commit-size problem worse.

Single-device, single-process. There is no multi-GPU or distributed training path today.

Early-stage API. Version 0.3.x is classified alpha. The core — `update`, `forward`, the buffer layout — is stable in intent, but the surface may still move, and checkpoints are tied to the state layout rather than a versioned format. Behaviour has changed between minor versions when a measurement demanded it — the clause budget became a hard cap in 0.3.0 (Section 6.5) — and the changelog says so each time.

11.2 Where it goes next

A fused feedback kernel. The measurement points at exactly one target. A Triton or CUDA kernel that draws the binomial counts and applies the clamped update in a single pass over the state would cut the largest term in both the CPU and GPU profiles, and it fits the existing architecture as a drop-in replacement for one function in `functional.py`.

Data-parallel training, almost for free. This one falls out of the design. The feedback accumulator holds *additive* event counts: combining the counts from two chunks is addition, and nothing about that changes if the chunks were computed on different devices. Data-parallel training over n GPUs is therefore an all-reduce of four count tensors before a single commit — mathematically identical to a single-device update at n times the batch size, with no approximation introduced beyond the one batching already makes. The same property makes gradient-free federated training natural: what is exchanged is a count matrix, not a model.

A bit-packed inference path. Training benefits from dense float matmuls; inference does not need them. A packed representation for `predict` — and an export path to the integer artefacts that the hardware literature [15,16,17] consumes — would close the loop between this framework and deployment.

More of the literature. In rough order of how well each fits the current architecture: clause-size and sparsity-oriented variants (a feedback-policy change), TM composites (a wrapper over several trained machines), absorbing automata (a state-transition change in `apply_feedback`), and graph machines (a new input structure, in the same place convolution lives).

Volumes, further. A dense 3D head (per-voxel segmentation) is the natural next mix-in, and the same proofs carry over. Anisotropic MRI spacing is already expressible through the depth window and stride, and a cyclic position code for cardiac phase would be a change to the position literals only.

Validation against the reference implementations. A test suite that checks `torchtsetlin`'s sequential path against the reference C implementation example by example, on fixed seeds, would turn "faithful to the published algorithm" from a claim into a measurement. The batched path could then be characterised as a controlled deviation from that baseline rather than from the literature's reported numbers.

In short. The honest summary: `torchtsetlin` wins on integration and on large-model GPU throughput, loses to hand-written kernels on small CPU workloads, approximates the classical algorithm by default, and implements a subset of a fast-moving literature. Every one of those is a stated, measured trade rather than a surprise waiting for a user.

12 Availability

Version	0.3.2 (19 October 2026)
Source	https://github.com/vlbthambawita/torchtsetlin
Package	https://pypi.org/project/torchtsetlin/ — `pip install torchtsetlin`
Documentation	https://vajira.info/torchtsetlin/
Literature catalog	*Benchmarks* → *Literature results* in the documentation; https://huggingface.co/spaces/vlbthambawita/TsetlinMachines
Licence	MIT
Python	3.9, 3.10, 3.11, 3.12
Required	`torch >= 2.0`, `numpy >= 1.22` — no compiler, no CUDA toolkit, no C extension
Optional extras	`[viz]` matplotlib | `[vision]` torchvision | `[sklearn]` scikit-learn | `[docs]` MkDocs Material | `[dev]` the lot, plus pytest and ruff

12.1 Getting started

```
pip install torch          # pick the CPU or CUDA build from pytorch.org
pip install torchtsetlin   # + optional extras: [viz] [vision] [sklearn] [all]
```

From source, with the development environment:

```
git clone https://github.com/vlbthambawita/torchtsetlin && cd torchtsetlin
pip install -e ".[dev]"
pytest -q                  # the full suite
ruff check src tests       # exactly what CI runs
mkdocs serve               # the documentation site at localhost:8000
```

12.2 Reproducing the measurements

Every number in Section 8 comes from the driver in `benchmarks/`. It runs one suite at a time, merging its JSON after each, so a crash costs only the running suite, and then regenerates the figures, tables and the documentation page from those records.

```
./benchmarks/run_benchmarks.sh                # every suite, cpu + cuda:0 (~20-25 min)
./benchmarks/run_benchmarks.sh --quick        # ~1 min smoke test of the pipeline
./benchmarks/run_benchmarks.sh -s batch models # a subset
./benchmarks/run_benchmarks.sh --plots-only   # re-render from results already on disk
```

The raw records land in `benchmarks/results/cpu_vs_gpu.json`. Replacing them with your own hardware's is the whole workflow — the derived page is regenerated, not hand-edited.

12.3 Quality gates

Continuous integration runs `ruff` and the full test suite on Python 3.9, 3.11 and 3.12, then deploys the documentation on pushes to `main`. Releases are cut from a `v*` tag through PyPI Trusted Publishing; the release workflow re-runs the suite and smoke-imports the built wheel in a clean virtual environment before uploading — a gate added after a `.gitignore` rule silently excluded a subpackage from the 0.1.0 wheel and made `import torchtsetlin` fail for everyone who installed it.

12.4 Documentation

The documentation site is MkDocs Material with `mkdocstrings` reading Google-style docstrings directly from the source, so the public API reference cannot drift from the code. It covers installation and a quick start; concept pages on the algorithm, Booleanization, batched versus sequential feedback, convolution (including 3D volumes), segmentation and hyper-parameters, with animated explanatory figures; guides on training, evaluation, interpretability, visualisation, saving, regression and multi-label, segmentation and extending; worked examples; and the generated CPU-versus-GPU benchmark page.

12.5 Citation

The repository's `CITATION.cff` drives GitHub's *Cite this repository* button. The five-page summary report (`https://vajira.info/torchtsetlin/cite/`) is the document to cite:

```
@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}
}
```

Please also cite the Tsetlin machine papers whose algorithms you rely on — the reference list that follows is a reasonable starting point, and the concept pages in the documentation name the specific paper behind each option.

12.6 Contributing

Issues and pull requests are welcome at the repository. The most useful contributions, in order: a model variant from the literature implemented against the existing hooks (Section 5); a fused feedback kernel; validation against the reference implementations; and applications — a worked notebook on a real dataset is worth more to the next user than another synthetic benchmark.

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

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

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

[7] K. D. Abeyrathna, O.-C. Granmo, X. Zhang, L. Jiao, M. Goodwin. *The Regression Tsetlin Machine: A Novel Approach to Interpretable Non-linear Regression*. Philosophical Transactions of the Royal Society A,

378(2164), 2020.

[8] J. Sharma, R. Yadav, O.-C. Granmo, L. Jiao. *Drop Clause: Enhancing Performance, Robustness and Pattern Recognition Capabilities of the Tsetlin Machine*. AAAI 2023; arXiv:2105.14506.

[9] O.-C. Granmo. *An Introduction to Tsetlin Machines*. https://tsetlinmachine.org.

[10] M. L. Tsetlin. *On the Behaviour of Finite Automata in Random Media*. Automation and Remote Control, 22:1210–1219, 1961.

[11] K. D. Abeyrathna et al. *Massively Parallel and Asynchronous Tsetlin Machine Architecture*. ICML 2021; arXiv:2009.04861.

[12] S. R. Gorji et al. *Increasing the Inference and Learning Speed of Tsetlin Machines with Clause Indexing*. arXiv:2004.03188, 2020.

[13] Halenka et al. *Exploring the Effects of Hyperdimensional Vectors for Tsetlin Machines*. arXiv:2406.02648, 2024.

[14] O.-C. Granmo et al. *The Tsetlin Machine Goes Deep: Logical Learning and Reasoning with Graphs*. arXiv:2507.14874, 2025.

[15] A. Wheeldon, R. Shafik, A. Yakovlev et al. *Low-Latency Asynchronous Logic Design for Inference at the Edge*. arXiv:2012.03402, 2020.

[16] *IMPACT: In-Memory Computing Architecture Based on Y-Flash Technology for Coalesced Tsetlin Machine Inference*. arXiv:2412.05327, 2024.

[17] *TsetlinKWS: 65 nm State-Driven Convolutional Tsetlin Machine Accelerator for Keyword Spotting*. arXiv:2510.24282, 2025.

[18] K. D. Abeyrathna, O.-C. Granmo, M. Goodwin. *Convolutional Regression Tsetlin Machine: An Interpretable Approach to Convolutional Regression*. ICMLT 2021. doi:10.1145/3468891.3468901

[19] *CTM-UNet: Integrating Convolutional Tsetlin Machines with U-Net Architecture for Semantic Segmentation*. ISTM 2025.

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

[21] *Interpretable Tsetlin Machine-based Premature Ventricular Contraction Identification from ECG*. arXiv:2301.10181, 2023.

[22] G. T. Berge, O.-C. Granmo, T. Tveit, M. Goodwin, L. Jiao, B. Matheussen. *Using the Tsetlin Machine to Learn Human-Interpretable Rules for High-Accuracy Text Categorization with Medical Applications*. arXiv:1809.04547, 2018.

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

[24] *Contracting Tsetlin Machine with Absorbing Automata*. arXiv:2310.11481, 2023.

[25] *The Sparse Tsetlin Machine: Sparse Representation with Active Literals*. arXiv:2405.02375, 2024.

[26] *The Probabilistic Tsetlin Machine: A Novel Approach to Uncertainty Quantification*. arXiv:2410.17851, 2024.

arXiv identifiers and DOIs are given so that every entry can be resolved directly. Where an author list is abbreviated or omitted, the title and identifier are authoritative. A BibTeX version of this list ships alongside the source of this document as `references.bib`.