Caching and Batching for Neural Clause Selection in Saturation-Based Theorem Proving

Alexander Pluska
TU Wien, Austria

Florian Zuleger
TU Wien, Austria

Abstract

A neural clause selector sits in the hot path of a saturation-based theorem prover: the model is invoked on every generated clause and the prover blocks on each result. Whether the ML guidance pays off under a fixed wall budget depends on two axes, the model's selection quality and the per-call cost the integration imposes. This paper holds the first axis fixed — a GCN encoder paired with a transformer scorer — and isolates two engineering decisions on the second: whether clause embeddings are cached across iterations, and whether embed requests are fired one at a time (eager) or buffered to the selection boundary (batched). On 16 TPTP problems run to a 256-iteration cap, caching changes per-problem wall time by at least a factor of $\approx 28\times$, and without it the prover routinely exhausts the 60-second wall budget; the batching decision is smaller and workload-dependent (eager wins by 1–40 % on this set, where $|\Delta| \leq 20$ new clauses are generated per iteration; on heavier workloads the ordering reverses). The two decisions are independent and orthogonal to the model, so the same trade-off recurs in any encoder/scorer-shaped clause selector.

2012 ACM Subject Classification Theory of computation → Automated reasoning; Computing methodologies → Neural networks

Keywords and phrases automated theorem proving, saturation, clause selection, neural networks, asynchronous inference

Digital Object Identifier 10.4230/LIPIcs...

Supplementary Material *Software*: `https://github.com/lammdachs/proofatlas`

Funding The authors acknowledge the support of the project VASSAL: "Verification and Analysis for Safety and Security of Applications in Life" funded by the European Union under Horizon Europe WIDERA Coordination and Support Action/Grant Agreement No. 101160022.

1 Introduction

Machine learning guidance for saturation-based theorem provers has progressed from hand-crafted clause features [22, 8] to graph neural network encoders over clause graphs [15, 5]. Across these systems the learned component sits in the prover's hot path: on each newly generated clause the model is invoked, and the prover cannot make its next selection decision until the result is in. For calculi like superposition, which can generate tens of thousands of clauses per proof attempt, the model is invoked tens of thousands of times before a proof is found.

This puts the ML system between two competing forces. The model is chosen for its *selection quality*: a smarter scorer picks clauses that lead to proofs faster. But every invocation costs wall-clock time, and the prover runs against a fixed wall budget; the more time the model takes per call, the fewer iterations the prover gets to take in the same budget. The metric that matters, proof rate within a fixed time, depends on both axes.

The model's design fixes one axis. The other, the per-call cost, depends on how the model is *invoked* from the prover loop. This paper isolates two engineering decisions that

we have found to sit on that second axis. Both are about scheduling around the model, not about the model itself; both are independent of which encoder or scorer architecture is used.

Caching of clause embeddings.

A clause's embedding depends only on the clause, not on the rest of the prover state (Section 2). When a clause enters the unprocessed set, its embedding can be computed once and cached for all subsequent selection steps in which the clause participates. The alternative is to re-embed every clause in both the unprocessed and processed sets at every selection step; this multiplies the backend workload per iteration by a large factor and is, in our measurements, the larger of the two levers.

Batching at the selection boundary.

Within one iteration, the prover's forward and backward simplification loops generate a set Δ of new clauses; each surviving clause needs an embedding before the next selection step. There is a real choice about *when* the model is invoked to compute those embeddings: either as each new clause arrives, paying any fixed per-call overhead $|\Delta|$ times, or once per iteration over all of Δ together, paying it once. The first pattern lets the model run in parallel with the prover's continued simplification; the second amortizes a single invocation over more work. Which of these dominates depends on how many clauses an iteration generates and on the relative cost of an invocation.

Contributions.

- We isolate the caching decision and the batching decision and measure their independent effects on a single configuration (a GCN encoder paired with a transformer scorer, referred to throughout as *encode_score*) on 16 TPTP problems that the prover runs to its 256-iteration limit under every cell. We run three repetitions per cell.
- We show that the caching decision changes per-problem wall time by at least a factor of $\approx 28\times$ (a lower bound: roughly half of the no-cache runs hit the 60-second wall budget before completing the iteration budget). Without caching the inference cost alone is enough to push these problems past the wall.
- We show that the batching decision is workload-dependent: on the studied iter-cap problems ($|\Delta| \leq 20$ new clauses per iteration) eager submission wins by 1–40 % over batched submission, because the embed work overlaps with the prover's continued simplification; on heavier workloads (e.g. LCL084-10 with $|\Delta| \approx 80$) batched wins instead.
- We place these engineering decisions in their proof-rate context. On a broader 11,091-problem TPTP slice, the cached encode_score configuration proves 27.8 % of problems versus 26.9 % for the non-learned age–weight heuristic. The ML selection quality buys roughly one percentage point of proof rate at roughly $10\times$ the per-problem wall cost of the heuristic, and that comparison only exists because the caching decision keeps the per-problem wall cost from blowing past the time budget.
- We describe the implementation of both decisions in our prover ProofAtlas[1] and discuss the implications for other ML-guided saturation systems.

[1] Source at `https://github.com/lammdachs/proofatlas`.

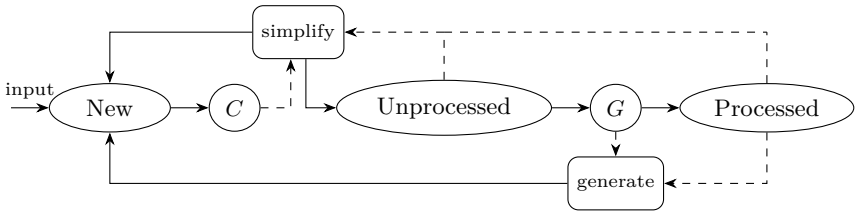

Figure 1 The given-clause loop. Dashed arrows show inputs to inference steps; solid arrows show clause flow.

2 Background

2.1 Saturation-Based Theorem Proving

We work in first-order logic with equality. Given a set of clauses S, the goal is to derive the empty clause $\square$, proving unsatisfiability. To show that a conjecture follows from axioms, we negate the conjecture and prove the combined clause set unsatisfiable.

▶ **Example 1.** As a running example, we use TPTP problem GRP001-1 [21]: if the square of every element in a group is the identity, the group is commutative. The 11 input clauses include unit clauses such as $\mathit{product}(\mathit{identity}, X, X)$ (left identity), $\mathit{product}(X, Y, \mathit{multiply}(X, Y))$ (defining *multiply* as a Skolem function for *product*), and $\mathit{product}(X, X, \mathit{identity})$ (hypothesis), as well as multi-literal clauses such as

$$\neg \mathit{product}(X, Y, Z) \vee \neg \mathit{product}(X, Y, W) \vee Z \approx W$$

encoding well-definedness of *product*.

The given-clause algorithm.

Saturation-based provers iteratively enumerate consequences of the initial clause set, applying *generating inferences* to derive new clauses and *simplifying inferences* to delete or rewrite existing ones. The search terminates with *proof found* when the empty clause is derived, or *saturated* when no further new clauses can be generated. Implementations vary in how they organise the workspace; the version we describe here, and use in ProofAtlas, maintains three clause sets — *new*, *unprocessed*, and *processed* [24] — and proceeds (Figure 1):

1. While there are clauses in new:
   a. Pick a clause C; if $C = \square$, return *proof found*.
   b. Apply forward simplification; if C is deleted, repeat.
   c. Apply backward simplification using C; move C to unprocessed.
2. If unprocessed is empty, return *saturated*. Otherwise, select a clause G (the *given clause*) and move it to processed.
3. Apply generating inferences between G and processed (results go to new).

Clause selection.

The choice of G in step 2—*clause selection*—is the main branching decision of the algorithm. The standard non-learned heuristic is *age–weight interleaving* [17]: selection alternates between the oldest clause (ensuring fairness) and the lightest clause (preferring simpler terms) according to a configurable ratio. Replacing this heuristic with a learned selector is the setting in which the present paper's contribution operates.

2.2 The Superposition Calculus

The superposition calculus [2] extends resolution with equality handling. Inferences are parametrized by a *term ordering* $\succ$ and a *literal selection strategy*: a function assigning to each clause a subset of its literals.

Generating inferences.

Underlined literals must be selected. $L[s]$ denotes literal L containing subterm s.

$$\frac{C \vee \underline{L} \qquad D \vee \underline{\neg L'}}{(C \vee D)\sigma}\ \textsc{Res} \qquad\qquad \frac{C \vee \underline{L} \vee L'}{(C \vee L)\sigma}\ \textsc{Fac}$$

where $\sigma = \mathrm{mgu}(L, L')$ is the most general unifier of L and L'.

$$\frac{\underline{l \approx r} \vee C \qquad \underline{L[l']} \vee D}{(L[r] \vee D \vee C)\sigma}\ \textsc{Sup}$$

where $\sigma = \mathrm{mgu}(l, l')$, l' is not a variable, and $l\sigma \not\preceq r\sigma$. If $L[l']$ is an equality literal $s[l'] \approx t$ or $s[l'] \not\approx t$, additionally $s[l']\sigma \not\preceq t\sigma$.

$$\frac{C \vee \underline{s \not\approx t}}{C\sigma}\ \textsc{EqRes} \qquad\qquad \frac{\underline{s \approx s'} \vee t \approx t' \vee C}{(s' \not\approx t' \vee s \approx s' \vee C)\sigma}\ \textsc{EqFac}$$

where $\sigma = \mathrm{mgu}(s, t)$, and in EqFac additionally $s\sigma \not\preceq s'\sigma$, $s\sigma \not\preceq t'\sigma$, $s'\sigma \not\preceq t'\sigma$.

Simplifying inferences.

Four simplifying inferences reduce the search space:

- **Tautology deletion**: Delete a clause that contains complementary literals or a reflexive equality $t \approx t$.
- **Demodulation**: Given a unit equality $l \approx r$ with $l \succ r$ and a substitution σ, replace all occurrences of $l\sigma$ in a clause by $r\sigma$.
- **Condensation**: Replace clause C by $C\sigma$ when there exists a substitution σ such that $C\sigma$ has strictly fewer literals than C and $C\sigma$ subsumes C.
- **Subsumption**: Delete clause D if there exists a clause C and substitution σ such that every literal of $C\sigma$ is a literal of D.

Term ordering.

We use the Knuth–Bendix ordering (KBO) [12] with alphabetic symbol precedence and constant 1 weight function.

Literal selection.

We use the Sel21 strategy of Hoder et al. [7]: select the unique maximal literal if it exists, else a maximal negative literal, else all maximal literals.

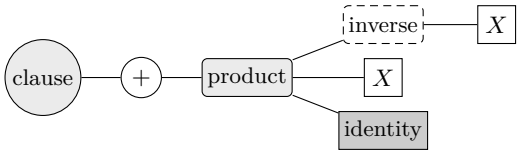

Figure 2 Clause graph for $product(inverse(X), X, identity)$. Each symbol occurrence becomes a separate node. Node types: clause root (filled circle), literal (circle, $+/-$), predicate (filled rounded), function (dashed rounded), variable (plain), constant (dark filled). Edges are bidirectional.

2.3 Graph Neural Networks

Graph neural networks (GNNs) learn representations of graph-structured data through iterated message passing: each layer updates node representations by aggregating information from neighbors, and a graph-level representation is obtained by pooling over the final node embeddings. A graph convolutional network (GCN) [11] updates all node representations simultaneously by averaging neighbor features with symmetric degree normalization, preventing high-degree nodes from dominating. We apply GraphNorm [3] after each layer, which normalizes within each graph independently using a learnable mean-subtraction parameter. A graph-level embedding is obtained by pooling over node representations. GNNs are widely used as clause encoders in saturation-based theorem proving [14, 8, 6, 5].

2.4 Transformers

Transformers [23] process sequences through layers of multi-head attention and feed-forward networks. In scaled dot-product attention, one sequence produces queries while another produces keys and values; each query computes a dot-product similarity with all keys and uses the resulting weights to aggregate values. When queries and keys come from the same sequence, this is *self-attention*; when from different sequences, *cross-attention*. A transformer layer combines multi-head attention with a position-wise feed-forward network, connected by residual connections and pre-norm LayerNorm [25].

2.5 Learned Clause Selection

A learned selector replaces age–weight scoring with a neural model: each unprocessed clause is encoded into a fixed-dimensional embedding, a scoring head produces a real-valued score per clause, and the given clause is sampled with temperature-softmax over scores. Encoder and scorer together form an *encoder–scorer* architecture following TRAIL [5]. We refer to a specific encoder–scorer pair as a *configuration*. The question of which configuration to use is largely orthogonal to our contribution; we fix a single configuration — a GCN clause encoder paired with a transformer scorer (cross-attention from U to P followed by per-token feed-forward sublayers), which we denote *encode_score* — for all experiments in this paper. The architectural choice itself, and how the model is trained, are not the subject of this paper; the questions we study are about how a configuration of this shape should be *invoked* from inside the prover loop.

Clause graphs.

The GCN encoder represents each clause as a tree-structured graph (Figure 2) with nodes for the clause root, literals, predicates, functions, variables, and constants, connected by bidirectional parent–child edges following the term structure. Each occurrence of a symbol

or variable becomes a separate node. Each node carries only *structural* features—node type (one-hot), arity, and argument position (sinusoidally encoded). Symbol identity is not used as an input, matching the symbol-independent design of prior GNN-based clause encoders [14, 8]. Importantly, separate clauses constitute separate graphs: there are no edges between clauses, so the encoder treats each clause as an independent input. This makes the per-clause embedding a function of the clause alone, which is what permits the caching strategy of Section 3.

2.6 Deployment

In ProofAtlas, neural models run on a single *backend worker* thread with exclusive access to compute resources (CPU threads for small models, one GPU for larger ones). A prover worker submits inference requests to the backend via a channel. The backend's receive loop drains any concurrently arriving requests (via non-blocking `try_recv`) and dispatches them as a batch to the model.

3 Two Implementation Decisions

Integrating a learned scorer into the saturation loop forces two implementation decisions. We name them here and measure their effects in Section 5. The two decisions are independent: any combination of choices is a valid implementation.

Decision 1: caching of clause embeddings.

A clause's embedding is a function of the clause alone — the encoder consumes the clause graph in isolation, and separate clauses constitute separate graphs (Section 2). Once computed at transfer time, the embedding does not need to be re-computed at subsequent selection steps in which the same clause participates. The implementation choice is whether to *cache* the embedding or to recompute it: caching pays the forward-pass cost once per clause and reads it back at every selection step, while recomputing re-embeds every clause in $U \cup P$ at each selection step — $O(|U| + |P|)$ embed calls per iteration rather than $O(|\Delta|)$. Both implementations produce the same scores; they differ only in where the work is done.

Decision 2: dispatch timing of embed requests.

Within one iteration the prover's forward and backward simplification loops generate $|\Delta|$ new clauses, each needing an embedding before the next selection step. The implementation choice is whether to dispatch each embed request as the corresponding clause is generated (*eager*), or to buffer the $|\Delta|$ requests at the processor and fire them as a single burst at the selection boundary (*batched*). The two patterns trade off in opposite directions: eager submission lets the embed work proceed in parallel with the prover's continued simplification of subsequent clauses, while batched submission amortizes any fixed per-call overhead (kernel launch, thread wake-up) over the whole iteration's worth of embeds. Both implementations produce the same scores; they differ only in dispatch schedule.

Summary

Each of the four combinations (cache vs. no cache, eager vs. batched) is a valid implementation that produces the same proof search modulo floating-point reproducibility. Section 5 measures the per-problem wall-time cost of each combination on a 16-problem benchmark.

The caching decision turns out to dominate the table by an order of magnitude; the dispatch-timing decision is smaller and points in different directions depending on the workload.

4 Implementation

We describe how the caching and batching decisions of Section 3 are realized in ProofAtlas. Both are local changes to the clause-selection processor that sits between the prover's on_transfer/select callbacks and the inference backend; neither changes the prover, the clause manager, the saturation calculus, or the model itself.

4.1 Embedding cache

The processor maintains two index maps, $U_{\text{emb}} : idx \to \mathbb{R}^d$ and $P_{\text{emb}} : idx \to \mathbb{R}^d$, of clause embeddings for the unprocessed and processed sets respectively. The prover's clause-management callbacks update these maps:

- on_transfer(c): record an embed request for c's index, populated at the next drain point (see below) into U_{emb}.
- on_activate(c): move c's embedding from U_{emb} to P_{emb}.
- on_simplify(c): remove c's embedding from whichever map holds it. If c was simplified before its embed had fired, drop the pending request entirely.

When select needs scores, it reads the current U_{emb} and P_{emb} (after firing any pending embed requests and draining their responses) and submits a single score_context call with the two embedding sets.

The cache is a pure performance optimization: a clause's embedding depends only on the clause so the cached value is identical to what re-computation would produce. Disabling the cache (ablated in Section 5) means U_{emb} and P_{emb} are recomputed from scratch at every select, by embedding every clause currently in U and P.

4.2 Batched submit at the selection boundary

The processor maintains a small local buffer $to\_submit : \mathsf{Vec}\langle(idx, \mathsf{PrebuiltInput})\rangle$ of embed requests not yet sent to the backend. Each PrebuiltInput is the per-clause graph and feature vector produced by translating the clause into the backend's input representation; this translation is done eagerly on the processor thread in on_transfer, so that the CPU graph-building work overlaps with the prover's continued simplification loop rather than running on the backend thread. The buffer therefore carries already-tensorized inputs, not the clause itself. The buffering policy is parameterized by a single integer B, the *embed batch size*:

- In on_transfer, the clause is translated into a PrebuiltInput and the pair is appended to to_submit. If $|to\_submit| \geq B$, the buffer is flushed (fire_buffer): each entry is sent to the backend via submit_async in rapid succession.
- At the next select call the buffer is flushed regardless of whether B has been reached, the backend responses are awaited, and the resulting embeddings are written into U_{emb} before scoring proceeds.

$B = 1$ is the eager policy: each embed request fires its own backend round trip. B larger than the typical $|\Delta|$ is the batched policy: the buffer fills only at drain time, and the backend receives one rapid burst of $|\Delta|$ submit_async calls per iteration. Because the backend's receive loop drains non-blocking after each recv, this burst is observed as a single batch of size $|\Delta|$ and dispatched as one forward pass.

We set B to a value much larger than any plausible $|\Delta|$ (specifically 10^6, well above the largest per-iteration $|\Delta|$ we have ever observed) when the batched policy is selected. The buffer cap is conceptually a function of GPU memory available to the embed model; the same memory budget that already accommodates the score_context call's $|U| + |P|$ inputs comfortably accommodates any realistic embed batch.

4.3 What does not change

The cache and the batching policy are orthogonal: any combination of {cache, no cache} × {eager, batched} is valid. The 2×2 grid is the experimental design of Section 5. Neither lever changes the model, the weights, the prover, or the calculus. The set of clauses passed to score_context at each selection step is identical across all four cells.

5 Evaluation

5.1 Setup

All experiments run ProofAtlas with one prover worker on a Linux host with an NVIDIA RTX A5000 (24 GB). Problems come from TPTP version 9.0.0. Per-problem prover limits are 60 s of wall time, 256 given-clause iterations, and 64 MB of clause storage. These three limits are chosen so that on the problems of interest the iteration cap is the binding constraint: the wall and memory budgets are slack on the cached configurations and the prover terminates at iteration 256 with status resource_limit, so per-problem wall time becomes a direct measurement of inference throughput rather than of whether a proof was found. The term ordering is KBO with constant-1 weight (Section 2) and the literal selection strategy is Sel21 [7]. We use the *encode_score* configuration (GCN encoder + transformer scorer, Appendix A); the weights are unchanged across cells.

The problem set consists of 16 problems (listed in Appendix B) for which the prover hits the 256-iteration limit under every cell of the ablation. The problems span 8 TPTP domains; mean $|\Delta|$/iteration on this set ranges from 3.7 to 19.9 clauses (median 8.2).

The four cells of the 2×2 ablation are $\{\text{cache=T}, \text{cache=F}\} \times \{\text{batched}, \text{eager}\}$; batched uses embed batch size $B = 10^6$ (i.e., flush only at drain), eager uses $B = 1$ (each on_transfer submits immediately). For reference we additionally run cache=T with *sequential* submit, in which on_transfer drains its own response inline before returning, and the non-learned age–weight heuristic, which performs no inference at all. Each cell is run 3 times per problem; reported numbers are per-problem medians over the 3 reps.

5.2 Caching

Table 1 reports the median per-problem wall time in seconds. The cache vs. no-cache axis dominates the table.

The mechanism is direct: without the cache, every clause currently in $U \cup P$ is re-embedded at every select call, and $|U|$ grows into the hundreds over the course of the search. The cumulative re-embedding cost dominates the per-problem wall time and routinely exceeds the wall budget — 24 of 48 batched and 25 of 48 eager runs in the cache=F row hit the 60-second wall before completing the 256-iteration budget. The factor $\approx 28\times$ is consistent across the batched and eager columns, confirming that the caching axis is independent of the dispatch-timing axis.

Table 1 Per-problem wall time (median across 16 problems, 3 reps each). About half of the no-cache runs hit the 60-second wall budget before completing the 256-iteration budget, so the cache=F figures are truncated by the timeout and the 20× ratio is a lower bound.

	batched	eager	sequential
cache=T	2.37 s	2.07 s	3.07 s
cache=F	58.94 s†	59.79 s†	—
Heuristic baseline (no inference): age–weight			0.22 s

† truncated at the 60 s wall budget on roughly half the runs.

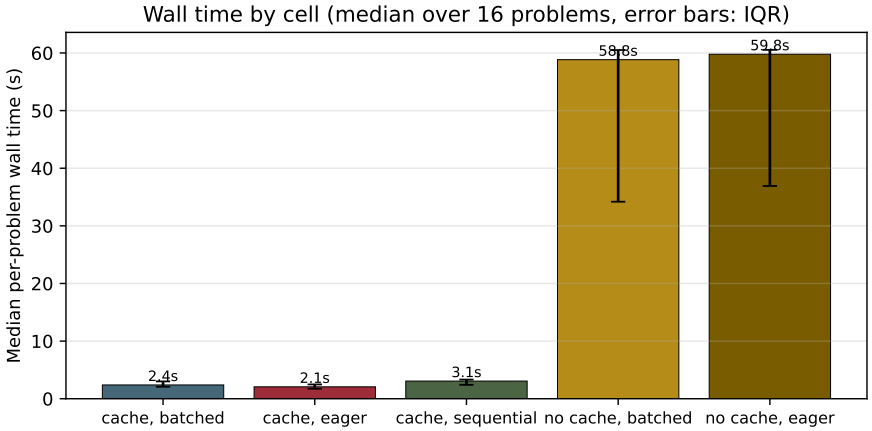

Figure 3 Median per-problem wall time across the 16 problems for each cell of the 2×2 ablation plus the sequential-submit reference. Error bars are the inter-quartile range across problems. The two cache=F cells brush against the 60-second wall budget; the ratio between cache rows is a lower bound on the cost of the re-embedding workload.

Absolute scale and proof-rate context.

The non-learned age–weight heuristic processes the same 16 problems for 256 iterations each in a median of 0.22 s per problem (single-rep data on the same machine). The cached encode_score configuration is therefore ≈ 10× slower per problem on the wall-clock axis; the uncached configuration is roughly two orders of magnitude slower. The encode_score model does not win on wall time — its per-iteration cost is real — but its *selection decisions* are better, which is what the ML is there to provide. On an 11,091-problem TPTP slice (all CNF and FOF problems under 64 kB whose declared status is unsatisfiable or unknown) under the same 60-second, 256-iteration budget, age–weight proves 2984 problems (26.9 %) and the cached encode_score configuration proves 3088 problems (27.8 %) — about +1 percentage point of proof rate that the better selection decisions buy back over the wall-time overhead. Selection quality and per-iteration cost are the two axes that determine whether an ML-guided prover beats the heuristic; this paper holds the model (and therefore the selection quality) fixed and measures what the engineering decisions cost on the second axis. Without the cache, the per-problem overhead would multiply by another ≈ 20× (per Table 1) and most TPTP problems would exhaust the 60-second wall before the better selection decisions had a chance to matter; the modest proof-rate gain would be erased.

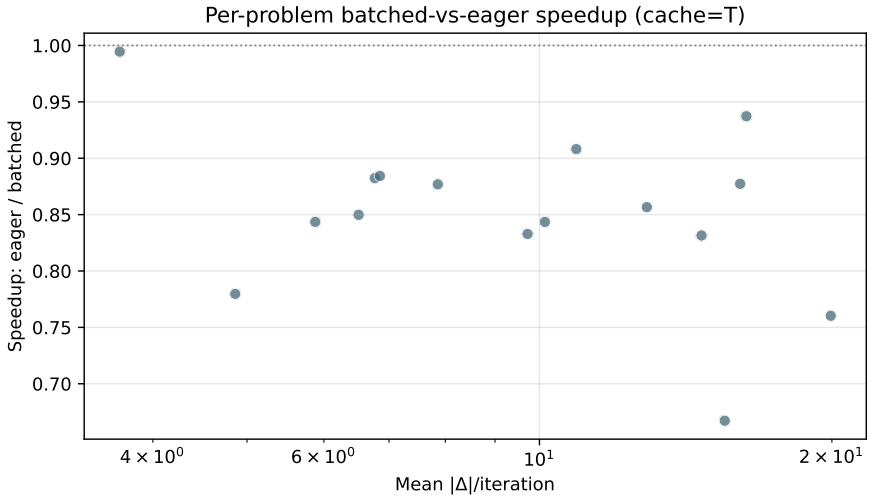

Figure 4 Per-problem eager/batched wall-time ratio as a function of mean $|\Delta|$/iteration across the 16 problems (cache=T). Values below 1.0 mean eager is faster; the dashed line marks parity. The cross-over to batched-faster lies beyond this set's $|\Delta|$ range.

5.3 Batching

Within the cache=T row of Table 1, eager (2.07 s) is faster than batched (2.37 s), and sequential is the slowest of the three (3.07 s). The reason: at the low $|\Delta|$/iteration values on this problem set (mean 3.7 to 19.9), the prover's between-call work in the simplify loop is large relative to the cost of one embed forward pass, so firing each embed eagerly lets the backend process it in parallel with that simplify work; by the time `select` fires, the responses are mostly ready and the drain is fast. The batched policy gives up this overlap, accumulating submits at the processor while the backend is idle. The amortization advantage of batched submission — one kernel launch per iteration instead of $|\Delta|$ — is real but small at these $|\Delta|$ values.

Figure 4 shows the per-problem eager/batched wall-time ratio against $|\Delta|$/iteration; every problem in this set falls below the parity line, so eager wins everywhere. The ratio trends toward parity as $|\Delta|$ grows, consistent with the prediction that batched submission eventually wins when the per-iteration embed workload is large enough for amortization to exceed the lost overlap. A separate CPU measurement on LCL084-10 (mean $|\Delta| \approx 80$) saw batched faster than eager by $\approx 16\%$, suggesting the cross-over on our setup is somewhere in $|\Delta|/\text{iter} \in [20, 80]$. The right dispatch policy for a given workload therefore depends on which side of that range its typical $|\Delta|$ sits.

5.4 Search nondeterminism

Even though every cell hits the 256-iteration limit in every run, the four cells process embed requests in batches of different sizes, and deep-learning kernels are not bitwise-stable across batch sizes. Score vectors therefore differ between cells at the level of low bits, which can flip individual selection decisions and lead to different searches. The 256-iteration constraint guarantees that the wall-time numbers across cells reflect throughput on equally long runs, but the specific clauses chosen at each step differ between cells. The wall-time comparisons in this section should be read accordingly.

6 Related Work

6.1 ML-guided saturation provers

Learned clause selection has been explored in E [16] via ENIGMA [9, 10, 8], in Vampire [13, 19, 20], and in TRAIL [5, 1], a saturation prover whose clause-selection policy is itself a neural model with an attention-based action head. These systems typically integrate the learned model as an in-process component invoked from the prover's main loop. The systems differ in their model architectures (gradient boosting in early ENIGMA variants, graph neural networks over clause graphs in later ENIGMA work, attention over clause embeddings in TRAIL) and in their training data generation strategies, but they share the same overall integration pattern: each new clause must be evaluated before the next selection decision can be made.

Among these systems, the caching and batching decisions we measure are typically not discussed explicitly. ENIGMA's clause-feature extraction is per-clause, so reusing the extracted feature vector across selection steps is in effect a memoization of the model's input; the ENIGMA papers do not, however, frame the presence or absence of such reuse as an explicit design decision. Whether the neural-encoder-based saturation selectors [5, 8, 6] cache embeddings or recompute them is, to our knowledge, also not reported, and likewise for whether they batch embed requests across the clauses generated in one iteration.

A recurring theme in this literature is the tension between model expressiveness and inference speed. Suda reports model-inference cost as a significant fraction of overall proof time in Vampire [20]; Jakubův et al. in ENIGMA Anonymous compare gradient-boosted decision trees and graph neural networks head-to-head as clause selectors, and a substantial part of that paper is the engineering required to bring the GNN-based selector's wall-clock cost into a range where its selection quality is competitive with the cheaper GBDT alternative in real-time proof counts and complementary to it on hard problems [8]. Our work is complementary to this line: rather than making the model cheaper, we characterize how the cost of invoking a fixed model depends on integration choices.

6.2 Clause encoders and selection heads

Recent neural clause-selection systems for saturation provers combine graph neural network encoders over clause graphs [5, 8] with various scoring heads, including the attention-based selection policy of TRAIL [5]. Graph encoders of logical formulas have also been studied outside the saturation-selection setting, for instance for tactic and premise prediction over HOL Light formulas in the HOList benchmark [15]; we reuse the general representation idea but our integration setting and target task are distinct. Our contribution does not propose a new encoder or new scorer: we pair a GCN encoder over clause graphs with a Transformer scorer operating on clause-level embeddings, with cross-attention from the unprocessed set to the processed set (Appendix A). The caching and batching decisions we measure apply identically to other encoder/scorer choices of similar shape.

6.3 Inference serving and batch scheduling

In the broader ML serving literature, request batching at a shared backend is standard practice [4, 18]; systems like Clipper and Nexus expose APIs for the very pattern we describe at the process level. Our contribution is to show that the same pattern matters in the tight-loop in-process setting of a saturation prover — where the "client" (prover) and the "server"

(inference backend) share an address space and the client issues many requests per iteration — and that getting it wrong has measurable proof-rate consequences.

6.4 Parallel saturation

Prior work on multi-strategy and parallel saturation (E's strategy scheduler [17], Vampire's portfolio modes [13]) explores the *search* axis — running multiple configurations or prover instances against the same problem. Our work is orthogonal: we accelerate a single prover worker's interaction with the inference backend. How the caching and batching decisions interact with cross-worker scheduling is left for future work.

7 Discussion

7.1 What this paper is, and is not, about

Neither the embedding cache nor the batched-submit policy is novel in any deep sense. Caching of a quantity that depends only on a stable input is how any sensible system handles memoization. Batching of inference requests at a known boundary is how any sensible system handles GPU-side overhead. Yet ML-guided saturation provers in the literature do not typically report which of these decisions they made or measure their impact, so a first-time implementer has no guidance and a re-implementer has no baseline. Our contribution is to isolate the caching and batching decisions, characterize their effects on a single configuration, and demonstrate that each one independently changes per-problem wall time by a factor that matters for the proof-rate metric the field actually cares about.

We have intentionally restricted the configuration under study to a single encoder–scorer pair (*encode_score* in the body and Appendix A). The point of the paper is not which encoder or scorer to use; the point is that for any encoder–scorer pair shaped like ours, the cache and the batching policy dominate the throughput question.

7.2 Limitations

One configuration.

We measure one encoder–scorer pair (GCN encoder + transformer scorer). A clause-intrinsic scorer that produces a scalar per clause without consulting a score_context over U and P would have a different shape and a different cost profile; in particular, the cache would memoize scalars rather than embeddings. Our experimental conclusions are likely qualitatively similar in that case but the per-problem savings should be smaller, because the lookup is cheaper.

Single prover worker.

All experiments use one prover thread. The natural follow-up is multi-worker scaling, where multiple prover threads share one inference backend. We have intentionally restricted ourselves to a single worker so that the throughput effects we report are attributable to the cache and the batching policy alone, not to scheduling at a shared backend. A proper multi-worker study — with controls for backend queue depth, batch dispatch policy, and floating-point determinism — is left for future work.

Search nondeterminism across cells.

Even when problems hit the iteration limit under every cell of our ablation, the batched and eager policies process embed requests in batches of different sizes, and deep-learning kernels are not bitwise-stable across batch sizes. Score vectors therefore differ at the level of low bits between cells, which can flip individual selection decisions and lead to different searches. We chose problems where this divergence does not flip the proof outcome (Section 5), but the wall-time numbers across cells should be understood as comparing different search trajectories, not bit-identical ones.

7.3 Future work

Selection quality versus inference time.

The deepest open question this paper opens is the trade-off the paper itself sidesteps by holding the model fixed: *how much selection quality is worth how much inference time?* A larger, more accurate scorer makes better choices per call but takes longer per call, so under a fixed wall budget it makes fewer choices. The question of which encoder–scorer pair maximises proof rate is therefore not "which is most accurate"; it is "which sits at the right point on a Pareto curve between per-call quality and per-call cost." Characterising that curve, on a fixed proving setup and across model sizes / architectures, is the natural next study and would put the engineering decisions in this paper into context: caching shifts the cost axis (the same model gets much cheaper per problem), batching shifts it more modestly, and together they enlarge the region of model-space that is *affordable* under a given wall budget.

Multi-worker scaling.

With multiple prover workers feeding one inference backend, the dispatch policy at the backend interacts with the within-worker batching policy in non-obvious ways. A controlled study would treat the within- and across-worker batching decisions as orthogonal axes; we expect cross-worker concurrency to push the crossover between eager and batched leftward (toward smaller $|\Delta|$) because the backend already sees more concurrent requests at any given moment.

7.4 Conclusion

A naive ML-guided saturation prover can pay unnecessary cost in two distinct ways: by re-embedding clauses whose embeddings have already been computed, and by dispatching embed requests one at a time when they could be batched together. Each of these is a separate engineering decision; each is independently measurable. In a system of our shape, the cache decision is decisive: caching saves at least a factor of $\approx 28\times$ on the wall-clock cost of running the prover's iteration budget, and without it the prover routinely exhausts the wall budget before completing that budget. The batching decision is smaller and workload-dependent: on problems with few clauses generated per iteration, firing embed requests eagerly is faster because the backend's work overlaps with the prover's simplification; on heavier problems, buffering the requests and firing them as a batch wins because the per-call dispatch overhead gets amortized. We hope that naming these decisions explicitly, and quantifying their impact on a controlled set of problems, helps both new implementers of ML-guided provers and authors of empirical comparisons between them.

AI Disclosure

Claude (Anthropic) was used extensively as a programming assistant in the implementation of ProofAtlas and for editorial assistance in drafting this paper. All technical contributions, experimental design, and scientific conclusions are the authors' own.

References

1. Ibrahim Abdelaziz, Maxwell Crouse, Bassem Makni, Vernon Austel, Cristina Cornelio, Shajith Ikbal, Pavan Kapanipathi, Ndivhuwo Makondo, Kartik Srinivas, Michael Witbrock, and Achille Fokoue. Learning to guide a saturation-based theorem prover. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2022.
2. Leo Bachmair and Harald Ganzinger. Resolution theorem proving. In *Handbook of Automated Reasoning*, pages 19–99. Elsevier, 2001.
3. Tianle Cai, Shengjie Luo, Keyulu Xu, Di He, Tie-yan Liu, and Liwei Wang. GraphNorm: A principled approach to accelerating graph neural network training. In *ICML*, pages 1204–1215, 2021.
4. Daniel Crankshaw, Xin Wang, Guilio Zhou, Michael J Franklin, Joseph E Gonzalez, and Ion Stoica. Clipper: A low-latency online prediction serving system. In *NSDI*, 2017.
5. Maxwell Crouse, Ibrahim Abdelaziz, Bassem Makni, Spencer Whitehead, Cristina Cornelio, Pavan Kapanipathi, Kartik Srinivas, Veronika Thost, Michael Witbrock, and Achille Fokoue. A deep reinforcement learning approach to first-order logic theorem proving. In *AAAI*, 2021.
6. Zarathustra A Goertzel, Karel Chvalovský, Jan Jakubův, Miroslav Olšák, and Josef Urban. Fast and slow enigmas and parental guidance. In *FroCoS*, pages 173–191, 2021.
7. Kryštof Hoder, Giles Reger, Martin Suda, and Andrei Voronkov. Selecting the selection. In *IJCAR*, pages 313–329, 2016.
8. Jan Jakubův, Karel Chvalovský, Miroslav Olšák, Bartosz Piotrowski, Martin Suda, and Josef Urban. ENIGMA anonymous: symbol-independent inference guiding machine (system description). In *IJCAR*, 2020.
9. Jan Jakubův and Josef Urban. ENIGMA: efficient learning-based inference guiding machine. In *CICM*, volume 10383 of *LNCS*, pages 292–302, 2017.
10. Jan Jakubův and Josef Urban. Hammering mizar by learning clause guidance (short paper). In *ITP*, 2019.
11. Thomas N Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. In *ICLR*, 2017.
12. Donald E Knuth and Peter B Bendix. Simple word problems in universal algebras. pages 263–297, 1970.
13. Laura Kovács and Andrei Voronkov. First-order theorem proving and Vampire. In *CAV*, pages 1–35, 2013.
14. Miroslav Olšák, Cezary Kaliszyk, and Josef Urban. Property invariant embedding for automated reasoning. In *ECAI*, pages 1395–1402, 2020.
15. Aditya Paliwal, Sarah Loos, Markus Rabe, Kshitij Bansal, and Christian Szegedy. Graph representations for higher-order logic and theorem proving. In *AAAI*, 2020.
16. Stephan Schulz. E—a brainiac theorem prover. In *AI Communications*, volume 15, pages 111–126, 2002.
17. Stephan Schulz. System description: E 1.8. In *LPAR*, pages 735–743, 2013.
18. Haichen Shen, Lequn Chen, Yuchen Jin, Liangyu Zhao, Bingyu Kong, Matthai Philipose, Arvind Krishnamurthy, and Ravi Sundaram. Nexus: a GPU cluster engine for accelerating DNN-based video analysis. In *SOSP*, 2019.
19. Martin Suda. Improving ENIGMA-style clause selection while learning from history. In *CADE*, 2021.
20. Martin Suda. Vampire with a brain is a good ITP hammer. In *FroCoS*, 2021.

21 Geoff Sutcliffe. The tptp problem library and associated infrastructure: From cnf to th0, tptp v6.4.0. volume 59, pages 483–502, 2017.

22 Josef Urban, Geoff Sutcliffe, Petr Pudlák, and Jiří Vyskočil. MaLARea SG1 — Machine Learner for Automated Reasoning with Semantic Guidance. In *IJCAR*, volume 5195 of *LNCS*, 2008.

23 Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *NeurIPS*, pages 5998–6008, 2017.

24 Uwe Waldmann, Sophie Tourret, Simon Robillard, and Jasmin Blanchette. A comprehensive framework for saturation theorem proving. In *IJCAR*, pages 316–334, 2020.

25 Ruibin Xiong, Yunchang Yang, Di He, Kai Zheng, Shuxin Zheng, Chen Xing, Huishuai Zhang, Yanyan Lan, Liwei Wang, and Tieyan Liu. On layer normalization in the transformer architecture. In *ICML*, pages 10524–10533, 2020.

A Encoder–scorer architecture

This appendix documents the *encode_score* configuration used in all experiments in this paper. Hyperparameters not listed below take the framework's defaults.

GCN clause encoder.

Three graph-convolutional layers [11] with hidden dimension $d = 128$, ReLU activations, and GraphNorm after each layer [3]. Node features are concatenated from node type (one-hot, 6 dim), arity (log-scaled, 1 dim), and argument position (sinusoidal encoding, 8 dim); symbol identity is not used as an input. The features are linearly projected to d before the first GCN layer. Clause-level pooling uses $1/\sqrt{N}$-scaled sum pooling, where N is the number of nodes in the clause graph.

Transformer scorer.

Four cross-attention layers with $h = 4$ heads, hidden dimension d, and pre-norm LayerNorm [25]. Unprocessed clause embeddings serve as queries; processed clauses provide keys and values. Each layer contains an attention sub-layer and a position-wise feed-forward sub-layer (hidden multiplier 4). The final clause representation is projected to a scalar score.

Selection.

At each given-clause step, scores over the unprocessed set are mapped to a categorical distribution by softmax with temperature $\tau = 1$, and one clause is sampled.

B Throughput-ablation problem set

The 16 TPTP problems used in Section 5 are, by domain: GEO082+1, GEO492+1 (geometry); GRP317-1, GRP366-1 (groups); LCL337-3, LCL670+1.010 (logic calculi); NUM255-1, NUM543+1 (number theory); SET017-4, SET175-6 (set theory); SEU218+3, SEU425+1 (set theory in untyped form); SWC063-1, SWC310-1 (software certification); SYN208-1, SYN683-1 (syntactic). All sixteen hit the 256-iteration cap under every cell of the ablation; the per-problem mean $|\Delta|$ per iteration is reported in the project's experiment log.