Lattice Deduction Transformers

Liam Davis
Amherst College
ljdavis27@amherst.edu

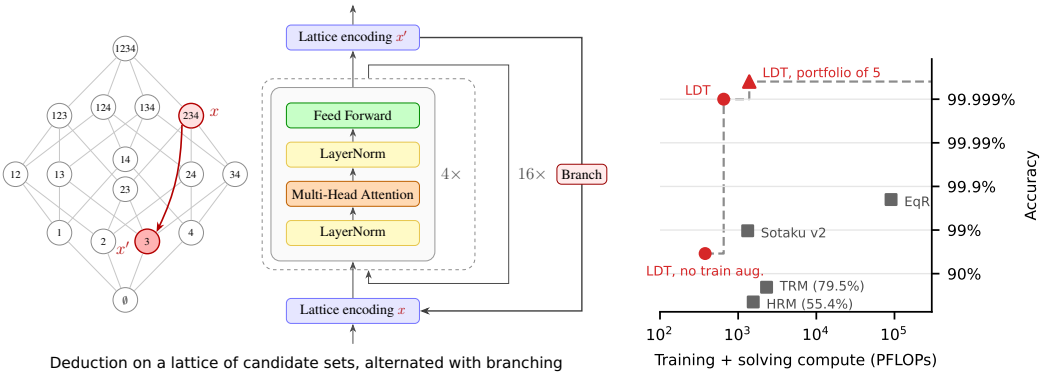
Leopold Haller
Axiom
leo@axiommath.ai

Alberto Alfarano
Axiom
alberto@axiommath.ai

Mark Santolucito
Barnard College, Columbia University
msantolu@barnard.edu

Abstract

We introduce the Lattice Deduction Transformer (LDT), a recurrent transformer that approximates logically sound deduction by projecting its latent state through a lattice between forward passes. We train on-policy in a process that mirrors deduction in a search-based constraint solver and supervise training via a domain-agnostic approximation of the set of solution candidates based on abstract interpretation. An 800K-parameter LDT outperforms prior small recurrent reasoners on Sudoku-Extreme with under 2% of their estimated training FLOPs, solving 99.999% of the 422,786 test puzzles. The model is nearly always sound, in that it gives a wrong answer for 4 of 1.27M puzzle evaluations, and otherwise returns a correct answer or abstains. A 1.8M-parameter LDT variant reaches 99.8% accuracy on Maze-Hard. Our code is available at github.com/lcrh/lattice-deduction-transformers.



(a) (b)

Figure 1: (a) The Lattice Deduction Transformer (LDT) performs deduction on a lattice using a recurrent transformer which is alternated with search via stochastic branching. (b) Sudoku-Extreme accuracy against the estimated FLOPs of training plus solving the 422,786-puzzle test set.

1 Introduction

Reasoning in large language models has become largely synonymous with the chain-of-thought paradigm [1] and its test-time-compute extensions [2, 3], in which the model generates intermediate reasoning tokens before producing a final answer. In contrast, automated logical reasoning procedures

40th Conference on Neural Information Processing Systems (NeurIPS 2026).

operate by iterative refinement of a partial information state. While the latent representations of neural networks can exhibit useful algebraic regularities, such as the vector arithmetic of word embeddings ("king" $-$ "man" $+$ "woman" $\approx$ "queen" [4]), partial information states require operations that approximate logical primitives such as conjunction and disjunction. Lattices used in abstract interpretation provide these primitives as greatest lower bounds (meet, $\sqcap$) and least upper bounds (join, $\sqcup$). For example, a lattice-based latent representation could represent "king" $\sqcup$ "queen" $\approx$ "royalty" and "royalty" $\sqcap$ "woman" $\approx$ "queen". We train a recurrent transformer whose output is an element of such a lattice, so that symbolic reasoning procedures can operate directly on its state.

We consider reasoning from the perspective of *abstract interpretation* [5, 6], which models information states as elements of a lattice with a top element $\top$ representing no information (every solution still possible) and a bottom element $\bot$ representing inconsistency (no solution remains). Deduction is a process that descends from $\top$ toward more informative states; a state that collapses to $\bot$ signals that no valid solution remains. Satisfiability procedures such as DPLL [7, 8], CDCL [9] and Stålmarck's procedure [10, 11] can be viewed as procedures that combine lattice-based deduction with search [12, 13]. Their deduction steps are sound: they may fail to derive a true fact, but never derive a false one. LDT replaces these hand-written steps with a learned transformer and does not inherit this guarantee; for LDT, soundness is an empirical property that we measure.

We use a recurrent transformer whose latent state is projected to and from a lattice representation between forward passes; we call this the Lattice Deduction Transformer (LDT). The interpretability of the abstract domain lets us leverage symbolic search processes at both training and inference time. On hard problems with logical structure, LDT exceeds comparable reasoning transformers with smaller models and less training. On some domains it further achieves empirical soundness given a sufficient training budget: the search either returns a correct answer or abstains.

We make four contributions. First, an architecture that projects a recurrent transformer's latent state onto an abstract lattice between forward passes, turning each forward pass into an approximately sound deduction step. Second, an on-policy training procedure where lattice states evolve under the model's own forward pass, keeping training states in-distribution with search states encountered during inference. Third, the *alpha operator*, which aggregates the valid solutions still consistent with the current lattice state. This places our training in a regime we are not aware of in prior work: on-policy yet still supervised by a domain-agnostic, state-dependent target. Fourth, an experimental characterization of the train/test compute trade-off in learned, approximately sound inference, showing that additional training shortens inference search by orders of magnitude.

The rest of the paper is organized as follows: Section 2 reviews related work. Section 3 introduces the lattices LDT reasons over and details its architecture, training and inference. Section 4 provides an evaluation of the Lattice Deduction Transformer against comparable reasoning transformers. Finally, we conclude and discuss future directions in section 5.

2 Related Work

Our approach draws from research on small recurrent reasoning models. Adaptive Computation Time [14] introduced variable-iteration neural computation for recurrent networks, later adapted to transformers by Universal Transformers [15]. The Hierarchical Reasoning Model [16] uses a dual-loop recurrent architecture, and the Tiny Recursive Model [17] shows that a 2-layer network with recursion can outperform it with a smaller model. Equilibrium Reasoners [18] train such recursive models to converge to attractors and scale their inference in depth and breadth. We build directly on Sotaku [19], a recurrent transformer specialized for Sudoku.

LDT is a neuro-symbolic method, combining a neural network with the lattice representation that search-based constraint solvers use to track partial information. SATNet [20] and Learning Modulo Theories [21] embed MAX-SAT and SMT solvers as network layers, preserving a hard neural/symbolic boundary. Closer to our setting, machine learning has been applied throughout the abstract-interpretation pipeline: Neural Abstract Interpretation [22] trains sound differentiable abstract transformers, LAIT [23] learns a neural policy that prunes redundant constraints, and SAIL [24] synthesizes sound transformers via LLMs with SMT validation. Earlier work targets the control loop instead, selecting among sound transformers via RL [25] or learning widening thresholds [26].

D3PM [27], the discrete-diffusion generalization of masked language modeling [28], iteratively commits information about partially-masked sequences; we recast this process as a fixed-point computation over a lattice. D3PM in fact admits an arbitrary token-transition matrix Q, which can be instantiated over tokens that themselves form a lattice – closely related in spirit to our setup, though we do not develop our method as a diffusion procedure. Several recent diffusion variants push in this direction. For example, ReMDM [29] reintroduces backtracking by allowing already-generated tokens to be remasked at inference time, Prime [30] relaxes the masked/unmasked dichotomy by letting tokens occupy intermediate states between $\top$ and a concrete value, and HDLM [31] organizes the vocabulary into a refinement hierarchy whose forward process abstracts upward and reverse process refines downward. Each can be read as moving along a particular partial-information ordering; in our framing, that ordering is the lattice.

3 Methodology

3.1 Abstract Interpretation

Following abstract interpretation [5, 6] (Appendix A), LDT represents what is known about a solution as an element of a *lattice*: a partially ordered set in which every pair of elements a, b has a greatest lower bound (meet) $a \sqcap b$ and a least upper bound (join) $a \sqcup b$. The order encodes precision: lower elements are more informative, the top element $\top$ carries no information, and the bottom element $\bot$ denotes inconsistency.

A solved Sudoku is a grid $g : \{1, \dots, 9\}^2 \to \{1, \dots, 9\}$. The sets of possible solutions form the powerset lattice $C = \mathcal{P}(G)$ over the set G of all grids, ordered by inclusion: an element is the set of grids still considered possible, $\top = G$ is our initial knowledge state, and $\bot = \emptyset$ means the puzzle has no solution. We call C the *concrete domain*. Since it is expensive to represent, we work instead in the following *abstract domain*, which keeps for each cell the digits still considered possible and drops all correlations between cells: $A = \{1, \dots, 9\}^2 \to \mathcal{P}(\{1, \dots, 9\})$, ordered pointwise ($a \sqsubseteq b$ iff $a(c) \subseteq b(c)$ for every cell c), with meet and join the pointwise intersection and union. $\top$ allows every digit in every cell, and a cell with no candidates means $\bot$. We call A the *grid powerset lattice*. The two domains are linked by an abstraction function $\alpha : C \to A$, which maps a set of grids to the digits each cell takes in some grid, $\alpha(G')(c) = \{g(c) \mid g \in G'\}$, and a concretization function $\gamma : A \to C$, which maps candidate sets to the grids they allow, $\gamma(a) = \{g \in G \mid \forall c.\ g(c) \in a(c)\}$. They form a Galois connection (Figure 2), so any property established on A holds on C.

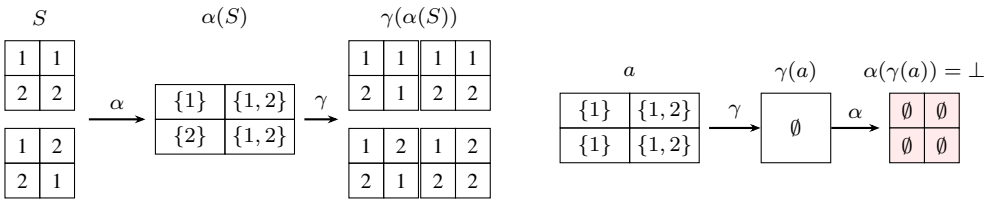


Figure 2: α and γ form a Galois connection with $\gamma \circ \alpha \supseteq \mathrm{id}_C$ and $\alpha \circ \gamma \sqsubseteq \mathrm{id}_A$.

For a puzzle p with solutions $\|p\| \subseteq G$, the most precise sound deduction operator on A is

$$ded_p(a) \;=\; \alpha\big(\gamma(a) \cap \|p\|\big),$$

which keeps a candidate only if some solution consistent with a uses it. Sudoku rules such as naked and hidden singles are sound but weaker deduction operators: each step removes candidates without ruling out a solution. LDT trains a transformer to approximate ded_p from solution samples. The learned operator is not guaranteed to be sound, so soundness becomes an empirical property, and how much to rely on training versus search becomes a trade-off, which we measure in Section 4.1.

The construction applies to any problem whose solutions are fixed-length token strings over a vocabulary V: with solution space $S = \{1, \dots, k\} \to V$, we take $C = \mathcal{P}(S)$ and $A = \{1, \dots, k\} \to \mathcal{P}(V)$, with α, γ and ded_p as above, positions in place of cells. We assume only that samples from $\|p\|$ are available for each training instance.

3.2 Architecture

Lattice encoding We represent the grid powerset lattice as a multi-hot encoding: $|V|$ binary sigmoids per cell (so $9 \times 9 \times 9 = 729$ sigmoids for 9×9 Sudoku; see Figure 3). Each sigmoid encodes the confidence that its candidate is still alive. Solutions y are one-hot (one bit alive per cell), and the abstraction α over a set of solutions reduces to a bit-level OR.

We give $\bot$ a dual representation. The first is implicit in the candidate encoding: a cell with an empty candidate set is the natural representation of $\bot$ in the powerset lattice. The second is an explicit binary sigmoid attached to a distinguished CLS token, which we train to fire on any unsatisfiable state. Carrying separate representations lets us train the two heads with different losses, discussed below.

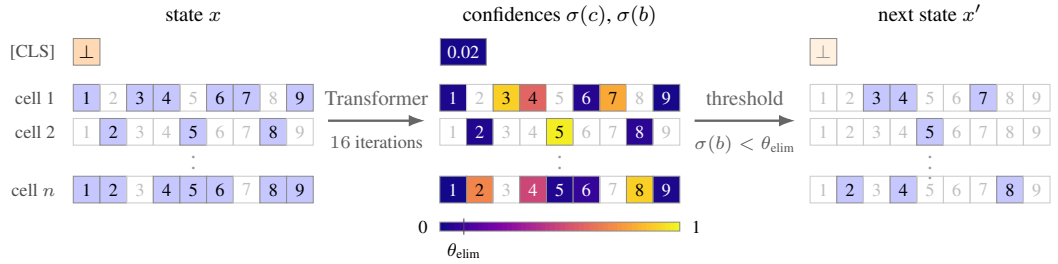


Figure 3: One step on the lattice-tensor encoding. The transformer maps the state x (alive candidates highlighted, eliminated ones greyed) to a confidence per alive candidate and a conflict confidence on the CLS token (color scale below). Thresholding removes candidates with $\sigma(b) < \theta_{\text{elim}}$ and flags a conflict if $\sigma(c) > \theta_{\text{CLS}}$; here cell 2 collapses to a single candidate. If the resulting state is neither a conflict nor a solution, a branching step then pins one cell (Algorithm 1).

Recurrent transformer The model is a recurrent transformer closely following Sotaku [19]: a stack of 4 attention layers is unrolled for 16 internal iterations, with the output of iteration ℓ becoming the input to iteration $\ell + 1$. We re-inject the input lattice encoding at every iteration as a residual signal. Each iteration emits its own candidate, digit and conflict logits. At training time every iteration is supervised and contributes equally to the loss; at inference time we read out only the final iteration. Position is encoded with a learned 2D positional embedding added to the input; for the larger 30×30 Maze grids we instead apply 2D RoPE [32] inside attention. We find large gains from recurrence (Appendix C.1): with a single iteration instead of 16, Sudoku-Extreme accuracy drops from 99.9% to 48%, and a single-iteration model still underperforms with $4\times$ the training compute or the full training split.

3.3 Solve Loop and Step Operator

A solve applies the step operator (Algorithm 1) repeatedly, starting from an initial lattice state x_0, until a step reaches a conflict or a solution. We run the same solve procedure at training and inference time, which ensures that the distribution of lattice states the model is trained on matches what it encounters at inference. Our setup shares similarities with classical supervised fine-tuning, on-policy distillation [33] and reinforcement learning [34]: as in the latter two, the training distribution is generated by the model's own rollouts and supervision is state-conditional; as in SFT, the target $\hat{y}$ is computed from data, which we do in a domain-agnostic manner via the abstraction operator α. This gives strong per-step supervision rather than a sparse trajectory-level reward or an approximate teacher signal. The solve loop is a stochastic search that moves down the lattice with deterministic deduction and random branching. The lattice guarantees termination and monotone tightening and lets the model abstain by leaving candidates alive; the search is sound only if every elimination is correct, which depends on the learned model and which we measure empirically (Section 4). This outer-loop structure contrasts with recurrent reasoners like TRM [17] and HRM [16], whose outer loop passes a latent embedding between steps; ours passes through the lattice itself. We do not propagate gradients across solve steps; Appendix F examines carrying continuous state between them.

A single step is summarized in Algorithm 1. We run the recurrent transformer, which unrolls its 16 internal iterations and produces per-candidate confidences at each grid cell and a CLS conflict signal. We then eliminate every candidate whose confidence falls below a threshold θ_{elim}. If the resulting state is neither a complete singleton solution nor flagged inconsistent, we pick a uniformly random multi-candidate cell (see Appendix D) and pin it to a single digit sampled from a softmax of the digit logits s over its alive candidates at temperature τ_{decide} (at inference, the argmax). The conflict and solved flags always come from the model: its CLS prediction and the empty-cell test.

Algorithm 1 Step operator $\mathrm{step}(x, Y)$

Require: lattice state x; ground-truth solutions Y (empty at inference)

$\big(b^{(1..L)},\, s^{(1..L)},\, c^{(1..L)}\big) \leftarrow f_\theta(x)$ ▷ candidate, digit and CLS logits at every iteration

$b \leftarrow b^{(L)}; \quad s \leftarrow s^{(L)}; \quad c \leftarrow c^{(L)}$

$x' \leftarrow x$; for every (i, j) with $\sigma(b_{ij}) < \theta_{\text{elim}}$, set $x'_{ij} \leftarrow 0$ ▷ threshold elimination

conflict $\leftarrow [\sigma(c) > \theta_{\text{CLS}}]$ **or** some cell of x' has zero alive candidates

solved $\leftarrow$ every cell of x' has exactly one alive candidate **and not** conflict

if $Y \neq \emptyset$ **then** ▷ in training, also use the ground truth

  $\hat{y} \leftarrow x \sqcap \alpha\big(\{y \in Y \mid y \text{ consistent with } x\}\big)$

  $\mathcal{L} \leftarrow \mathrm{compute_loss}\big(\hat{y},\, b^{(1..L)},\, c^{(1..L)}\big)$

  Update θ by one optimizer step on $\mathcal{L}$

if not conflict and not solved **then** ▷ branch (sample-pin one cell)

  Pick i^* uniformly at random from cells with ≥ 2 alive candidates

  Sample $d^* \sim \mathrm{softmax}(s_{i^*}/\tau_{\text{decide}})$ over alive candidates of i^* ▷ argmax at inference

  Pin cell i^* to d^* in x' (zero all other candidates of i^*)

return x', conflict, solved

Loss design The candidate-elimination and conflict-detection heads sit at opposite ends of the soundness/completeness trade-off. Candidate elimination must be sound (a wrongly eliminated candidate is unrecoverable) while conflict detection must be complete (a missed conflict can stall search or lead to returning a wrong answer). We supervise every one of the 16 internal iterations with three loss terms: $\mathcal{L}^{(\ell)}_{\text{BCE}}$, an asymmetric BCE between $\sigma(b^{(\ell)})$ and $\hat{y}$ with positive class weight w^+ and negative class weight $w^- < w^+$ to penalize false eliminations more heavily than false retentions; $\mathcal{L}^{(\ell)}_{\text{CLS}}$, a symmetric BCE between $\sigma(c^{(\ell)})$ and $\mathbf{1}[\hat{y} = \bot]$; and $\mathcal{L}^{(\ell)}_{\text{CE}}$, a per-cell softmax cross-entropy on digit logits $s^{(\ell)}$ from a separate linear head, at cells where $\hat{y}$ has a single alive candidate; s also picks the branching digit. $\mathcal{L}_{\text{CE}}$ substantially improves accuracy and sharpens the digit choice at branching steps (Appendix C.5). The total loss is

$$\mathcal{L} \;=\; \frac{1}{L}\sum_{\ell=1}^{L}\Big[\mathcal{L}^{(\ell)}_{\text{BCE}} + \lambda_{\text{cls}}\,\mathcal{L}^{(\ell)}_{\text{CLS}} + \lambda_{\text{ce}}\,\mathcal{L}^{(\ell)}_{\text{CE}}\Big]. \tag{1}$$

At inference time we eliminate candidates whose probability falls below a threshold θ_{elim} matched to the natural ratio induced by the asymmetric loss; this outperforms training with a symmetric BCE and tuning θ_{elim} post-hoc. When no $y \in Y$ is still consistent with x, the surviving set is empty and $\hat{y}$ collapses to $\bot$, which would push every candidate sigmoid toward zero and overload the BCE with conflict-detection pressure. The lattice contains many representations of an empty solution set; we pick one that lives further up the lattice by caching the last non-empty $\hat{y}^{\text{prev}}$ per chain and using $x \sqcap \hat{y}^{\text{prev}}$ as the BCE target once the surviving set becomes empty. This keeps the conflict localized to the individual cells where x has eliminated every candidate alive in $\hat{y}^{\text{prev}}$, focuses the BCE on sound deduction within still-satisfiable cells, and primarily leaves conflict detection to the CLS head (the candidate head can still flag localized conflicts when a cell collapses to empty). In our experiments we use $w^+/w^- = 8$, $\lambda_{\text{cls}} = 0.1$, $\lambda_{\text{ce}} = 0.2$, $\theta_{\text{elim}} \approx 0.1$, and decide temperature $\tau_{\text{decide}} = 1.5$, tuned on Sudoku; we use these settings as the starting point for the other domains and tune individual settings where needed (Appendix L).

Multi-solution supervision via the α operator Many problems of interest admit a set of solutions rather than a unique answer. Standard supervised fine-tuning, written against a single privileged ground truth, arbitrarily penalizes the model for committing to any of the equally valid alternatives,

echoing the insufficency of using token-level matching (BLEU score) for code generation [35]. We study this on Maze-Hard, where a maze admits many shortest paths. For each maze we pre-compute up to K valid solutions $Y = \{y^{(1)}, \ldots, y^{(K)}\}$ (each a one-hot point on the lattice), sampled uniformly at random from the all-shortest-paths DAG, and apply the abstraction function α to the subset of Y still consistent with the current state, as in Algorithm 1. As the state commits against solutions, the surviving subset shrinks: at branching cells where two or more solutions still survive, α remains multi-alive, and once the state has committed to a single solution's path, α collapses to a one-hot. A candidate is thus a positive target if it appears in at least one surviving solution. The special case $K = 1$ recovers standard single-target supervised learning, which we use for Sudoku and Snowflake Sudoku. Train-time overhead is minimal: the number of training steps does not depend on K, and K adds only minor bookkeeping per step – in our Maze experiments, $K = 512$ is roughly 4% slower per step than $K = 1$.

Parallel solve A solve follows a single linear chain from the initial state to a terminal, but in practice we parallelize both training and inference. For training, we maintain a pool of recent partially-deduced states, similar to a replay buffer except each instance contributes to a single gradient step. Each batch samples instances from the pool, runs Step, removes those the model solves, those it flags as conflicted when they truly are (checked against Y), and those older than 100 steps, and returns the rest to the pool. Over training, the pool develops a natural distribution over lattice depths, giving the model an implicit curriculum of progressively deeper states. At inference, we seed the batch with copies of a single puzzle and step each chain in parallel until one reaches a solution; detected conflicts are replaced by fresh puzzle instances.

Training and inference details The step operator reads the final iteration's logits, with dropout off, both in training and at inference; only the loss uses every iteration. Averaging the logits over iterations instead lowers accuracy (Appendix E.4). (i) Each step is wrapped in a random symmetry of the problem (Appendix L), which we invert after deduction and branching. We apply the same per-step augmentation during training (except for Sudoku, where it did not help). (ii) At inference we pin the highest-confidence candidate of the chosen cell instead of sampling it; diversity across chains comes from the random choice of branching cell, the per-step symmetry and eval-time dropout (not used for Maze). (iii) The CLS threshold at inference, $\theta_{\text{CLS}}^{\text{eval}}$, is set slightly higher than at training (we use 0.6 for Sudoku and Snowflake, and 0.53 for the 30×30 Maze-Hard runs). The conflict head grows more confident as the puzzle fills in, so raising the threshold suppresses false-positive conflicts on early-stage states, reducing the number of forward passes per solve. (iv) For Sudoku and Snowflake, a puzzle entering the training pool is the raw puzzle (20%), has some blanks filled with correct digits (55%), or has some filled with a share of wrong digits (25%), which gives the conflict head unsatisfiable states beyond those its own branching reaches and improves soundness (Appendix C.7). We skip this for Maze, whose mazes have multiple solutions.

4 Evaluation

4.1 Sudoku-Extreme

We evaluate Lattice Deduction Transformers on Sudoku-Extreme, a benchmark of hard Sudoku puzzles introduced alongside HRM [16]. We train an 800k parameter Lattice Deduction Transformer on the $1{,}000$ puzzle train set using the procedure described in Section 3.3. Sudoku-Extreme puzzles have unique solutions, so each instance is paired with a single ground-truth solution and the alpha operator reduces to the $K{=}1$ special case. We augment the training set with the symmetries of Sudoku: any digit permutation and the dihedral group D_4 acting on the grid (rotations and reflections of the 9×9 board). We find that tuning the conflict detection threshold during inference $\theta_{\text{CLS}}^{\text{eval}}$ can improve solver performance by reducing calls and avoiding timeouts; specifically, we use $\theta_{\text{CLS}}^{\text{eval}} = 0.6$, tuned on the train set.

We compare to TRM [17], HRM [16] and Sotaku v2 [36], evaluating released checkpoints (for TRM, a public alphaXiv reproduction), and to Equilibrium Reasoners (EqR) [18] as reported. Appendix J evaluates frontier chain-of-thought LLMs zero-shot on all three benchmarks.

Results Table 1 summarizes the results. LDT rows are means over 3 seeds, with at most $1{,}000$ search rounds per puzzle unless noted; EqR is as reported. To save compute, we evaluate the less

Table 1: Results on the 422,786-example Sudoku-Extreme test set.

					Inference	Training		
Model	**Params**	**Train set**	**Acc. %**	**Wrong %**	TFLOPs	PFLOPs	B200	MFU
HRM [16]	27M	1K‡	55.4	44.6	0.22	1,466	51m†	21%
TRM [17]	5M	1K‡	79.5	20.5	0.41	2,136	4.7h†	5.6%
Sotaku v2 [36]	800K	2.7M	98.96	1.04	0.14	1,265	4.1h†	3.8%
EqR [18]	5M	1K‡	99.8	—	210	1,640	30m†	40%
LDT, 1K train steps	800K	1K‡	99.36	0.06	9.6	4.7	3m	1.0%
LDT, 2K train steps	800K	1K‡	99.98	0.003	3.2	9.4	7m	1.1%
LDT, 4K train steps	800K	1K‡	99.988	0.0003	1.5	18.7	13m	1.1%
10,000 search rounds	800K	1K‡	**99.999**	0.0003	1.5	18.7	13m	1.1%
portfolio of 5 models, agree-2	4M	1K‡	99.9996	0.00008	3.1	93.5	1.1h	1.1%
portfolio of 15 models, agree-3	12M	1K‡	**100**	0	4.6	281	3.2h	1.1%
no training augmentation	800K	1K	96.54	3.46	0.85	18.7	11m	1.3%

‡Trained with data augmentation. †Estimated from measured B200 throughput.

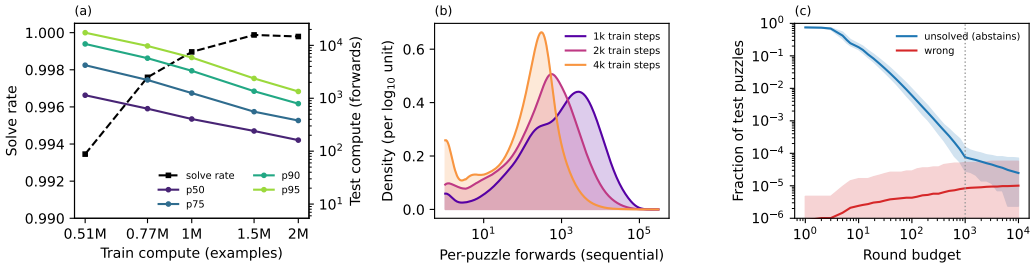


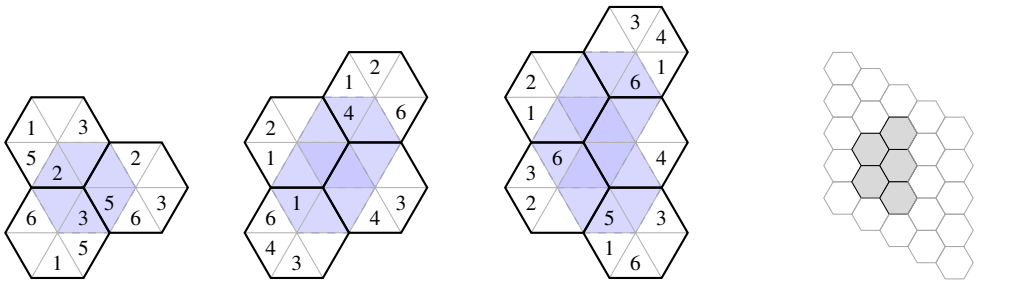
Figure 4: Test/train compute trade-off on Sudoku-Extreme. (a) Solve rate and percentiles of sequential forward passes per solved puzzle against training examples (training steps $\times$ batch size 512), mean over 3 seeds. (b) Density of per-puzzle sequential forward passes for models trained for 1K, 2K and 4K steps, pooled over seeds. (c) Fraction of the 422,786 test puzzles left unsolved (the model abstains) and answered wrong within a round budget, for 15 models trained for 4K steps on the same 1,000 puzzles (mean, range shaded); the dotted line marks the default budget of 1,000 rounds.

accurate models (HRM, TRM, Sotaku v2, the 1K- and 2K-step rows and the row without training augmentation) on random subsets of the test set (Appendix M). FLOPs and MFU are estimates (Appendix N). Trained for 4K steps on the 1,000 training puzzles, LDT solves 99.988% of all 422,786 test puzzles within 1,000 search rounds and 99.999% within 10,000, with 4 wrong answers in 1.27M puzzle evaluations over 3 seeds; every other unsolved puzzle is a timeout, where the model abstains. The released HRM and TRM checkpoints solve 55.4% and 79.5%, and their training uses 78 to 114 times as many FLOPs (Figure 1(b)). Training longer does not help on the 1,000 training puzzles: at 8K steps accuracy drops slightly and wrong answers rise to 282. The row marked "no training augmentation" trains without dataset or per-step augmentation but still applies per-step augmentation at evaluation. With only 1,000 training puzzles, training augmentation matters mostly for soundness: without it the model still solves 96.5% of the test set, but 3.5% of its answers are wrong, against 0.0003% with it (and it solves only 31.6% if evaluation is not augmented either, as it memorizes the training puzzles). Trained on the full 3.8M-puzzle training split without augmentation, the same model solves 99.994%. Portfolios of independently trained models raise accuracy further (Appendix B): on the test set, 5 models sometimes solve every puzzle (one of three portfolios), and requiring three of all 15 models to agree solves every test puzzle (Table 1). Search remains stochastic, so a small chance of error always remains.

Figure 4(a) shows the underlying train/test compute trade-off: more training shortens search, with the median and tail percentiles (p50, p75, p90, p95) of per-puzzle forward passes all falling, the median by a factor of 7 between 1K and 4K training steps (from 1,142 to 162, mean over seeds). Figure 4(b) shows the distribution: as training increases, the fraction of puzzles solved by the first deduction step grows from 2.5% to 10.9%, and the peak of the search mode moves to fewer forward passes;

better deductions translate to less guessing. Figure 4(c) shows the trade-off between completeness and soundness as the search budget grows: from 1 to 10,000 rounds, the fraction of unsolved puzzles falls from 75% to 0.0025%, while wrong answers rise from about 1 to 10 per million puzzles.

4.2 Snowflake Sudoku



Order 3 Order 4 Order 5 Order 5 in the covering grid

Figure 5: Snowflake Sudoku puzzles of orders 3, 4 and 5, each with a unique solution. The six cells of each hexagon must hold distinct values, as must the six shaded cells around each point where three hexagons meet. Every puzzle is placed in a fixed grid of 25 hexagon positions.

We use snowflake Sudoku to study whether LDT carries over to puzzles of variable size, and how it generalizes to sizes not seen in training. A snowflake Sudoku of order n consists of n hexagons joined edge to edge, each divided into six triangular cells, so it has $6n$ cells (Figure 5). Each cell takes a value in $\{1, \ldots, 6\}$. The six cells of each hexagon must be distinct, and so must the six cells around every point where three hexagons meet (two from each hexagon). Every puzzle has its own randomly grown layout, so the number and position of these cross-hexagon groups vary between puzzles. We place puzzles of every order in a fixed covering grid, with a read-only mask per cell marking which positions belong to the puzzle; Appendix K describes the data and training setup.

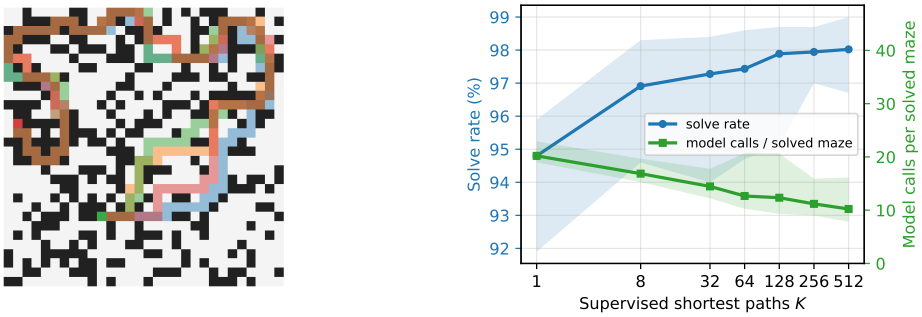
We train and test on minimal puzzles (no given can be removed without losing a unique solution) that naked and hidden singles alone cannot solve. These are still much easier than Sudoku-Extreme: current frontier LLMs solve 96–100% of them zero-shot, against 44–68% of Sudoku-Extreme puzzles (Appendix J). We compare LDT, the same LDT model in one shot (one model call, the highest-scoring digit in every cell), and direct prediction (the same model trained only with cross-entropy against the solution and evaluated in one shot). Table 2 summarizes the results. Trained on 1,000 puzzles of orders 4 to 8 for under an hour on a single B200, LDT solves every test puzzle in every seed, against 95.5% in one shot and 88.9% for direct prediction; with 2,000 training puzzles the gap to direct prediction narrows to 2.4 points. On puzzle sizes not seen in training the gap is larger (Appendix H). Trained on orders 6 to 8 and tested on orders 9 and 10, LDT solves 99.4% with 0.2% wrong answers, while one shot (78.9%) and direct prediction (78.4%) are tied, so the gain comes from search. Trained only on orders 4 to 6, every method degrades: LDT solves 77.2% of orders 7 to 10 but gives 15.8% wrong answers, against 67.4% for direct prediction. When training data is scarce or the test puzzles are far from the training sizes, LDT training appears to help twice: the LDT model's one-shot predictions are better than direct prediction's (95.5% against 88.9% with 1,000 puzzles, 54.0% against 32.6% trained on orders 4 to 6), and search improves them further. Training on states from the model's own solving may thus also help prediction without search.

Table 2: Snowflake Sudoku, minimal puzzles that singles cannot solve: accuracy / wrong answers (%), mean over 3 seeds. In distribution: orders 4 to 8, 1K or 2K training puzzles. Order transfer: training orders → test orders.

	In distribution		**Order transfer**	
	1K	**2K**	**6–8 → 9–10**	**4–6 → 7–10**
LDT	**100.00 / 0.00**	**100.00 / 0.00**	**99.40 / 0.22**	**77.21 / 15.80**
LDT, one shot	95.52 / 4.48	96.77 / 3.23	78.88 / 21.12	53.99 / 46.01
Direct prediction	88.87 / 11.13	97.63 / 2.37	78.43 / 21.57	32.60 / 67.40

4.3 Maze-Hard

We evaluate Lattice Deduction Transformers on Maze-Hard, a benchmark introduced alongside HRM [16]. The puzzles are 30×30 mazes whose shortest path between start and goal cells has length at least 110. Unlike Sudoku, a single instance can admit millions of distinct shortest paths (Figure 6a), so the alpha operator can run at $K > 1$ by sampling uniformly from the all-shortest-paths DAG, which is computationally inexpensive. Each cell takes one of two values (on or off the path), so eliminating one value selects the other; here, the lattice's role is to let the model leave a cell undecided.



(a) Maze-Hard example (30×30).

(b) Solve rate and model calls vs. K.

Figure 6: Maze-Hard task and multi-solution supervision via α. (a) one 30×30 Maze-Hard instance with start, goal, and several of the many shortest paths of equal length. (b) solve rate and model calls per solved maze as a function of the sample budget K: mean over 9 seeds (seed range shaded) on 1,000 test mazes of size 15×15, trained for 4,000 steps.

We follow the same experimental setup as Sudoku-Extreme, with adjustments for the larger grid: embedding dimension $d = 192$, 2D RoPE in place of the learned positional embedding, 20,000 training steps at batch size 192 and a conflict threshold of 0.53 at evaluation (Appendix L.3). We report two settings, both trained on the 1,000-puzzle Maze-Hard split (matching the TRM setup). The first uses $K = 1$, pairing each instance with a single ground-truth solution; the second uses $K = 512$, sampling 512 shortest paths per instance from the all-shortest-paths DAG.

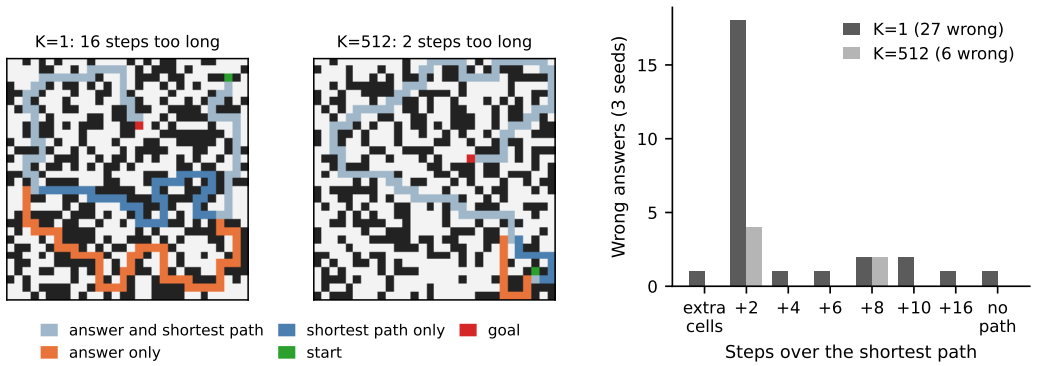
Table 3: Maze-Hard test results. A path is correct if it is *any* shortest path from start to goal. Every row reports the best checkpoint over training, selected on the test set. LDT: mean over 3 seeds. Training covers each method up to its reported checkpoint; FLOPs are estimates.

					Training		
Model	**Params**	**Train set**	**Acc. %**	**Wrong %**	PFLOPs	B200	MFU
HRM [16]	27M	1K‡	84.6	15.4	46,944	20.7h†	28%
TRM [17]	7M	1K	91.9	8.1	5,971	7.9h†	9.3%
LDT, $K = 1$	1.8M	1K‡	99.0	0.9	1,024	6.9h	1.8%
LDT, $K = 512$	1.8M	1K‡	**99.8**	0.2	1,303	9.1h	1.8%

‡Trained with data augmentation. †Estimated from measured B200 throughput.

Training is unstable for every method, so Table 3 reports the best checkpoint over training for each, selected on the test set (Appendix M); FLOPs are estimated as in Appendix N. HRM and TRM count a path as correct only if it matches the dataset's reference path; since every shortest path solves the maze, we accept any of them for all methods. We train HRM and TRM with and without $8\times$ dihedral augmentation and report each one's better run: augmentation lowers TRM's best accuracy from 91.9% to 21.3% and raises HRM's from 82.2% to 84.6%, reached only after about 122K steps instead of 39K. Under exact match, the reported HRM and TRM checkpoints score 16.7% and 84.5%. Table 3 reports two LDT settings, both trained on the 1,000-puzzle Maze-Hard split. With $K = 1$, LDT solves 99.0% of the test mazes; with $K = 512$, 99.8%, against 91.9% for the best TRM checkpoint. Neither setting is empirically sound: almost all unsolved mazes are wrong answers

rather than abstentions (27 wrong answers and 3 timeouts over 3 seeds at $K = 1$, 6 wrong answers at $K = 512$). Nearly all wrong answers are paths from start to goal that are slightly longer than the shortest path, most often by 2 steps (Figure 7; Appendix G).



(a) One wrong answer at each K.

(b) Excess length of all wrong answers.

Figure 7: LDT wrong answers on Maze-Hard (Table 3 checkpoints). (a) each answer against the shortest path that shares the most cells with it. (b) wrong answers over 3 seeds by the number of steps their path is longer than the shortest path; "extra cells" marks a shortest path with additional cells off the path, "no path" an answer that does not reach the goal.

Multi-solution supervision via α A typical 30×30 maze admits millions of distinct shortest paths, so $K = 512$ samples only a small fraction of the valid solution set. The solve loop's per-step target is $\hat{y} = x \sqcap \alpha(\{y \in Y \mid y \text{ consistent with } x\})$, so as K grows we feed the model an increasingly informative intersection of valid solutions and the supervision signal sharpens without changing the architecture or loss. Figure 6b plots solve rate and model calls per solved maze as a function of K; larger K yields a strictly tighter target, a higher solve rate, and shorter searches at inference. The gain is concentrated at small K and saturates quickly: most of the benefit is recovered with a handful of sampled solutions, and further increases yield diminishing returns. Tasks with different solution-set geometry may show different curves.

5 Conclusion

We introduced the Lattice Deduction Transformer (LDT), an architecture that iteratively refines a solution through forward passes capturing sound step-by-step deduction, rather than opaque latent updates. The model is trained via the solve procedure on a batch of partially-deduced lattice states that evolves under its own forward pass and supervised against the alpha operator. At inference, a parallel solve runs many chains per puzzle through the same deduction operator and exploits its soundness to either return a correct answer or abstain. Together, this allows an 800,000-parameter model to exceed prior small recurrent reasoners on hard reasoning benchmarks while rarely returning a wrong answer. While we instantiate LDT on combinatorial puzzles, the architecture is agnostic to the underlying lattice. In principle, any abstract domain admitting a deduction operator could be substituted.

Limitations LDT is well-suited to tasks with clear logical structure, where a fixed set of rules induces a sound deduction operator that the model learns to apply. Benchmarks like ARC-AGI [37, 38, 39] test programming-by-example, which demands something stronger: each task carries its own rules, which the model must *infer* from a handful of demonstrations before it can deduce anything about the test input. A naive port of LDT plateaus at roughly 36% pass rate, with no gains from test-time search. This is because the conflict head becomes unreliable, so the search cannot tell good branches from bad. Closing this gap will likely require additional machinery from TRM [17], in particular deep supervision that pushes a useful gradient through every intermediate lattice state and stronger recursive structure across deduction steps in place of simple iteration. More speculatively, this setting may require a different abstract domain entirely: one over the *programs* that produce solutions, rather than over the solution states themselves.

References

[1] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed Chi, Quoc Le, and Denny Zhou. Chain-of-thought prompting elicits reasoning in large language models. In *Advances in Neural Information Processing Systems*, volume 35, 2022.

[2] Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc Le, Ed H. Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. Self-consistency improves chain of thought reasoning in language models. In *International Conference on Learning Representations (ICLR)*, 2023.

[3] Charlie Snell, Jaehoon Lee, Kelvin Xu, and Aviral Kumar. Scaling LLM test-time compute optimally can be more effective than scaling parameters for reasoning. In *International Conference on Learning Representations (ICLR)*, 2025.

[4] Tomas Mikolov, Wen-tau Yih, and Geoffrey Zweig. Linguistic regularities in continuous space word representations. In *Proceedings of the 2013 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 746–751, 2013.

[5] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 238–252, Los Angeles, California, 1977. ACM Press, New York, NY.

[6] P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *Conference Record of the Sixth Annual ACM Symposium on Principles of Programming Languages*, pages 269–282, San Antonio, Texas, 1979. ACM Press, New York, NY.

[7] Martin Davis and Hilary Putnam. A computing procedure for quantification theory. *Journal of the ACM*, 7(3):201–215, 1960.

[8] Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem-proving. *Communications of the ACM*, 5(7):394–397, 1962.

[9] João P. Marques-Silva, Inês Lynce, and Sharad Malik. Conflict-driven clause learning SAT solvers. In Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, editors, *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*, pages 131–153. IOS Press, 2009.

[10] Mary Sheeran and Gunnar Stålmarck. A tutorial on stålmarck's proof procedure for propositional logic. In Ganesh Gopalakrishnan and Phillip Windley, editors, *Formal Methods in Computer-Aided Design*, pages 82–99, Berlin, Heidelberg, 1998. Springer Berlin Heidelberg.

[11] Sergei Leonov and Liam Davis. Two optimizations on the stålmarck procedure, 2025.

[12] Vijay D'Silva, Leopold Haller, and Daniel Kroening. Abstract conflict driven learning. In *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 143–154. ACM, 2013.

[13] Aditya V. Thakur and Thomas W. Reps. A generalization of stålmarck's method. In *Static Analysis – 19th International Symposium (SAS)*, volume 7460 of *Lecture Notes in Computer Science*, pages 334–351. Springer, 2012.

[14] Alex Graves. Adaptive computation time for recurrent neural networks, 2016.

[15] Mostafa Dehghani, Stephan Gouws, Oriol Vinyals, Jakob Uszkoreit, and Łukasz Kaiser. Universal transformers. In *International Conference on Learning Representations (ICLR)*, 2019.

[16] Guan Wang, Jin Li, Yuhao Sun, Xing Chen, Changling Liu, Yue Wu, Meng Lu, Sen Song, and Yasin Abbasi Yadkori. Hierarchical reasoning model, 2025.

[17] Alexia Jolicoeur-Martineau. Less is more: Recursive reasoning with tiny networks, 2025.

[18] Benhao Huang, Zhengyang Geng, and Zico Kolter. Equilibrium reasoners: Learning attractors enables scalable reasoning. In *Proceedings of the 43rd International Conference on Machine Learning*, 2026. arXiv:2605.21488.

[19] Cheng Lou. Sotaku: From-scratch experiments on iterative neural Sudoku solvers. Software, GitHub repository `https://github.com/chenglou/sotaku`, commit `9e13341`, 2026. Accessed: 2026-05-07.

[20] Po-Wei Wang, Priya L. Donti, Bryan Wilder, and J. Zico Kolter. SATNet: Bridging deep learning and logical reasoning using a differentiable satisfiability solver. In *Proceedings of the 36th International Conference on Machine Learning (ICML)*, volume 97 of *Proceedings of Machine Learning Research*, pages 6545–6554. PMLR, 2019.

[21] Matt Fredrikson, Kaiji Lu, Saranya Vijayakumar, Somesh Jha, Vijay Ganesh, and Zifan Wang. Learning modulo theories. *arXiv preprint arXiv:2301.11435*, 2023.

[22] Shaurya Gomber and Gagandeep Singh. Neural abstract interpretation. In *ICLR 2025 Workshop: VerifAI: AI Verification in the Wild*, 2025.

[23] Jingxuan He, Gagandeep Singh, Markus Püschel, and Martin Vechev. Learning fast and precise numerical analysis. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 1112–1127, 2020.

[24] Qiuhan Gu, Avaljot Singh, and Gagandeep Singh. SAIL: Sound abstract interpreters with LLMs. *Proceedings of the ACM on Programming Languages*, 10(PLDI), 2026. To appear.

[25] Gagandeep Singh, Markus Püschel, and Martin Vechev. Fast numerical program analysis with reinforcement learning. In *Computer Aided Verification (CAV)*, volume 10981 of *Lecture Notes in Computer Science*, pages 211–229. Springer, 2018.

[26] Sooyoung Cha, Sehun Jeong, and Hakjoo Oh. Learning a strategy for choosing widening thresholds from a large codebase. In *Programming Languages and Systems (APLAS)*, volume 10017 of *Lecture Notes in Computer Science*, pages 25–41. Springer, 2016.

[27] Jacob Austin, Daniel D. Johnson, Jonathan Ho, Daniel Tarlow, and Rianne van den Berg. Structured denoising diffusion models in discrete state-spaces. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2021.

[28] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (NAACL-HLT)*, pages 4171–4186, 2019.

[29] Guanghan Wang, Yair Schiff, Subham Sekhar Sahoo, and Volodymyr Kuleshov. Remasking discrete diffusion models with inference-time scaling. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2025.

[30] Chen-Hao Chao, Wei-Fang Sun, Hanwen Liang, Chun-Yi Lee, and Rahul G. Krishnan. Beyond masked and unmasked: Discrete diffusion models via partial masking. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2025.

[31] Cai Zhou, Chenyu Wang, Dinghuai Zhang, Shangyuan Tong, Yifei Wang, Stephen Bates, and Tommi Jaakkola. Next semantic scale prediction via hierarchical diffusion language models. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2025.

[32] Jianlin Su, Murtadha Ahmed, Yu Lu, Shengfeng Pan, Wen Bo, and Yunfeng Liu. RoFormer: Enhanced transformer with rotary position embedding. *Neurocomputing*, 568:127063, 2024.

[33] Rishabh Agarwal, Nino Vieillard, Yongchao Zhou, Piotr Stanczyk, Sabela Ramos, Matthieu Geist, and Olivier Bachem. On-policy distillation of language models: Learning from self-generated mistakes. In *International Conference on Learning Representations (ICLR)*, 2024.

[34] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. MIT Press, 2 edition, 2018.

[35] Shuo Ren, Daya Guo, Shuai Lu, Long Zhou, Shujie Liu, Duyu Tang, Neel Sundaresan, Ming Zhou, Ambrosio Blanco, and Shuai Ma. Codebleu: a method for automatic evaluation of code synthesis, 2020.

[36] Cheng Lou. Sotaku v2: From-scratch experiments on iterative neural Sudoku solvers. Software, GitHub repository `https://github.com/chenglou/sotaku`, commit `d0ab125`, 2026. Accessed: 2026-10-23.

[37] François Chollet. On the measure of intelligence. *arXiv preprint arXiv:1911.01547*, 2019.

[38] Francois Chollet, Mike Knoop, Gregory Kamradt, Bryan Landers, and Henry Pinkard. Arc-agi-2: A new challenge for frontier ai reasoning systems, 2025.

[39] ARC Prize Foundation. Arc-agi-3: A new challenge for frontier agentic intelligence, 2026.

[40] Anthropic. Introducing claude opus 4.6. Anthropic News, February 2026. Accessed: 2026-04-29.

[41] DeepSeek-AI. Deepseek-v4 preview release: Ushering in the era of 1m context length. DeepSeek Official Blog, April 2026. Accessed: 2026-04-29.

[42] OpenAI. Gpt-5.4 thinking system card. Technical report, OpenAI, March 2026.

[43] Haniel Barbosa, Clark W. Barrett, Martin Brain, Gereon Kremer, Hanna Lachnitt, Makai Mann, Abdalrhman Mohamed, Mudathir Mohamed, Aina Niemetz, Andres Nötzli, Alex Ozdemir, Mathias Preiner, Andrew Reynolds, Ying Sheng, Cesare Tinelli, and Yoni Zohar. cvc5: A versatile and industrial-strength SMT solver. In Dana Fisman and Grigore Rosu, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 28th International Conference, TACAS 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2-7, 2022, Proceedings, Part I*, Lecture Notes in Computer Science, pages 415–442. Springer, 2022.

A Abstract Interpretation for Logical Problems

This appendix gives a formal introduction to abstract interpretation. The framework is most often presented in the setting of program analysis, where the concrete domain is the (typically infinite) set of program states; here we sketch the simpler instance that suffices for our purposes, in which the concrete domain captures candidate solutions to a finite-domain logical problem such as Sudoku.

A.1 Lattices and Galois Connections

A *lattice* $(L, \sqsubseteq, \sqcup, \sqcap)$ is a set L equipped with a partial order $\sqsubseteq$ in which every pair $a, b \in L$ has a least upper bound $a \sqcup b$ (the *join*) and a greatest lower bound $a \sqcap b$ (the *meet*). A lattice is *bounded* if it has a top element $\top$ above every element and a bottom element $\bot$ below every element. Following the main text, we read $\sqsubseteq$ as a precision ordering: $a \sqsubseteq b$ means a is at least as informative as b, with $\bot$ inconsistent and $\top$ uninformative.

Recall that a Sudoku solution is a function $g : \{1, \dots, 9\}^2 \to \{1, \dots, 9\}$, and write G for the set of all such grids. We construct two lattices over G.

Our *concrete domain* is the *powerset lattice* $C = (2^G, \subseteq, \cup, \cap)$ over G, with $\bot = \emptyset$ and $\top = G$. An element of C is the set of complete grids still considered viable; smaller sets carry more information about which grid is correct and therefore sit lower in the lattice.

The *grid powerset lattice* is the per-cell pointwise version, $A = (\{1, \dots, 9\}^2 \to \mathcal{P}(\{1, \dots, 9\}))$, ordered by $a \sqsubseteq b$ iff $a(c) \subseteq b(c)$ for every cell c. Meet is pointwise intersection, join is pointwise union, $\top$ is the constant map sending every cell to $\{1, \dots, 9\}$, and $\bot$ is identified with any map having an empty candidate set at some cell. An element of A records, for each cell, which digits are still considered viable for that cell. By construction, the grid powerset lattice loses all relational information across cells: any correlation between distinct cell values present in a concrete state is forgotten when only per-cell candidate sets are tracked.

An abstract element $a \in A$ can be read as the set of complete grids that respect its candidate sets. We make this precise with the *concretization* map $\gamma : A \to C$,

$$\gamma(a) = \{\, g \in G : g(c) \in a(c) \text{ for every cell } c \,\}.$$

Concretization is monotone: $a \sqsubseteq b$ implies $\gamma(a) \subseteq \gamma(b)$.

Conversely, given $S \in C$ we look for the most informative abstract element whose concretization still covers S — that is, $S \subseteq \gamma(a)$, so that abstracting away does not discard any candidate in S. The *abstraction* map $\alpha : C \to A$ returns this best sound approximation, given elementwise by

$$\alpha(S)(c) = \{\, g(c) : g \in S \,\},$$

the set of digits assigned to cell c by some grid in S. The pair (α, γ) forms a *Galois connection*, written in the standard notation

$$C \underset{\alpha}{\overset{\gamma}{\leftarrows\!\!\rightarrow}} A.$$

Here α is the *lower adjoint* (the lower, rightward arrow) and γ is the *upper adjoint* (the upper, leftward arrow). The two arrows encode two round-trip conditions, each obtained by tracing a path from one side back to itself:

$$S \subseteq \gamma(\alpha(S))$$

$$\alpha(\gamma(a)) \sqsubseteq a.$$

We use $\subseteq$ for the concrete order on C and $\sqsubseteq$ for the abstract order on A. The first condition says that going to the abstract domain and back can only *add* candidates to the original solution set: abstraction is sound because the round trip overapproximates. The second is more intuitive: going to the concrete and back lets us *refine* the abstract representation by stripping out structure that no concrete element witnesses. Figures 8 and 9 illustrate the two directions on 2×2 Sudoku puzzles.

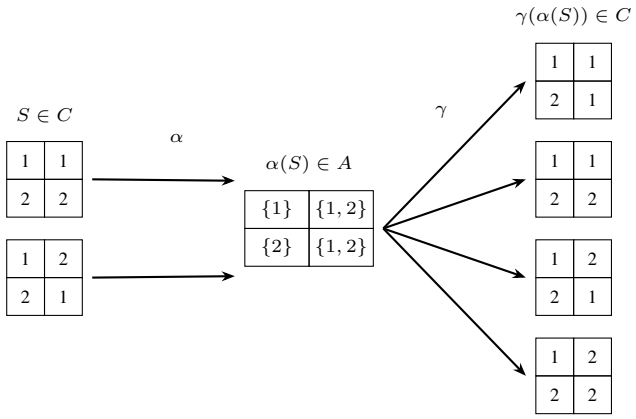


Figure 8: Round-trip $\gamma \circ \alpha$ on 2×2 Sudoku. The set S contains two grids that differ in the right-hand column. Abstraction drops the cell-to-cell correlation, so concretizing the result yields four grids – the original two plus two additional grids that satisfy the per-cell candidate sets but were not in S.

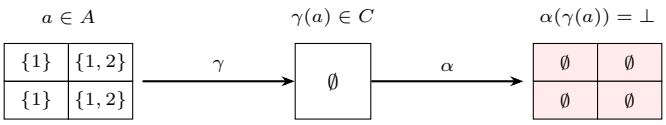


Figure 9: Round-trip $\alpha \circ \gamma$ on an impossible abstract element. Column 1 of a forces both rows to take the value 1, contradicting the column constraint, so $\gamma(a) = \emptyset$. Re-abstracting then collapses every cell to the empty candidate set, refining a all the way to $\bot$.

A.2 Sound Function Approximation and Deduction

Abstract interpretation is usually concerned not with single states but with *operations* on states – for instance, the function that takes a program point to its successor state. We are therefore interested in approximating monotone functions $f : L \to L$ on a lattice. In the abstract interpretation literature these are called *transformers*; we avoid that term to prevent confusion with the transformer neural architecture, and simply refer to monotone functions on L.

The intuition for monotonicity is that the function is applied uniformly to every member of a state of knowledge: if a is at least as informative as b, then $f(a)$ is at least as informative as $f(b)$, because strengthening the input can only produce a more committed output.

Given lattices L_1 and L_2, the monotone functions $L_1 \to L_2$ themselves form a lattice under the pointwise order: $f \sqsubseteq g$ iff $f(x) \sqsubseteq g(x)$ for every x. A Galois connection $C \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} A$ on point lattices lifts to a Galois connection between the function lattices $[C \to C]$ and $[A \to A]$, with abstraction $\alpha^\sharp(f) = \alpha \circ f \circ \gamma$ and concretization $\gamma^\sharp(f^\sharp) = \gamma \circ f^\sharp \circ \alpha$. Figure 10 sketches the construction.

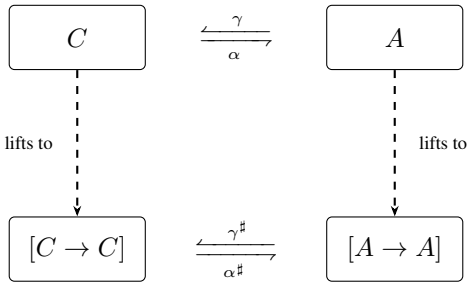


Figure 10: Lifting a Galois connection to monotone-function lattices. The Galois connection between the point lattices C and A induces a Galois connection between the corresponding lattices of monotone functions, with $\alpha^\sharp(f) = \alpha \circ f \circ \gamma$ and $\gamma^\sharp(f^\sharp) = \gamma \circ f^\sharp \circ \alpha$.

For a concrete monotone function f, the lifted abstraction $\alpha^\sharp(f) = \alpha \circ f \circ \gamma$ is the most precise sound approximation of f on A (traditionally called the *best abstract transformer*). It is typically intractable to compute directly, so in practice one works with a weaker abstract function $f^\sharp$ that is still sound, meaning $\alpha \circ f \circ \gamma \sqsubseteq f^\sharp$.

The result that justifies all of this is that fixed points transfer [5, 6]: if f is monotone on C and $f^\sharp$ is any sound abstraction, then

$$\alpha(\mathrm{lfp}\, f) \sqsubseteq \mathrm{lfp}\, f^\sharp \qquad \text{and} \qquad \alpha(\mathrm{gfp}\, f) \sqsubseteq \mathrm{gfp}\, f^\sharp.$$

Computing a fixed point in the abstract domain therefore yields a sound overapproximation of the corresponding fixed point in the concrete.

Logical problems and deduction In a logical problem the goal is solution-finding rather than the analysis of a dynamic system over time. The function of interest is the *deduction function* of the puzzle ϕ,

$$\mathrm{ded}_\phi : C \to C, \qquad \mathrm{ded}_\phi(S) = \{\, g \in S : g \text{ satisfies } \phi \,\},$$

which filters a candidate set down to those grids that respect the constraints imposed by ϕ. (The function ded_ϕ is a *lower closure operator*: monotone, reductive $\mathrm{ded}_\phi(S) \subseteq S$, and idempotent.) The same pattern applies to any logic L whose formulas are interpreted over a set of models: ded_ϕ keeps only the models of ϕ.

The greatest fixed point of ded_ϕ from $\top = G$ is the set of all solutions to ϕ, and a single application of the best abstract transformer $\alpha \circ \mathrm{ded}_\phi \circ \gamma$ would solve the puzzle in one shot. Practical deduction rules – naked singles, hidden singles, naked pairs, X-wings, and so on – are sound but weaker abstract approximations of ded_ϕ: each removes only some of the candidates the full filter would. Solving a Sudoku then amounts to computing the greatest fixed point of an abstract deduction function in the grid powerset lattice, starting from $\top$ and descending until no further candidates can be removed. Depending on the strength of the chosen rules, the fixed point is either a fully determined grid (the

puzzle is solved) or one that leaves multiple candidates open in some cells, in which case search must be invoked to finish the job.

A dual function of interest is the *abduction function* $\mathrm{abd}_\phi : C \to C$, which extends a candidate set by adding all grids that fail to satisfy ϕ:

$$\mathrm{abd}_\phi(S) = S \cup \{\, g \in G : g \not\models \phi \,\}.$$

Where deduction lives in the over-approximation regime we have discussed so far – abstract states bound the true answer from above – abduction lives in the dual under-approximation regime: an underapproximation of abd_ϕ identifies a region of the search space whose elements are all guaranteed failures. This is the engine of conflict analysis in modern SAT solvers [9]: on a failed search branch, the solver generalizes the witnessed failure into a learned clause that excludes a much larger region than just the assignment that failed, and this adaptive search-space compression (clause learning) is the core reason for the extraordinary performance of modern SAT solvers. The clause-learning step can itself be cast as an abstract interpretation [12]. One motivation for this work is to begin closing the gap between diffusion-style sampling procedures and the advanced search techniques developed in the automated reasoning community.

A note for readers familiar with program analysis The shape of this setup may look unfamiliar to readers fluent in abstract interpretation as it is usually presented. Classically one computes a *least* fixed point over a monotone function encoding program semantics – typically the strongest-postcondition transformer forward, or the weakest-precondition transformer backward. Here we instead compute a *greatest* fixed point over an approximation of a lower closure operator. The difference is purely conventional and historical: abstract interpretation is fundamentally a theory of sound approximation and representation in inferential computation, and the program-semantics framing is one application among several. Logical deduction, constraint satisfaction, and the puzzle setting we use here all fall comfortably within the same theory.

Program analyzers already compute greatest fixed points of this kind when they evaluate a branch condition. To refine the abstract state that enters the branch, they apply the guard's conjuncts repeatedly until the state stops changing. Consider the guard

```
if x < y and y < z and z < 3:
```

with integer variables in $[0, 100]$, analyzed in the interval domain. One pass through the three conjuncts gives $z \in [2, 2]$ but leaves $y \in [1, 99]$ and $x \in [0, 99]$; the second pass gives $y \in [1, 1]$; only the third gives $x \in [0, 0]$, and a fourth pass confirms that the state no longer changes. Each pass is a sound deduction step in our sense, and the refined state is the greatest fixed point of these steps below the incoming state.

B Portfolios on Sudoku-Extreme

A single 4K-step model leaves about 1 in 8,000 test puzzles unsolved within 1,000 rounds and answers a few wrong (Table 1). Independently trained models fail on largely different puzzles, so a portfolio of models can resolve most of these failures. We train 15 models for 4K steps on the same 1,000 training puzzles, differing only in the training seed (seeds 0–2 are the 4K models of Table 1).

Rules A portfolio runs its models in a fixed order and combines their answers with one of four rules: *first* returns the first answer; *agree-k* runs models until k of them return the same answer and abstains if none does; *majority* returns the most frequent answer of all models and abstains on a tie. A model's answer is its answer within 1,000 rounds; on the test split, a model that times out is given 10,000 rounds instead.

Choosing a rule on held-out puzzles To choose the rule and portfolio size without the test set, we evaluate all 15 models on 100,000 training-split puzzles that none of them was trained on, and simulate every portfolio size m over 200 random model orders. Table 4 scales the resulting failure rates to the 422,786 test puzzles, against the portfolio's training compute (m times the 12.7 B200 minutes of one model).

Taking the first answer quickly removes unresolved puzzles but keeps every wrong answer. Requiring two agreeing answers removes the wrong answers entirely in simulation: no two models returned

Table 4: Expected wrong answers / unresolved puzzles on the 422,786 test puzzles for same-data portfolios of m models, extrapolated from simulations on 100,000 held-out puzzles at 1,000 rounds (mean over 200 random model orders). Training is the portfolio's total B200 time.

m	Training	first	agree-2	majority
1	0.2h	3.0 / 59	—	3.0 / 59
2	0.4h	3.3 / 20	0 / 102	0.7 / 25
3	0.6h	3.4 / 11	0 / 38	0.4 / 12
5	1.1h	3.8 / 4.8	0 / 17	0.4 / 5.8
7	1.5h	3.8 / 2.6	0 / 8.9	0.1 / 3.4
10	2.1h	3.8 / 1.6	0 / 5.1	0 / 1.6
13	2.8h	3.8 / 0.5	0 / 4.4	0 / 0.5
15	3.2h	3.8 / 0	0 / 4.2	0 / 0

the same wrong grid on any held-out puzzle, so agree-2 never accepted one. Majority voting lies in between and reaches zero failures with all 15 models.

Abstention keeps portfolios cheap A single model abstains on 0.014% of the held-out puzzles and answers wrongly on 0.0007%. Because a model that has not found a solution abstains rather than guessing, a portfolio can accept an answer as soon as it appears and run further models only on the puzzles that the earlier ones left unresolved. With *first*, a 15-model portfolio uses 1.01 times the model calls of a single model; with agree-2, which waits for a second matching answer, 2.0 times. A model that always answered would give no such signal, so every puzzle would need votes from several models, at m times the cost of one model for majority voting over m models.

Failure rate beyond the test set To estimate failure rates below what a test set can measure, we model each puzzle i as failing a model with probability p_i: the model returns no correct answer within 10,000 rounds. A failure is a wrong answer with probability $r = 0.41$, the share of the test failures that are wrong answers, and an abstention otherwise. We fit a $\mathrm{Beta}(a, b)$ prior on p by maximum likelihood to the number of the 15 models that fail each test puzzle ($a = 1.2 \times 10^{-4}$, $b = 4.6$), and update it with each puzzle's own count to a posterior $\mathrm{Beta}(a + f_i, b + 15 - f_i)$. A new model is drawn from this posterior, independently of the others. Majority voting over m new models fails a puzzle when none of them answers correctly, or when exactly one does and another returns a wrong grid (a tie, assuming distinct wrong grids). We compute this probability in closed form, which equals the limit of simulating many drawn models, and average it over the test puzzles. Figure 11 compares the derived rate with majority voting measured over subsets of the 15 models. The derived rate is within 30% of the measured one up to 5 models and up to 3.6 times lower from 8 to 13 models. It falls below one failure per million puzzles at 8 models and below one per billion at 34. A bootstrap over puzzles and models (resampled with replacement, prior refit each time) puts these at 6 to 17 models and at 21 to more than 200 models (90% intervals): the estimate is precise only within the range of measured portfolio sizes. As an out-of-sample check, the same model fit on the 100,000 held-out puzzles at 1,000 rounds predicts majority voting measured on the test split at 1,000 rounds within a factor of 0.7 to 1.1 for 1 to 13 models.

Test results We then evaluate three disjoint 5-model portfolios and all 15 models on the full test split (Table 5). With agree-2, one 5-model portfolio and the 15-model portfolio solve every test puzzle with no wrong answer. Agreement is not a guarantee of correctness, however: on 3 of the 422,786 test puzzles, two or three models returned the same wrong grid, and one of these is the wrong answer of portfolio seeds 10–14 under agree-2. Agree-k also depends on the order in which the models run: over random orders of the 15 models, agree-2 solves every test puzzle in 60% of orders and agree-3 in 82%, while majority voting over 14 or 15 models solves every test puzzle for every choice of models.

C Ablation Study on Sudoku-Extreme

The Sudoku-Extreme LDT configuration combines several design choices: a weight-tied backbone of 4 layers unrolled for $L = 16$ internal iterations, supervision of every internal iteration, an auxiliary

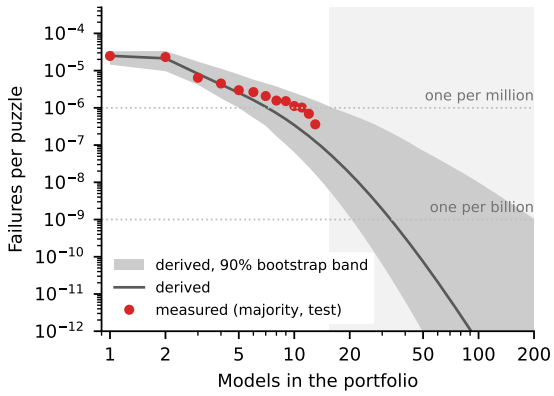


Figure 11: Failure rate (wrong answers and unresolved puzzles per puzzle) of majority voting against portfolio size on the test split, measured over subsets of the 15 models and derived from per-puzzle failure probabilities, with a 90% bootstrap band. The shaded region lies beyond the 15 trained models.

Table 5: Same-data portfolios on all 422,786 Sudoku-Extreme test puzzles: wrong answers / unresolved puzzles (accuracy %).

Portfolio	**first**	**agree-2**	**agree-3**	**majority**
seeds 0–4	2 / 0 (99.9995)	0 / 0 (**100**)	0 / 2 (99.9995)	0 / 0 (**100**)
seeds 5–9	4 / 0 (99.9991)	0 / 2 (99.9995)	0 / 4 (99.9991)	0 / 1 (99.9998)
seeds 10–14	3 / 0 (99.9993)	1 / 2 (99.9993)	0 / 6 (99.9986)	1 / 1 (99.9995)
all 15	2 / 0 (99.9995)	0 / 0 (**100**)	0 / 0 (**100**)	0 / 0 (**100**)

per-cell cross-entropy $\mathcal{L}_{\text{CE}}$, an asymmetric candidate-elimination loss, and the CLS conflict head. This appendix changes one of these choices at a time and measures the effect.

Protocol Every arm uses the Sudoku-Extreme configuration of Appendix L.1 ($d = 128$, 4 layers, $L = 16$, batch size 512, the same 1,000 training puzzles with dataset-level augmentation) with two changes: training stops at 2,000 steps instead of 4,000, and training also applies per-step augmentation (Appendix L). We call this configuration the reference. At 2,000 steps the reference fails on about 1 in 1,700 test puzzles, which leaves enough failures to compare arms at half the training cost of the 4,000-step model. Each arm changes the settings listed in its table row and nothing else. Arms are trained with seeds 0, 1 and 2; arms with $4\times$ the reference's training compute or the full training split use seeds 0 and 1. Training compute is counted in FLOPs as in Appendix N; the reference uses 9.35 PFLOPs. All arms use the inference settings of Table 1 (64 chains, 1,000 rounds, $\theta_{\text{elim}} = 0.1$) unless a row changes them.

Each seed is evaluated with the sequential rule of Appendix M on the same fixed ordering of the test split: evaluation stops at 100 failures (wrong answers plus timeouts) or once the 95% accuracy interval has half-width at most 0.1 percentage points, and in any case after 4,096 puzzles. For the reference (except in Table 10), the loop-count arms, both λ_{ce} arms and the two $32\times$-asymmetry arms, the cap is 10,240 puzzles instead. Arms with low accuracy stop early at 100 failures per seed, so their accuracies have wider intervals.

Each table reports, per arm: n, the number of test puzzles evaluated, summed over seeds; accuracy, the percentage of these puzzles solved (mean $\pm$ standard deviation over seeds); Δ, the accuracy difference from the reference in percentage points, paired by puzzle within each seed over the puzzles both runs evaluated, pooled over seeds, with a 95% interval in brackets; calls per solve ("Calls"), the number of batched model calls spent on all evaluated puzzles divided by the number solved; and the numbers of wrong answers and timeouts, summed over seeds. A wrong answer is a returned grid that differs from the solution; a timeout is a puzzle for which no chain returns a grid within the round budget.

C.1 Is recurrence necessary?

We vary the number of internal iterations L and keep the weight-tied 4-layer, $d = 128$ backbone ($\approx$ 800K parameters) and all other settings fixed (Table 6). Every row has the same parameter count and the same 2,000 training steps; the computation per model call, and therefore the training compute, is proportional to L.

Table 6: Number of internal iterations L at a fixed parameter budget ($\approx$ 800K), 2,000 training steps, 3 seeds. Training compute is relative to the reference ($L = 16$).

L	Compute	n	Acc. %	Δ (pp)	Calls	Wrong	Timeout
1	0.06×	578	47.86 ± 4.30	-51.90 [-55.98, -47.83]	169	0	300
2	0.13×	4,781	93.35 ± 1.80	-6.21 [-6.90, -5.52]	31.6	0	300
4	0.25×	30,720	99.68 ± 0.09	-0.26 [-0.32, -0.19]	7.7	0	97
8	0.5×	23,200	$\mathbf{99.98 \pm 0.01}$	$+0.04$ [$+0.00$, $+0.07$]	3.5	0	4
16 (reference)	1×	30,720	99.94 ± 0.02	—	3.9	0	18
32	2×	30,720	99.91 ± 0.04	-0.03 [-0.07, $+0.01$]	3.7	0	27

Accuracy rises from 47.9% at $L = 1$ to 93.4% at $L = 2$ and 99.7% at $L = 4$. From $L = 8$ on it stays between 99.9% and 100%: $L = 8$ is 0.04 points above the reference, and $L = 32$ is not distinguishable from it. Calls per solve fall from 169 at $L = 1$ to between 3.5 and 3.9 for $L \geq 8$, so more iterations per call means fewer guesses per solved puzzle. No loop count returns a wrong answer. Rows with fewer iterations also use less training compute; Section C.2 removes this difference.

C.2 Can more training match the performance of recurrence?

We retrain models with $L = 1$ and $L = 2$ for more optimizer steps, so that their total training compute matches the reference (1×) or is four times larger (4×). One 4× arm also replaces the 1,000 training puzzles with the full training split ($\approx$ 3.8M puzzles). Table 7 lists the arms; the 2,000-step rows for $L = 1$ and $L = 2$ are repeated from Table 6.

Table 7: Models with $L = 1$ or $L = 2$ trained with more optimizer steps. Training compute is relative to the reference. "Full split" trains on the full training split instead of 1,000 puzzles. 3 seeds; rows marked * use 2.

Model, steps	Compute	n	Acc. %	Δ (pp)	Calls	Wrong	Timeout
$L = 2$, 2K	0.13×	4,781	93.35 ± 1.80	-6.21 [-6.90, -5.52]	31.6	0	300
$L = 2$, 16K	1×	12,288	99.81 ± 0.09	-0.11 [-0.20, -0.03]	6.7	1	22
$L = 1$, 2K	0.06×	578	47.86 ± 4.30	-51.90 [-55.98, -47.83]	169	0	300
$L = 1$, 32K	1×	2,902	89.15 ± 2.69	-10.27 [-11.38, -9.16]	40.3	0	300
$L = 1$, 128K*	4×	6,017	96.59 ± 0.78	-3.26 [-3.71, -2.80]	21.0	1	199
$L = 1$, 128K, full split*	4×	6,900	97.62 ± 1.68	-2.07 [-2.42, -1.72]	18.8	0	149
Reference ($L = 16$, 2K)	1×	30,720	$\mathbf{99.94 \pm 0.02}$	—	3.9	0	18

At matched training compute, $L = 2$ reaches 99.81%, 0.11 points below the reference, with 1.7× the calls per solve and one wrong answer. $L = 1$ reaches 89.1% at matched compute and 96.6% at four times the compute. Training on the full split instead of 1,000 puzzles raises this to 97.6%, still 2.1 points below the reference, with 4.8× its calls per solve. In every configuration we tested, a model with a single iteration stays below the recurrent reference.

C.3 Can a different non-recurrent architecture substitute?

A non-recurrent transformer cannot apply the same weights several times per call, so its computation per call is fixed by its parameter count. We test two ways of compensating. First, we keep $\approx$ 800K parameters and redistribute them across depth and width, with one iteration ($L = 1$) and 2,000 training steps (Table 8). The 4×128 row is the $L = 1$ row of Table 6. Each of these models uses about 0.07× the reference's training compute.

Table 8: Non-recurrent models ($L = 1$) at the reference's parameter budget, varying depth and width. 2,000 training steps, 3 seeds.

Layers × width	Params	n	Acc. %	Δ (pp)	Calls	Wrong	Timeout
4×128	0.80M	578	47.86 ± 4.30	-51.90 [-55.98, -47.83]	169	0	300
8×92	0.83M	1,229	73.23 ± 8.78	-24.41 [-26.81, -22.01]	80.7	9	291
16×64	0.80M	525	41.42 ± 10.85	-57.14 [-61.38, -52.91]	189	39	261
32×44	0.76M	453	30.53 ± 17.80	-66.23 [-70.58, -61.87]	492	35	265
Reference ($L = 16$)	0.80M	30,720	$\mathbf{99.94 \pm 0.02}$	—	3.9	0	18

Doubling the depth to 8 layers raises accuracy from 47.9% to 73.2%. Deeper and narrower models (16×64, 32×44) are less accurate than 8×92, vary widely across seeds, and return wrong answers (74 in total over the two arms); the recurrent reference returns none.

Second, we give non-recurrent models more parameters and scale the optimizer steps so that each arm uses about the reference's training compute; one arm uses $4\times$ the compute and the full training split (Table 9).

Table 9: Non-recurrent models ($L = 1$) with larger parameter budgets. Training compute is relative to the reference (4×128, $L = 16$, 2K steps). "Full" trains on the full training split instead of 1,000 puzzles. 3 seeds; rows marked * use 2.

Layers × width, steps	Params	Compute	n	Acc. %	Δ (pp)	Calls	Wrong	Timeout
8×128, 16K	1.6M	$1\times$	12,288	99.40 ± 0.41	-0.53 [-0.67, -0.39]	11.1	25	49
16×128, 8K	3.2M	$1\times$	12,288	99.69 ± 0.28	-0.24 [-0.35, -0.13]	7.0	17	21
4×256, 8K	3.2M	$0.95\times$	781	61.45 ± 2.85	-38.41 [-41.82, -35.00]	111	0	300
16×128, 32K, full*	3.2M	$4\times$	8,192	99.78 ± 0.03	-0.15 [-0.26, -0.03]	2.6	8	10
Reference ($L = 16$)	0.80M	$1\times$	30,720	$\mathbf{99.94 \pm 0.02}$	—	3.9	0	18

At matched training compute, deep non-recurrent models come within 0.2–0.5 points of the reference: 99.40% with 8 layers (1.6M parameters) and 99.69% with 16 layers (3.2M). The 16-layer model with $4\times$ the compute and the full training split reaches 99.78%, 0.15 points below the reference, and needs fewer calls per solve (2.6 vs. 3.9). All three deep models return wrong answers (25, 17 and 8), whereas the recurrent reference returns none in 30,720 puzzles. Parameter count alone does not determine accuracy: the 4×256 model has as many parameters as the 16-layer model and solves 61.5%. Depth of computation does, and recurrence provides it with 0.8M parameters, $1\times$ training compute and 1,000 training puzzles.

C.4 Does supervising every internal iteration matter?

The reference applies the loss to the output of each of the 16 internal iterations and averages over iterations; at inference only the final iteration is read out. The ablated arm applies the loss to the final iteration only and is otherwise identical (Table 10). Both arms are evaluated with the 4,096-puzzle cap, so the reference row covers 4,096 puzzles per seed.

Table 10: Supervising all internal iterations vs. only the final one. 2,000 training steps, 3 seeds, 4,096 test puzzles per seed.

Supervision	n	Acc. %	Δ (pp)	Calls	Wrong	Timeout
All 16 iterations (reference)	12,288	$\mathbf{99.93 \pm 0.00}$	—	3.9	0	9
Final iteration only	12,288	99.72 ± 0.04	-0.20 [-0.31, -0.10]	6.2	16	18

Supervising only the final iteration lowers accuracy by 0.20 points, increases calls per solve from 3.9 to 6.2, and produces 16 wrong answers in 12,288 puzzles, against none for the reference.

C.5 Does the auxiliary cross-entropy matter?

We set the weight λ_{ce} of the auxiliary per-cell cross-entropy to 0 or 1 instead of 0.2 (Table 11).

Table 11: Weight λ_{ce} of the auxiliary cross-entropy. 2,000 training steps, 3 seeds.

λ_{ce}	n	Acc. %	Δ (pp)	Calls	Wrong	Timeout
0	395	24.02 ± 1.88	-75.95 [-80.16, -71.73]	375	0	300
0.2 (reference)	30,720	99.94 ± 0.02	—	3.9	0	18
1	23,100	$\mathbf{99.98 \pm 0.01}$	$+0.04$ [$+0.00$, $+0.07$]	1.6	2	2

Without the auxiliary cross-entropy the model solves 24.0% of the puzzles, and all of its failures are timeouts. With $\lambda_{\text{ce}} = 1$ accuracy is 0.04 points above the reference and calls per solve drop from 3.9 to 1.6, but the model returns 2 wrong answers in $23{,}100$ puzzles, against none for the reference. At the main 4K-step budget the advantage disappears: on $12{,}000$ validation puzzles (3 seeds), $\lambda_{\text{ce}} = 0.2$ solves 99.992% with no wrong answers and $\lambda_{\text{ce}} = 1$ solves 99.967% with 2 wrong answers, so we keep $\lambda_{\text{ce}} = 0.2$.

C.6 What enforces soundness?

Two mechanisms are meant to keep the model from returning a wrong answer: the asymmetric elimination loss, which weights false eliminations $w^+/w^- = 8\times$ more than false retentions ($w^+ = 4$, $w^- = 0.5$), and the CLS conflict head. We change the ratio by setting w^+ to 0.5, 1 or 16 with $w^- = 0.5$ fixed ($1\times$, $2\times$, $32\times$). Separately, we remove the conflict head: the model has no CLS token and $\lambda_{\text{cls}} = 0$, so at inference a conflict is detected only when a cell has no candidates left. The reference evaluates every ratio at $\theta_{\text{elim}} = 0.1$, which is close to $1/(1 + w^+/w^-) = 0.11$ for $8\times$. The last three rows re-evaluate the $1\times$, $2\times$ and $32\times$ checkpoints with $\theta_{\text{elim}} = 1/(1 + w^+/w^-)$, the threshold that matches each ratio (Table 12). "Unsound" is the percentage of the model's candidate eliminations that remove the reference digit, averaged over seeds.

Table 12: Soundness mechanisms: elimination-loss asymmetry w^+/w^-, the CLS conflict head, and the elimination threshold θ_{elim}. 2,000 training steps, 3 seeds.

Model	n	Acc. %	Δ (pp)	Calls	Wrong	Timeout	Unsound %
$1\times$ (symmetric)	403	25.55 ± 0.84	-74.44 [-78.70, -70.18]	320	2	298	1.39
$2\times$	2,242	85.66 ± 4.81	-13.29 [-14.70, -11.88]	37.4	0	300	1.35
$8\times$, conflict head (reference)	30,720	99.94 ± 0.02	—	3.9	0	18	1.76
$32\times$	27,140	$\mathbf{99.97 \pm 0.02}$	$+0.02$ [-0.02, $+0.05$]	3.5	0	10	0.14
$8\times$, no conflict head	409	26.63 ± 1.35	-73.35 [-77.63, -69.06]	19.0	300	0	0.12
$1\times$, $\theta_{\text{elim}} = 0.5$	360	16.64 ± 1.82	-83.33 [-87.18, -79.48]	562	11	289	8.89
$2\times$, $\theta_{\text{elim}} = 0.33$	399	24.80 ± 0.99	-75.19 [-79.43, -70.95]	331	0	300	4.98
$32\times$, $\theta_{\text{elim}} = 0.03$	25,740	99.94 ± 0.07	-0.02 [-0.06, $+0.02$]	3.7	12	5	0.01

The conflict head determines whether failures are abstentions or wrong answers. Without it, every failure is a wrong answer: the model returns an incorrect grid on 300 of 409 puzzles and never times out. The loss asymmetry determines accuracy. At $2\times$ accuracy drops to 85.7% and at $1\times$ to 25.6%, with almost all failures being timeouts. Raising the ratio from $8\times$ to $32\times$ does not change accuracy measurably ($+0.02$ points, interval includes 0), returns no wrong answers, and lowers the unsound-elimination rate from 1.76% to 0.14%. Evaluating each checkpoint at the threshold matched to its ratio lowers accuracy for $1\times$ and $2\times$ (to 16.6% and 24.8%) and adds 12 wrong answers for $32\times$. The low accuracy of the $1\times$ and $2\times$ models therefore comes from the trained weights, not from a mismatched inference threshold.

C.7 Do hinted and corrupted training puzzles help?

A puzzle entering the training pool is the raw puzzle (20%), the puzzle with a random share of its blanks filled correctly (55%), or the puzzle with a random share filled and some of those digits wrong (25%). Building these inputs is domain independent: it uses only a solution and the set of values a cell can take, not the rules of the puzzle. We train the 4K-step recipe of Table 1 ($1{,}000$ training puzzles, at most $1{,}000$ search rounds) with two other mixes, 3 seeds each: raw puzzles only, so that every unsatisfiable state in training comes from the model's own branching, and raw puzzles with correct hints only (20% / 80%). The arms are evaluated with the sequential stop rule ($5{,}900$ to $37{,}300$

test puzzles per seed); Table 13 compares each with the Table 1 models on the same puzzles and seeds.

Table 13: Training inputs on Sudoku-Extreme (4K steps, 3 seeds), each arm against the Table 1 models (trained with hinted and corrupted puzzles) on the same test puzzles and seeds: paired accuracy difference with a 95% interval, and wrong answers and timeouts on those puzzles, summed over seeds, as arm / reference.

Training inputs	Puzzles	Δ (pp)	Wrong	Timeouts
Raw and correct hints	43,900	−0.08 [−0.11, −0.05]	33 / 0	7 / 5
Raw only	84,500	−0.13 [−0.15, −0.10]	59 / 0	59 / 11

Without corrupted inputs the model returns more wrong answers: 33 in 43,900 puzzles with correct hints and 59 in 84,500 with raw puzzles only, against none for the reference on the same puzzles. The states the model reaches through its own branching do not give the conflict head enough unsatisfiable examples at this budget. Removing the correct hints as well also raises the number of timeouts.

D Search-Policy Ablations on Sudoku-Extreme

When deduction stops removing candidates, the solver branches: it picks one cell with two or more alive candidates and pins it to one digit. The default policy picks the cell uniformly at random and, at inference, pins the alive candidate of that cell with the highest digit logit (argmax). During training, the pool states are advanced with the same uniform cell choice but a digit sampled from the softmax at $\tau_{\text{decide}} = 1.5$. When a chain detects a conflict, it restarts from the puzzle. This appendix varies three parts of this policy: the cell and digit rules at inference, the digit rule during training, and where a chain restarts after a conflict.

Protocol We use the 2,000-step ablation baseline of Appendix C (the Sudoku-Extreme configuration with per-step training augmentation, stopped at 2,000 steps) and the same configuration stopped at 1,000 steps. The 1,000-step model deduces less, so it branches more often and differences between search policies are larger. Each model is trained with 3 seeds. All arms in Tables 14 and 16 only change the solver and reuse these checkpoints; the two arms in Table 15 that train with argmax digits are separate 2,000-step runs with the same 3 seeds. The solver uses 64 chains per puzzle and at most 1,000 rounds. Evaluation walks the fixed ordering of the test split (Appendix M) for at most 4,096 puzzles per seed and stops early at the 100th failure, so low-accuracy arms are evaluated on fewer puzzles (at least 140 per seed). The default policy on the 2,000-step model and the backtracking arms with exclusion are evaluated on up to 10,240 puzzles per seed, with the same failure rule and the confidence-sequence rule of Appendix M. Accuracy is the mean ± standard deviation over seeds. Δ is the accuracy difference from the default policy on the same model, paired by puzzle and pooled over seeds, with a 95% interval. *Calls* is the number of batched model calls divided by the number of puzzles solved; one call processes 512 chain states (8 puzzles with 64 chains each).

D.1 Cell and digit choice at inference

The cell rules are: *uniform*, a uniformly random cell; *fewest candidates*, a cell with the fewest alive candidates; *lowest entropy* and *highest entropy*, a cell whose softmax over alive candidates has the lowest or highest entropy. Ties break uniformly at random. The digit rules are: *sample*, a softmax sample at $\tau_{\text{decide}} = 1.5$; *argmax*, the highest-scoring alive candidate; *ranked*, where chain k of a puzzle ($k = 1, \ldots, 64$) takes the k-th highest-scoring alive candidate, or the lowest-scoring one if fewer than k remain. Under the ranked rule, most of the 64 chains therefore take the lowest-scoring candidate. Table 14 reports all 12 combinations on both models.

The default policy has the highest accuracy and the fewest calls on both models. Every other combination is less accurate, and each paired difference excludes zero. With a uniform cell, sampling the digit instead of taking the argmax lowers accuracy by 1.3 points ($[1.1, 1.6]$) on the 2,000-step model and by 16.6 points ($[14.9, 18.4]$) on the 1,000-step model, and it uses 3 to 5 times more calls. The fewest-candidates and lowest-entropy cell rules work better with a sampled digit than with argmax. With sampling they come close to the default on the 2,000-step model (-0.2 and -0.7

Table 14: Cell and digit rules at inference on Sudoku-Extreme, for models trained for 2,000 and 1,000 steps (3 seeds each). Uniform cell with argmax digit is the default policy. Calls are batched model calls per solved puzzle.

		2,000 training steps		1,000 training steps	
Cell	**Digit**	**Accuracy %**	**Calls**	**Accuracy %**	**Calls**
Uniform	Argmax (default)	$\mathbf{99.9 \pm 0.0}$	3.9	$\mathbf{99.2 \pm 0.2}$	9.8
Uniform	Sample	98.6 ± 0.8	13.1	82.6 ± 3.3	50.5
Uniform	Ranked	72.6 ± 3.3	80.2	48.0 ± 0.8	166.6
Fewest candidates	Argmax	96.0 ± 0.4	13.9	93.6 ± 0.6	21.1
Fewest candidates	Sample	99.7 ± 0.1	7.4	96.5 ± 1.3	20.3
Fewest candidates	Ranked	60.1 ± 3.0	110.5	45.4 ± 6.1	168.1
Lowest entropy	Argmax	80.1 ± 1.8	43.2	80.7 ± 1.4	45.5
Lowest entropy	Sample	99.2 ± 0.3	10.9	96.9 ± 0.9	20.2
Lowest entropy	Ranked	51.8 ± 1.1	140.5	44.9 ± 1.9	176.4
Highest entropy	Argmax	90.9 ± 0.2	27.0	70.1 ± 1.4	73.4
Highest entropy	Sample	88.9 ± 2.2	38.1	44.7 ± 3.7	173.7
Highest entropy	Ranked	44.7 ± 2.5	162.7	29.4 ± 1.4	303.4

points) and lose 2.5 and 2.2 points on the 1,000-step model. With argmax they lose between 3.9 and 19.7 points. The ranked digit rule is the worst digit rule for every cell rule, and the highest-entropy cell rule is the worst cell rule with sampled or ranked digits. Almost all failures in this table are timeouts: no arm returns more than 6 wrong answers over its 3 seeds.

D.2 Digit choice during training

The training pool advances its states with the same step operator as the solver, so the digit rule also changes which states the model is trained on. Table 15 crosses the training digit rule (sample or argmax) with the inference digit rule, on 2,000-step models with a uniform cell rule throughout.

Table 15: Digit rule during training and at inference on Sudoku-Extreme (2,000 training steps, uniform cell, 3 seeds). Training with a sampled digit and inferring with argmax is the default. Δ is paired against the default; wrong answers and timeouts are summed over seeds; n is the number of puzzles evaluated, summed over seeds.

Training	**Inference**	**Accuracy %**	**Δ (pp)**	**Calls**	**Wrong**	**Timeouts**	n
Sample	Argmax (default)	$\mathbf{99.9 \pm 0.0}$	—	3.9	0	18	30,720
Sample	Sample	98.6 ± 0.8	-1.34 $[-1.55, -1.13]$	13.1	0	174	12,288
Argmax	Argmax	99.7 ± 0.5	-0.19 $[-0.29, -0.09]$	2.4	32	0	12,288
Argmax	Sample	99.3 ± 0.8	-0.60 $[-0.75, -0.45]$	5.4	51	32	12,288

Argmax at inference is better under both training rules. For the model trained with argmax, sampling at inference lowers accuracy by 0.42 points ($[0.25, 0.58]$). Training with argmax lowers the calls per solve from 3.9 to 2.4, but it lowers accuracy by 0.19 points and produces wrong answers: 32 with argmax inference and 51 with sampled inference. All of these wrong answers come from one of the three seeds; the other two seeds return none. The default trains with a sampled digit and uses argmax only at inference.

D.3 Backtracking after a conflict

The default restarts a conflicting chain from the puzzle. The alternatives keep a stack of the chain's last 64 decisions, each with the post-deduction state it was made from, and restore one of these states: *last*, the state before the most recent decision; *geometric*, the state j decisions back with $j \sim \mathrm{Geometric}(0.5)$, $j \geq 1$; *uniform depth*, the state before a uniformly random earlier decision. A chain with no recorded decision restarts from the puzzle. With *exclusion*, the digit chosen at the restored decision is also removed from its cell; if this leaves the cell empty, the chain restarts from the puzzle. We evaluate exclusion on the 2,000-step model only. Table 16 also reports the *unsound*

rate: the percentage of the model's candidate eliminations that remove the solution's digit from a cell, over all chains. Digits removed by exclusion are not counted. Chains that already contain a wrong decision also count, so this rate depends on how much time the search spends below wrong decisions as well as on the model.

Table 16: Restart target after a conflict on Sudoku-Extreme (uniform cell, argmax digit, 3 seeds). Restarting from the puzzle is the default. Δ is paired against the default on the same model; wrong answers and timeouts are summed over seeds; n is the number of puzzles evaluated, summed over seeds; Unsound is the percentage of model eliminations that remove the solution's digit.

Restore	**Accuracy %**	**Δ (pp)**	**Calls**	**Wrong**	**Timeouts**	n	**Unsound %**
2,000 *training steps*							
Puzzle (default)	$\mathbf{99.9 \pm 0.0}$	—	3.9	**0**	18	30,720	1.76
Last	79.5 ± 5.3	-19.56 [-21.54, -17.57]	36.9	11	289	1,529	16.67
Geometric	98.3 ± 1.5	-1.49 [-1.72, -1.26]	9.0	6	168	11,102	11.10
Uniform depth	99.8 ± 0.0	-0.11 [-0.20, -0.02]	4.0	1	22	12,288	5.47
2,000 *training steps, with exclusion*							
Last	99.0 ± 1.2	-0.57 [-0.67, -0.47]	7.3	3	150	24,700	7.56
Geometric	99.9 ± 0.1	-0.06 [-0.11, -0.02]	3.9	4	30	28,440	5.26
Uniform depth	99.9 ± 0.0	0.00 [-0.04, 0.04]	3.1	1	14	28,440	3.87
1,000 *training steps*							
Puzzle (default)	99.2 ± 0.2	—	9.8	5	94	12,288	3.90
Last	72.4 ± 4.6	-26.78 [-29.40, -24.16]	57.9	31	269	1,109	21.92
Geometric	97.1 ± 1.0	-1.97 [-2.31, -1.63]	14.5	40	231	9,849	18.13
Uniform depth	$\mathbf{99.4 \pm 0.3}$	$+0.20$ [$+0.01$, $+0.39$]	8.2	10	64	12,288	13.30

Restoring the last decision without exclusion is the worst choice on both models (-19.6 and -26.8 points). Without exclusion, a chain restored to an earlier state can repeat the same decision. Restoring an earlier state at random works better, and uniform depth is the best of the three targets. On the 2,000-step model no backtracking arm is more accurate than the default. Uniform depth with exclusion has the same accuracy (0.00 points, $[-0.04, 0.04]$) and uses 3.1 instead of 3.9 calls per solve. On the 1,000-step model, uniform depth gains 0.20 points ($[0.01, 0.39]$) and uses 8.2 instead of 9.8 calls, but it returns 10 wrong answers instead of 5. Every backtracking arm has a higher unsound rate than the default: 3.87% to 16.67% against 1.76% on the 2,000-step model, and 13.30% to 21.92% against 3.90% on the 1,000-step model. Because the gains in accuracy and calls are small and the unsound rate and wrong answers increase, the default policy restarts from the puzzle.

Search that records information from conflicts, such as stack-based replay or tracking dead ends so that they are not revisited, is left to future work.

E Inference-Time Deduction Ablations

This section varies the step operator at evaluation time only. The trained weights are fixed. We vary four settings of the operator: the number of internal iterations of the recurrent transformer, the number of deduction passes before each decision, the elimination threshold θ_{elim}, and the conflict threshold $\theta_{\text{CLS}}^{\text{eval}}$. Each arm changes exactly one of these settings and keeps the others at their defaults.

Checkpoints All arms evaluate the same three checkpoints. They are trained with the Sudoku configuration of Section 4 for 2,000 steps instead of 4,000, with per-step augmentation in the training rollouts, on the same 1,000 training puzzles. The three checkpoints differ only in the training seed (0, 1, 2). Every arm uses one evaluation seed per checkpoint.

Default operator The reference arm uses the inference settings of the main experiments: 16 internal iterations (the trained value), one deduction pass per step, $\theta_{\text{elim}} = 0.1$, $\theta_{\text{CLS}}^{\text{eval}} = 0.6$, a random symmetry per step, a uniformly random branching cell, the highest-confidence candidate at that cell, 64 parallel chains per puzzle, restart from the original puzzle after a conflict, and at most 1,000 steps per puzzle.

Test puzzles Every arm walks the Sudoku-Extreme test split in the same fixed random order, so all arms see the same puzzles in the same positions. A run stops at the first of three conditions:

100 failures (wrong answers plus timeouts), a 95% confidence-sequence half-width of at most 0.1 percentage points (checked every 100 puzzles from 1,000 on), or a maximum number of puzzles. The maximum is 4,096 for the deduction-pass and elimination-threshold arms and 10,240 for the default, internal-iteration and conflict-threshold arms. The n column of each table gives the number of puzzles per seed.

Metrics Accuracy is the mean ± standard deviation over the three seeds. Δ is the accuracy difference from the default operator, paired by puzzle on the same seed, pooled over seeds, with a 95% interval. It uses only the puzzles both runs evaluated, so for a run stopped by the failure count it covers that run's first n puzzles. Calls per solve is the number of batched model calls (one call advances all 64 chains of a puzzle by one forward pass) divided by the number of solved puzzles. A pass of the operator is one call, so an arm with more passes per step spends more calls per step. Wrong counts accepted solutions that differ from the ground truth; timeouts count puzzles not solved within 1,000 steps. Both are summed over seeds. Unsound is the fraction of eliminated candidates that belong to the puzzle's solution, pooled over all chains and steps of a run and averaged over seeds.

E.1 Internal iterations at evaluation

The recurrent transformer applies the same weights 16 times per call during training. Because the weights are shared, the number of iterations can be set to a different value at evaluation. We evaluate 1, 2, 4, 8 and 32 iterations. Table 17 gives the results.

Table 17: Number of internal iterations at evaluation, for checkpoints trained with 16. Three seeds; n is the number of test puzzles per seed. Δ is in percentage points against the 16-iteration default, with a 95% interval.

Iterations	n	Acc. %	Δ (pp)	Calls/solve	Wrong	Timeouts	Unsound %
1	104	3.8 ± 0.0	-96.15 [-98.29, -94.02]	5,115	0	300	0.33
2	124–131	21.4 ± 2.2	-78.53 [-82.65, -74.42]	469	0	300	0.81
4	749–1,474	90.3 ± 3.4	-8.91 [-9.88, -7.94]	38.0	1	299	1.79
8	10,240	99.5 ± 0.2	-0.48 [-0.56, -0.39]	8.8	0	165	1.87
16 (default)	10,240	$\mathbf{99.9 \pm 0.0}$	ref.	3.9	**0**	18	1.76
32	10,240	99.9 ± 0.0	-0.05 [-0.09, -0.01]	3.0	1	32	1.97

Accuracy increases with the number of iterations up to 16. With 1 or 2 iterations the solver times out on most puzzles, and the runs stop after about 100 to 130 puzzles. With 4 iterations accuracy is 90.3% and a solve takes 38.0 calls. With 8 iterations accuracy is 0.48 points below the default and a solve takes 8.8 calls instead of 3.9. With 32 iterations a solve takes 3.0 calls, but each call runs twice as many iterations as the default, and accuracy is 0.05 points lower (interval $[-0.09, -0.01]$). The unsound rate is between 1.76% and 1.97% from 4 to 32 iterations.

E.2 Deduction passes before each decision

By default, a step runs the model once, eliminates candidates, and then branches. A multi-pass step runs the model again on the reduced state before it branches, with a new random symmetry for each pass. Each pass is one model call. A chain stops making passes once it is solved or in conflict. We test 2 and 4 passes, and repeating passes until a pass eliminates nothing (at most 16 passes). Table 18 gives the results.

Table 18: Deduction passes before each decision. Three seeds; n is the number of test puzzles per seed. Δ is in percentage points against the one-pass default, paired on the first 4,096 puzzles, with a 95% interval.

Passes	n	Acc. %	Δ (pp)	Calls/solve	Wrong	Timeouts	Unsound %
1 (default)	10,240	$\mathbf{99.9 \pm 0.0}$	ref.	**3.9**	0	18	1.76
2	4,096	99.9 ± 0.1	-0.03 [-0.10, $+0.04$]	7.7	0	13	2.05
4	4,096	99.9 ± 0.1	-0.05 [-0.12, $+0.03$]	13.3	0	15	2.28
Until no change (≤ 16)	4,096	99.7 ± 0.1	-0.24 [-0.35, -0.14]	54.6	0	39	2.87

Two and four passes do not change accuracy measurably: both intervals contain zero. They raise calls per solve from 3.9 to 7.7 and 13.3. Repeating until no change lowers accuracy by 0.24 points and takes 54.6 calls per solve. The unsound rate rises with the number of passes, from 1.76% with one pass to 2.87% when repeating until no change, because each pass makes further eliminations on the output of the previous pass. One pass per step has the lowest cost and the lowest unsound rate, with no measurable loss in accuracy.

E.3 Elimination and conflict thresholds

The elimination threshold θ_{elim} removes every candidate whose predicted probability is below it. The default is 0.1; we test 0.05, 0.2, 0.3 and 0.5. The conflict threshold $\theta_{\text{CLS}}^{\text{eval}}$ declares a state inconsistent when the conflict head's probability is above it. The default at evaluation is 0.6; we test 0.5 (the value used in training rollouts), 0.7 and 0.8. Table 19 gives the results.

Table 19: Elimination threshold θ_{elim} and conflict threshold $\theta_{\text{CLS}}^{\text{eval}}$ at evaluation. Three seeds; n is the number of test puzzles per seed. Δ is in percentage points against the default ($\theta_{\text{elim}} = 0.1$, $\theta_{\text{CLS}}^{\text{eval}} = 0.6$), with a 95% interval.

Threshold	n	Acc. %	Δ (pp)	Calls/solve	Wrong	Timeouts	Unsound %
Default	10,240	$\mathbf{99.9 \pm 0.0}$	ref.	**3.9**	0	18	1.76
$\theta_{\text{elim}} = 0.05$	4,096	99.9 ± 0.1	-0.05 [-0.13, $+0.03$]	4.8	0	15	0.67
$\theta_{\text{elim}} = 0.2$	4,096	99.7 ± 0.1	-0.27 [-0.38, -0.16]	4.1	3	39	3.53
$\theta_{\text{elim}} = 0.3$	4,096	98.4 ± 0.3	-1.57 [-1.80, -1.35]	6.7	9	193	4.89
$\theta_{\text{elim}} = 0.5$	617–762	85.6 ± 1.6	-14.18 [-15.68, -12.69]	26.0	44	256	6.21
$\theta_{\text{CLS}}^{\text{eval}} = 0.5$	320–447	72.7 ± 4.5	-26.81 [-29.41, -24.21]	75.7	0	300	0.09
$\theta_{\text{CLS}}^{\text{eval}} = 0.7$	10,240	99.9 ± 0.1	-0.04 [-0.08, $+0.01$]	3.7	5	23	2.21
$\theta_{\text{CLS}}^{\text{eval}} = 0.8$	8,600–10,240	99.8 ± 0.3	-0.15 [-0.20, -0.09]	3.9	36	24	2.54

A lower elimination threshold of 0.05 removes fewer candidates. It lowers the unsound rate from 1.76% to 0.67% and raises calls per solve from 3.9 to 4.8, with no measurable change in accuracy. Higher thresholds remove more candidates, including correct ones. The unsound rate rises to 3.53%, 4.89% and 6.21% at 0.2, 0.3 and 0.5, and accuracy falls by 0.27, 1.57 and 14.18 points. These are also the only elimination thresholds that produce wrong answers (3, 9 and 44 over three seeds).

At $\theta_{\text{CLS}}^{\text{eval}} = 0.5$ the conflict head stops too many chains that have not reached a contradiction. Accuracy falls to 72.7%, all 300 failures are timeouts, and a solve takes 75.7 calls. At 0.7, calls per solve fall from 3.9 to 3.7 and accuracy does not change measurably, but the solver returns 5 wrong answers, all on the seed-0 checkpoint. At 0.8, it returns 36 wrong answers, also all on the seed-0 checkpoint, and accuracy falls by 0.15 points. A higher threshold flags fewer states as inconsistent, so fewer chains are stopped before they reach a complete grid. The default of 0.6 is the highest tested value that returns no wrong answers.

E.4 Final-iteration and averaged logits

The step operator reads the logits of the last internal iteration. We also evaluate the same checkpoints with the logits of each head averaged over the 16 iterations (3 seeds, 10,240 test puzzles per seed).

Table 20: Logits read by the step operator. Δ is in percentage points against the default, paired by puzzle, with a 95% interval.

Logits	Acc. %	Δ (pp)	Calls/solve	Wrong	Timeouts	Unsound %
Final iteration (default)	$\mathbf{99.94 \pm 0.01}$	ref.	3.9	0	18	1.76
Mean over iterations	99.79 ± 0.07	-0.15 [-0.20, -0.09]	6.2	0	64	1.18

Averaging lowers accuracy by 0.15 points, needs 6.2 instead of 3.9 calls per solve and times out on 64 instead of 18 puzzles; neither returns a wrong answer. Its share of eliminations that remove a correct candidate is lower (1.18% against 1.76%).

F Carrying Continuous State Between Solve Steps

LDT passes only the lattice state from one solve step to the next. The transformer's hidden activations are discarded after every step. Recurrent reasoners such as TRM instead pass a continuous latent state between steps. This appendix tests whether adding such a state to LDT changes its accuracy.

Models We compare four models. All of them apply the 4-layer backbone 16 times per model call.

- *Lattice only*: the standard one-stream LDT. Nothing except the lattice is passed between steps.
- *Carry hidden state*: the one-stream model, with the final hidden state of the previous step passed through a learned linear map and added to the input embedding of the next step.
- *Carry scratchpad*: a two-stream model. Each model call runs 8 cycles. In each cycle, one backbone application updates a scratchpad stream from the embedding of the original puzzle, the answer stream and the scratchpad, and a second backbone application updates the answer stream from the answer stream and the scratchpad. Only the answer stream is read out. The final scratchpad of a step is passed through a learned linear map and becomes the initial scratchpad of the next step.
- *Zeroed scratchpad*: the same two-stream model, with the scratchpad set to zero at the start of every step. This control separates the effect of carrying the scratchpad from the effect of the two-stream architecture.

The linear maps are initialized to zero. The carried state is reset to zero when a puzzle enters the training pool and when a solve chain restarts from the original puzzle after a conflict. During training, the carried state is computed when the pool is advanced, without gradients, so no gradient crosses a solve-step boundary. The two-stream models have 8 supervised readouts per model call; the one-stream models have 16.

Setup All four models use the ablation protocol of Appendix C: the hyperparameters of Appendix L.1, 2,000 training steps on the 1,000 training puzzles, and seeds 0, 1 and 2. A carried vector is only meaningful if consecutive steps use the same digit labels and grid orientation. We therefore turn off per-step augmentation, in training and at inference, for all four models, including the lattice-only model. The augmentation applied to each puzzle when it enters the training pool is unchanged. Inference uses the defaults of Appendix L.1 (64 chains per puzzle, 1,000 rounds) without per-step augmentation.

Evaluation Every model and seed walks the same fixed ordering of the Sudoku-Extreme test split (Appendix M). Evaluation stops after 4,096 puzzles or at the 100th failure, whichever comes first. The lattice-only model reaches 4,096 puzzles for every seed. The other three models reach 100 failures first, so their evaluations cover fewer puzzles; Table 21 lists the range per seed. Because the models see different numbers of puzzles, we also report the difference in accuracy from the lattice-only model on the same puzzles: for each seed we compare the two models on the puzzles that both evaluated, and we pool these paired outcomes over seeds.

Table 21: Carrying continuous state between Sudoku-Extreme solve steps (2,000 training steps, no per-step augmentation, 3 seeds). n/seed is the range over seeds of the number of test puzzles evaluated. Accuracy is the mean $\pm$ standard deviation over seeds. Δ is the difference in accuracy from the lattice-only model on the same puzzles, pooled over seeds, with a 95% interval. "Calls/solve" is the number of batched model calls per solved puzzle, averaged over seeds. "Wrong" is the total number of wrong answers over seeds; all other failures are timeouts. "Unsound" is the percentage of eliminated candidates that are the solution digit, averaged over seeds.

State carried	n/seed	Acc. %	Δ (pp)	Calls/solve	Wrong	Unsound %
Lattice only	4,096	$\mathbf{98.4 \pm 0.1}$	—	8.3	1	1.32
Hidden state	262–547	71.6 ± 9.9	-24.7 [-27.3, -22.2]	74.9	0	0.23
Scratchpad	157–192	43.3 ± 6.2	-55.2 [-59.4, -50.9]	182.5	1	0.14
Zeroed scratchpad	425–1,198	85.8 ± 8.2	-10.4 [-11.7, -9.1]	14.3	231	3.93

Results Both models that carry continuous state are less accurate than the lattice-only model. Carrying the hidden state lowers accuracy by 24.7 points on the same puzzles, and carrying the scratchpad lowers it by 55.2 points. Both models also use more model calls per solved puzzle: $9\times$ and $22\times$ as many as the lattice-only model. Almost all of their failures are timeouts (599 of 600), not wrong answers. Their unsound-elimination rates are lower than the lattice-only model's.

The zeroed-scratchpad control shows that the two-stream architecture alone already reduces accuracy, by 10.4 points, and that it returns wrong answers: 231 of the $2{,}552$ puzzles it evaluated, against 1 of $12{,}288$ for the lattice-only model. Carrying the scratchpad reduces accuracy further: on the puzzles both evaluated, the carry scratchpad model is 42.6 points less accurate than the zeroed-scratchpad model (95% interval $[-47.0, -38.2]$).

In this training and search setup, the lattice is a better state to pass between solve steps than the final hidden activations or a dedicated scratchpad. The result applies to the two designs tested. It does not test a scratchpad with the same number of supervised readouts as the baseline, training with gradients through several solve steps, or a continuous state that is freely revised as in TRM. LDT's lattice can only lose candidates from step to step, and a continuous state would need to be designed to work with that constraint.

G Maze-Hard Failure Analysis

We examine every unsolved maze of the checkpoints reported in Table 3: each seed's best checkpoint, 3 seeds each at $K = 1$ and $K = 512$, $1{,}000$ test mazes per run. Of the 36 failures, 3 are timeouts and 33 are wrong answers, on 20 distinct mazes. The same mazes recur across runs: maze 299 is answered wrongly by 4 of the 6 runs, and mazes 640, 706 and 892 by 3.

Almost every wrong answer is a path from start to goal that is longer than the shortest path (counts summed over 3 seeds per setting):

Failure	$K = 1$	$K = 512$	**Total**
2 steps too long	18	4	22
4–10 steps too long	6	2	8
16 steps too long	1	0	1
shortest path plus extra marked cells	1	0	1
does not reach the goal	1	0	1
wrong answers	27	6	33
timeouts	3	0	3

Most of each wrong path lies on some shortest path (65 to 110 of its cells, typically about 90). Figure 12 shows two examples from each setting, with the shortest path that shares the most cells with the model's answer. Whether a path is shortest depends on the whole maze, not on any single cell, and the conflict head does not flag these answers.

H Snowflake Sudoku: Order Transfer and Training Set Size

A single model handles every Snowflake order through the shared covering grid, so we can test whether it solves larger puzzles than it was trained on. We train on hard puzzles of orders 4 to 6 (all $1{,}308$ training puzzles of these orders) and evaluate on orders 7 to 10 (768 test puzzles), and train on orders 6 to 8 ($2{,}000$ puzzles) and evaluate on orders 9 and 10 (445 test puzzles). All other settings follow the main Snowflake recipe (Section 4.2). We compare LDT with search, the same LDT model in one shot (one model call, the highest-scoring digit in every cell), and direct prediction (the same model trained only with cross-entropy against the solution, evaluated in one shot).

Table 22 gives the results by order. Trained on orders 6 to 8, LDT transfers to orders 9 and 10 almost without loss: it solves 99.4% with 0.2% wrong answers, and 98.5% of the puzzles that need at least 8 guesses. One shot and direct prediction solve 78.9% and 78.4%, and 63.2% and 67.4% of the puzzles that need at least 8 guesses, so the training objective alone does not explain the difference; search does. Trained on orders 4 to 6, whose puzzles have at most 36 cells against 60 at order 10, every method degrades with distance from the training orders. LDT stays close to sound one order above the training range (0.7% wrong at order 7), but at order 10 it solves 50.0% and gives 30.9% wrong

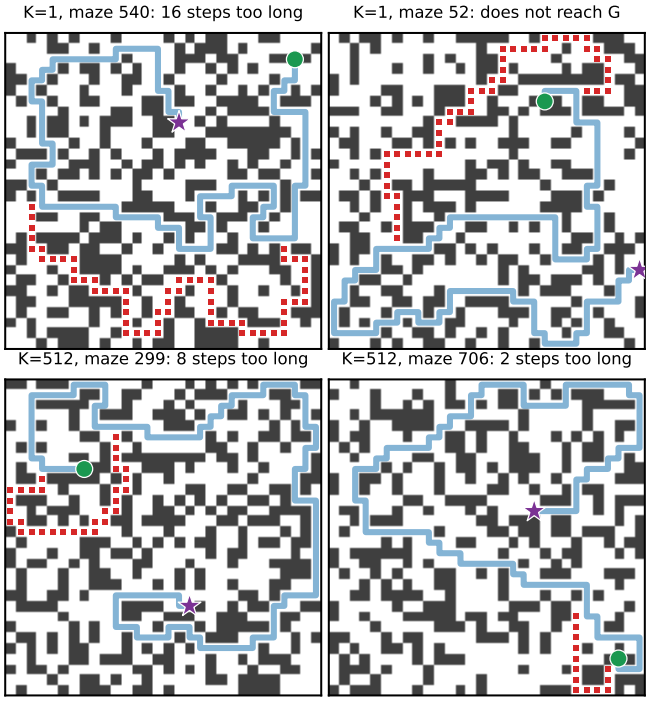


Figure 12: Wrong Maze-Hard answers, two each at $K = 1$ and $K = 512$. Blue: the true shortest path sharing the most cells with the model's answer; red: the answer's cells off that path. Green circle: start; purple star: goal.

Table 22: Snowflake Sudoku order transfer (hard puzzles): accuracy / wrong answers (%) on test puzzles of orders not seen in training, mean over 3 seeds. The remainder are timeouts; one shot and direct prediction always answer.

Train	Method	7	8	9	10	All
4–6	LDT	99.27 / 0.73	93.91 / 6.09	78.36 / 17.38	50.00 / 30.91	77.21 / 15.80
	LDT, one shot	91.24 / 8.76	75.63 / 24.37	46.60 / 53.40	21.65 / 78.35	53.99 / 46.01
	Direct prediction	81.27 / 18.73	51.79 / 48.21	16.59 / 83.41	3.28 / 96.72	32.60 / 67.40
6–8	LDT	—	—	100.00 / 0.00	98.86 / 0.43	99.40 / 0.22
	LDT, one shot	—	—	85.94 / 14.06	72.51 / 27.49	78.88 / 21.12
	Direct prediction	—	—	85.78 / 14.22	71.79 / 28.21	78.43 / 21.57

answers; search then finds assignments that the model accepts but that are wrong. Direct prediction solves 3.3% of order-10 puzzles.

Table 23: Snowflake Sudoku training set size: accuracy / wrong answers (%) on hard test puzzles of orders 4 to 8, trained on hard puzzles of the same orders, mean over 3 seeds. Training uses 8K steps unless noted.

Training puzzles	LDT	LDT, one shot	Direct prediction
1,000	100.00 / 0.00	95.52 / 4.48	88.87 / 11.13
2,000	100.00 / 0.00	96.77 / 3.23	97.63 / 2.37
2,000, 16K steps	100.00 / 0.00	98.42 / 1.58	96.05 / 3.95
all 3,957	100.00 / 0.00	98.29 / 1.71	99.28 / 0.72

In distribution (Table 23), LDT solves every hard test puzzle with as few as 1,000 training puzzles. Direct prediction needs more data: 88.9% with 1,000 puzzles, 97.6% with 2,000 and 99.3% with all 3,957. Training for 16K steps instead of 8K improves one shot but not direct prediction.

I Fine-Tuned LLM Baseline on Sudoku-Extreme

The LLMs in Appendix J are evaluated zero-shot. This appendix measures how a language model performs when it is trained on Sudoku-Extreme puzzles.

Model and training We fine-tune all parameters of Qwen3.5-0.8B, about $1{,}000\times$ the parameter count of LDT, in BF16. The training set is $1{,}000$ puzzles drawn at random (seed 42) from the Sudoku-Extreme training split. LDT's $1{,}000$ training puzzles are drawn with the same seed after a $10{,}000$-puzzle validation set is removed from the split, so the two training sets share 16 puzzles. There is no data augmentation. The prompt gives the puzzle as an 81-character row-major string with `0` for blank cells and asks for the solved 81-digit string with no other text. The target is the 81-digit solution, with no reasoning trace, and the loss is computed on the answer tokens only. Training runs for 16 epochs (512 steps) with AdamW, learning rate 2×10^{-5}, weight decay 0.1, batch size 32, a cosine schedule and a linear warmup over the first 10% of steps. Each run uses one seed (0).

Puzzle difficulty We train seven models. One is trained on the $1{,}000$ natural puzzles. The other six are trained on versions of the same $1{,}000$ puzzles with exactly K blank cells, for $K \in \{1, 2, 4, 8, 16, 32\}$. Each such version is made from the puzzle's solution by blanking K cells chosen uniformly at random; the chosen cells are fixed for the whole training run and are not limited to the cells that are blank in the natural puzzle. Natural Sudoku-Extreme puzzles have about 56 blank cells and are selected to require long chains of deduction. Puzzles with randomly placed blanks are much easier at the same blank count.

Evaluation Each model is evaluated on the first 256 puzzles of the fixed test ordering of Appendix M. These puzzles have 45 to 64 blank cells (55.8 on average). The model trained on K blanks is evaluated on the same 256 puzzles rebuilt with K randomly placed blanks. We draw 32 samples per puzzle (temperature 0.8, top-p 0.95, at most 128 new tokens). A sample is correct if, after removing surrounding whitespace, it is exactly the 81-digit solution. With c correct samples out of $n = 32$, pass@k for a puzzle is $1 - \binom{n-c}{k} / \binom{n}{k}$, the probability that at least one of k samples drawn without replacement is correct; we average it over the 256 puzzles. Pass@32 is the fraction of puzzles with at least one correct sample. It assumes that the correct sample can be identified, which the model itself does not do. We evaluate every second epoch and report the final checkpoint (epoch 16).

Table 24: Fine-tuned Qwen3.5-0.8B on Sudoku-Extreme: pass@k (%) at epoch 16 on the first 256 test puzzles, 32 samples per puzzle, 1 seed. Each row is a separate model, trained on $1{,}000$ training puzzles and evaluated on test puzzles of the same kind. In the first six rows, the puzzles are rebuilt from their solutions with exactly K randomly placed blank cells. The last row uses the natural puzzles (45–64 blanks in the evaluated set).

Blank cells	pass@1	pass@8	pass@32
$K = 1$ (random)	98.6	99.5	99.6
$K = 2$ (random)	97.7	98.7	99.2
$K = 4$ (random)	84.1	90.4	91.8
$K = 8$ (random)	78.6	87.7	89.8
$K = 16$ (random)	55.9	68.6	71.9
$K = 32$ (random)	8.7	14.4	16.4
Natural puzzles	0.0	0.0	0.0

Results With 1 or 2 random blanks the fine-tuned model solves almost every puzzle (99.6% and 99.2% pass@32), so the training and evaluation pipeline works. Accuracy falls as blanks are added: pass@32 is 91.8% at 4 blanks, 71.9% at 16 and 16.4% at 32. On the natural puzzles the model solves none of the 256 puzzles at epoch 16. Over all eight evaluated checkpoints (epochs $2, 4, \ldots, 16$), it solves one puzzle in one checkpoint (epoch 10) and none in the others. Its output format also degrades with training: the share of samples that are not a single 81-digit string falls from 32% at epoch 2 to 0.6% at epoch 6, then rises to 55% at epoch 16.

For comparison, LDT trained on $1{,}000$ natural puzzles without augmentation solves 96.54% of natural test puzzles, with one answer per puzzle (Table 1).

J Frontier LLM Baselines

We evaluate frontier chain-of-thought LLMs zero-shot with the prompts below.

April 2026 Claude Opus 4.6 [40], DeepSeek V4-Pro [41] and GPT-5.4 [42] were each evaluated on 200 test puzzles per benchmark and solved none of them. Their Snowflake puzzles come from an earlier version of the generator (`https://github.com/santolucito/snowflakesudoku`), with one fixed layout per order and including puzzles with givens added back, not the puzzles of Section 4.2.

Model (April 2026)	Sudoku-Extreme	Snowflake	Maze-Hard
Claude Opus 4.6	0.0	0.0	0.0
DeepSeek V4-Pro	0.0	0.0	0.0
GPT-5.4	0.0	0.0	0.0

Solve rate (%) on 200 test puzzles per benchmark.

October 2026 We evaluate Claude Opus 5.5, GPT-6 Astra and GPT-6.1 Sol through OpenRouter on the first 50 puzzles of each benchmark's test ordering: Sudoku-Extreme, the hard Snowflake puzzles of Section 4.2 and Maze-Hard. Each call uses reasoning effort "high", at most 100,000 output tokens and at most one hour; an answer that reaches a cap, or a call that ends in a provider error, counts as unsolved. Sudoku and Snowflake answers must equal the unique solution; a Maze answer must be a valid 30×30 grid with walls, start and goal unchanged and a shortest path from start to goal. Table 25 reports the solve rate and the mean output tokens and cost per puzzle.

Table 25: Frontier LLMs in October 2026, zero-shot, on the first 50 test puzzles of each benchmark: solve rate (%), mean output tokens per puzzle and mean cost per puzzle (US dollars, OpenRouter prices).

	Sudoku-Extreme			Snowflake (hard)			Maze-Hard		
Model	**Solved %**	**Tokens**	**$**	**Solved %**	**Tokens**	**$**	**Solved %**	**Tokens**	**$**
Claude Opus 5.5	54	50,794	1.02	100	3,797	0.08	26	14,319	0.29
GPT-6 Astra	68	28,336	1.42	100	2,303	0.13	26	11,906	0.60
GPT-6.1 Sol	44	18,797	0.19	96	2,447	0.03	12	11,457	0.12

Observed failures Hard Snowflake puzzles are solved almost always, in under a minute and a few thousand output tokens; they are much easier for these models than Sudoku-Extreme. On Sudoku-Extreme, many failures return no answer: Claude Opus 5.5 reaches the output-token cap on 14 puzzles, and GPT-6 Astra and GPT-6.1 Sol end in a provider error on 6 and 11 puzzles, after up to 52 minutes. Of the answers returned, most wrong ones break a row, column or box constraint (12 for GPT-6.1 Sol, 4 for GPT-6 Astra, 3 for Claude Opus 5.5); a few change a given or are valid grids that differ from the solution. On Maze-Hard, the most common failure is a path from start to goal that is longer than the shortest path (14 to 21 per model). The next most common is a malformed grid (16 to 18 per model), almost always a row of 29 characters instead of 30. A few answers do not connect start and goal or pass through a wall.

J.1 Sudoku-Extreme

System prompt

```
You are a Sudoku solver. You will be given a 9 × 9 Sudoku puzzle.
Solve it and respond ONLY with the completed 9 × 9 grid, one row per
line, digits only, no spaces or other characters. Example response
format:

534678912
672195348
...
345286179
```

User prompt

```
Solve this Sudoku:
{puzzle_text}
({puzzle_text} is the puzzle as 9 lines of 9 characters each, with
empty cells written as 0.)
```

J.2 Snowflake Sudoku

System prompt

```
You are solving a Snowflake Sudoku puzzle, a Sudoku variant played
on a snowflake-shaped topology of N cells, each holding a digit
from 1 to 6. The puzzle is defined by a list of constraint groups;
each group is a set of cell IDs whose digits must all be distinct.
You will be given (1) the total number of cells, (2) the list of
constraint groups, and (3) the current cell values, where ? marks
a cell you must solve and an integer 1-6 marks a given. Respond
with ONLY the solution: one line per cell in order, formatted
as CELL_ID:DIGIT with no spaces, comments, or other text. Output
exactly N lines. Example:

0:3
1:5
2:1
3:6
4:2
5:4
```

User prompt

```
Solve this Snowflake Sudoku:
{puzzle_text}
({puzzle_text} encodes the puzzle size, the constraint topology, and
the current cell values, e.g.:)

Puzzle (n=4, 24 cells):
Constraint groups (each group must contain distinct digits):
   Group 0:  [0, 1, 2, 3, 4, 5]
   Group 1:  [6, 7, 8, 9, 10, 11]
   Group 2:  [12, 13, 14, 15, 16, 17]
   ...
Cell values (?  = solve this):
   Cell 0:  3
   Cell 1:  ?
   Cell 2:  ?
   Cell 3:  6
   ...
```

J.3 Maze-Hard

System prompt

```
You are a maze solver. You will be given a 30 × 30 maze. Every cell
of the maze is exactly one of the following four characters:

'#' - wall (impassable)
' ' - free cell
'S' - the start cell (exactly one)
'G' - the goal cell (exactly one)

Find a path of orthogonally adjacent (up/down/left/right) free cells
that connects S to G without passing through any wall. Diagonal
moves are not allowed. Mark every free cell that lies on the path
with the character o. The cells #, S, and G are unchanged, and free
```

```
cells that are not on the path remain ' '.  Respond with ONLY the
completed 30  ×  30 grid:  exactly 30 lines, each containing exactly
30 characters drawn from the set {#, ' ', S, G, o}.  Do not include
row numbers, column markers, blank lines, code fences, prose, or any
other text.  Whitespace inside each row is significant:  a leading
or trailing space is itself a free cell, not formatting.  Example
response format (shown for a 6 × 6 maze for brevity; your output must
be 30 rows of 30 characters):

######
#Sooo#
#  #o#
#  #o#
## ooG
######
```

User prompt

```
Solve this maze:
{puzzle_text}
({puzzle_text} is the maze as 30 lines of 30 characters each, drawn
from {#, ' ', S, G}.)
```

K Snowflake Sudoku: Data and Encoding

We generate puzzles using CVC5 [43]. Every puzzle gets its own randomly grown, connected layout of order n; CVC5 finds a valid complete assignment, and a greedy minimization pass removes givens while CVC5 verifies that the puzzle retains a unique solution, so the puzzle is minimal. We rate each puzzle by the number of guesses a simple solver needs: it applies naked and hidden singles until nothing changes, then branches on a cell with the fewest candidates. A rating of 0 means singles alone solve the puzzle, which holds for 41% (order 8) to 96% (order 4) of minimal puzzles. We keep only puzzles with rating at least 1; the median test puzzle needs 3 guesses. Puzzles of orders 4 to 10 are split into test (10%), validation (5%) and training sets, giving $7{,}801$ training, 476 validation and 951 test puzzles. Small orders are rare because singles solve almost all of them: orders 4 to 8 have $3{,}957$ training and 506 test puzzles, of which 19 are of order 4. Unless noted, we train on $2{,}000$ puzzles of orders 4 to 8, and orders 9 and 10 are held out for the order-transfer experiments in Appendix H.

To handle the variable topology with a fixed-size transformer, we embed every puzzle into a fixed covering grid of 25 hexagon positions (5×5 in axial coordinates, 150 cells); layouts that do not fit are regrown. Each hexagonal cell maps to a deterministic position via axial coordinates (q, r) and triangular direction, with cells absent from a given puzzle zeroed out and excluded from the loss via a per-cell mask; during training each puzzle is translated by a random whole number of hexes within the grid. Because every layout differs, a cell's position no longer determines its constraint groups, so each cell also receives a multi-hot indicator of the groups it belongs to. Group identifiers are reassigned at random for every sample, so the model can use shared membership but cannot tie an identifier to a location. A single model thus handles all puzzle sizes without architecture changes. Training and inference follow the same solve procedure as Sudoku-Extreme, with 8K training steps; unlike Sudoku-Extreme, training also uses per-step augmentation and the random hex translation (Appendix L.2). Each snowflake puzzle has a unique solution, so α runs at $K = 1$. Full hyperparameters are in Appendix L.2. TRM, HRM, and Sotaku are not trained on snowflake Sudoku and are omitted.

L Hyperparameters

Augmentation Per-step augmentation wraps each solve step in a random symmetry of the problem and inverts it after deduction and branching: digit permutation composed with a dihedral grid symmetry for Sudoku, digit permutation for Snowflake (the dihedral group does not preserve the hex topology), and dihedral symmetries for Maze. All three benchmarks use it at evaluation; Snowflake and Maze also use it in training, while for Sudoku it did not help in training. Sudoku instead augments

at the dataset level, Snowflake translates each puzzle by a random whole number of hexes within the covering grid, and Maze applies a random dihedral symmetry and a random swap of start and goal to each training sample. The models in Figure 4(a, b) were trained with per-step augmentation, unlike the Sudoku rows of Table 1; panel (c) uses the portfolio models, which are trained as in Table 1.

L.1 Sudoku-Extreme

These settings are used for Sudoku-Extreme (Section 4.1) and are the base for the other benchmarks.

Architecture Looped transformer with embedding dimension $d = 128$, 4 layers, 4 attention heads, $L = 16$ internal loops per forward pass, FFN multiplier 4.0, dropout $p = 0.1$ during training, and learned 2D positional embeddings. Total parameter count is $\approx 800{,}000$. Sudoku-Extreme uses 9 input channels per cell, one candidate sigmoid per digit.

Optimizer AdamW with learning rate 3×10^{-3}, weight decay 0.1, $\beta = (0.9, 0.95)$, gradient clip 1.0, cosine schedule with a linear warmup over the first 10% of total training steps.

Solve-procedure training Batch size $B = 512$ with pool-to-batch multiplier 1 (the pool holds exactly one batch of states, all of which is consumed per step). Sudoku-Extreme is trained for $T = 4{,}000$ steps over the $1{,}000$-puzzle train split ($\approx 2{,}048$ epochs; $T \in \{1{,}000, 2{,}000\}$ also reported in Table 1, at ≈ 512 and $\approx 1{,}024$ epochs respectively). Maximum pool age $\tau_{\text{age}} = 100$ training steps since insertion. Pool states are advanced with elimination threshold $\theta_{\text{elim}} = 0.1$, conflict threshold $\theta_{\text{CLS}}^{\text{train}} = 0.5$ and digits sampled at decide temperature $\tau_{\text{decide}} = 1.5$. For Sudoku, each puzzle is wrapped in a random digit permutation composed with a dihedral grid symmetry before it enters the training pool, and training uses no per-step augmentation; all Sudoku-Extreme rows of Table 1 except the no-augmentation row use this. Training seeds 0, 1 and 2 (0 to 14 for the portfolio models).

No-augmentation variant The no-aug row of Table 1 drops both augmentation channels: training sees only the raw $1{,}000$-puzzle Sudoku-Extreme split with no dataset-level expansion and no per-step augmentation. Inference keeps the per-step augmentation below; without it, the model solves 31.6% (mean over seeds). All other hyperparameters match the base set.

Loss weights BCE class weights $w^+ = 4.0$ on positive candidates and $w^- = 0.5$ on negatives; auxiliary softmax cross-entropy weight $\lambda_{\text{ce}} = 0.2$; CLS BCE weight $\lambda_{\text{cls}} = 0.1$.

Inference defaults (Section 3.3) At inference we batch the parallel solve along two axes. A *slot* holds one puzzle from the moment it enters the batch until that puzzle either self-accepts (a chain reaches a complete, operator-accepted solution) or hits the per-puzzle round budget; we run $M = 8$ slots concurrently, evicting and refilling slots from a queue of remaining test puzzles. Within each slot, $H = 64$ *chains* explore the same starting state in parallel but diverge under the random choice of branching cell, per-step augmentation and eval-time dropout – effectively H independent restarts of the search for the same puzzle. One forward pass therefore spans $M \cdot H = 512$ rows. Per-puzzle round budget $R = 1{,}000$, elimination threshold $\theta_{\text{elim}} = 0.1$, eval-time dropout $p_{\text{drop}} = 0.05$, conflict threshold $\theta_{\text{CLS}}^{\text{eval}} = 0.6$, tuned on the Sudoku-Extreme train set. The branching cell is drawn uniformly at random among cells with at least two candidates, and the digit is the argmax of the digit logits s (τ_{decide} is used only in training). We additionally apply *per-step augmentation* at inference: at each Solve step the lattice state is wrapped in a fresh random digit-permutation composed with a dihedral symmetry, the model performs deduction and branching in that augmented frame, and we invert the symmetry before writing back to the canonical state. This decorrelates the parallel chains and broadens the inference search.

Sequential-cost estimation Figure 4(a, b) reports per-puzzle forward-pass counts as a sequential (batch-1) search would have paid, even though inference is run with the parallel slot/chain batch described above. Within a slot, we number *attempts* (a chain's run from the original puzzle until it is accepted, or until a conflict restarts it) in the order they start. Let attempt w be the first to reach an accepted solution. A sequential search would have run attempts $0, \ldots, w$ one after another, so the cost we report is $(w + 1)\,\bar{\ell}$, where $\bar{\ell}$ is the mean length in rounds of the attempts that ended in that slot. To observe the attempts numbered below w that are still running when w succeeds, we *drain* the

slot after the winner: it keeps running until each of them has ended (for at most 200 further rounds). Puzzles answered wrongly or not solved within the budget are charged the upper bound $R \cdot H$. This yields sequential-cost estimates without re-running the solve at batch 1.

L.2 Snowflake Sudoku

For Snowflake Sudoku (Section 4.2) we keep the Sudoku-Extreme settings above except for the following changes. Each puzzle is embedded in the fixed 150-cell covering grid (Appendix K) with 50 channels per cell: 6 candidate sigmoids, a read-only *in-puzzle* mask, and 43 read-only constraint-group channels; the deduction operator edits only the 6 candidate channels ($\approx 816{,}000$ parameters). Training runs for $T = 8{,}000$ steps on 2,000 puzzles of the training orders (1,000 for the 1K column of Table 2), with per-step digit-permutation augmentation in training and at evaluation and a random whole-hex translation of every training sample; there is no dihedral augmentation. Direct prediction uses the same model without the CLS token, trained only with the cross-entropy loss ($\lambda_{\text{ce}} = 1$, $w^+ = w^- = \lambda_{\text{cls}} = 0$) on puzzles as given, each used for a single step (pool age 0), and is evaluated in one shot without dropout.

L.3 Maze

For Maze (Section 4.3) we keep the Sudoku-Extreme settings above except for the following changes.

General Augmentation is dihedral only – digit-permutation is disabled because the channels (wall, free, start, goal, path) carry distinct semantics. Besides per-step augmentation in training and at evaluation, each training sample gets a random dihedral symmetry and a random swap of start and goal. Elimination threshold $\theta_{\text{elim}} = 0.1$ in training and at evaluation.

15×15 **setting** Used for the multi-solution supervision sweep (Figure 6b). Embedding dimension $d = 128$ as in the base set, batch size $B = 256$, total training steps $T = 4{,}000$ (≈ 102 epochs over a 10,000-puzzle synthetic training set), inference conflict threshold $\theta^{\text{eval}}_{\text{CLS}} = 0.6$ (matching the base set), $M = 4$ slots of 64 chains, and $K \in \{1, 8, 32, 64, 128, 256, 512\}$ swept across runs. Each K value is averaged over 9 training seeds (0 to 8); all seeds share one procedurally generated training set (data seed 42), so the seed controls initialization and training randomness only.

30×30 **setting** Used for the headline Maze-Hard results (Table 3). Embedding dimension $d = 192$ ($\approx$ 1.8M parameters total), with 2D RoPE inside attention in place of the base set's learned 2D positional embedding. Batch size $B = 192$, total training steps $T = 20{,}000$, pool-to-batch multiplier 2.0, eval-time dropout $p_{\text{drop}} = 0$ (vs. 0.05 elsewhere), one slot ($M = 1$) of 64 chains, inference conflict threshold $\theta^{\text{eval}}_{\text{CLS}} = 0.53$, calibrated on a held-out Maze-Hard set. We report two K configurations, both trained on the 1,000-puzzle Maze-Hard split for $T = 20{,}000$ steps ($\approx 3{,}840$ epochs): $K = 1$ (head-to-head with TRM) and $K = 512$.

M Evaluation Protocol

Every Sudoku-Extreme test evaluation walks one fixed random permutation of the 422,786 test puzzles (generated from a recorded seed and checked against a recorded SHA-256), so every model and seed sees the same puzzles in the same order. A puzzle is a timeout if no chain returns a solution within the round budget (1,000 rounds unless noted), and wrong if the returned grid differs from the solution. Full-split evaluations cover every test puzzle. Sequential evaluations stop at 100 failures (wrong answers or timeouts), or, from 1,000 puzzles on and checked every 100 puzzles, once the half-width of a 95% confidence sequence for the accuracy is at most 0.1 percentage points. A confidence sequence holds uniformly over the number of puzzles evaluated, so the interval stays valid under this data-dependent stopping. Snowflake Sudoku and Maze-Hard evaluations cover every test puzzle.

In Table 1, the 4K rows with training augmentation are evaluated on all test puzzles and the others sequentially; HRM, TRM and Sotaku v2 are evaluated on the first 10,000 puzzles of the same ordering. The number of puzzles evaluated per row, summed over seeds:

Row	Puzzles evaluated
HRM, TRM, Sotaku v2 (each)	10,000
LDT, 1K train steps	58,556
LDT, 2K train steps	26,900
LDT, 4K train steps	$3 \times 422{,}786$
10,000 search rounds	3×376 (reruns)
portfolios	$15 \times 422{,}786$ plus reruns
LDT, 4K, 1K puzzles, no training augmentation	12,914

The 10,000-round row reruns the 4K-step models' timeouts at 1,000 rounds on the 376 test puzzles that any of the 15 portfolio models left unsolved.

For Maze-Hard (Table 3), every method is evaluated on all 1,000 test mazes. LDT keeps a snapshot every 1,000 training steps; we evaluate them at $\theta_{\text{elim}} = 0.1$ and report the best per seed. For HRM and TRM we evaluate every saved checkpoint of our retrains and report the best; their exact-match accuracies (path identical to the dataset's reference path) are 16.7% (HRM, trained with augmentation) and 84.5% (TRM). Training compute in Table 3 counts each method up to its reported checkpoint; for LDT, the best snapshots are at 12K, 14K and 18K steps ($K = 1$) and at 20K, 18K and 18K steps ($K = 512$), and we report the mean over seeds.

N Compute Accounting

B200 time All times are on a single NVIDIA B200. LDT training time is the measured training loop after compilation, excluding periodic monitoring solves. We did not train HRM or TRM on Sudoku-Extreme. For their training time we run each released checkpoint's published training configuration for 12 minutes on one B200, read the steady-state time per step, and multiply by the checkpoint's step count:

Model	s / step	Steps	B200 hours
HRM	0.059	52,083	0.85
TRM (MLP)	0.524	32,550	4.7
EqR	0.036	50,000	0.50
Sotaku v2	0.294	50,000	4.1

HRM's step count is its 40,000 epochs over 1,000 puzzle groups at global batch 768; the TRM checkpoint was saved at step 32,550 of a run at global batch 1,536. HRM's release does not record its step, so its count is an upper bound. EqR shows no training progress without its logging service, so we time it as the difference between runs capped at 200 and 1,200 steps; Sotaku v2 is timed the same way on its 50K reference recipe (steps 200 and 900, after an untimed run that fills its data caches). For Maze-Hard we retrained HRM and TRM ourselves and derive their time to the reported checkpoint the same way, from measured throughputs of 0.61 (HRM) and 1.46 (TRM) seconds per step. These are the costs of the published recipes on our hardware; none of the methods, including ours, is tuned for B200 utilization. Inference cost is the evaluation's GPU time divided by the number of puzzles evaluated.

Benchmark	Method	Training	Inference (s / puzzle)
Sudoku-Extreme	HRM	51m†	0.014
	TRM, MLP	4.7h†	0.011
	Sotaku v2	4.1h†	0.0091
	EqR	30m†	0.61
	LDT, 1K train steps	3m	0.26
	LDT, 2K train steps	7m	0.094
	LDT, 4K train steps	13m	0.042
	10,000 search rounds	13m	0.044
	no training augmentation	11m	0.024
Maze-Hard	HRM	20.7h†	—
	TRM	7.9h†	—
	LDT, $K = 1$	6.9h	—
	LDT, $K = 512$	9.1h	—

[†] Estimated from measured throughput; LDT times are measured.

Inference FLOPs We count a multiply-add as 2 FLOPs and include, for every layer application, the attention projections ($4d^2$ per token), the attention scores and values ($2t^2d$ over t tokens) and the feed-forward block; embeddings, normalization and output heads are omitted (under 2% at these sizes). One LDT model call runs $L = 16$ iterations of 4 layers ($d = 128$, 81 cells plus the CLS token) for each of the 64 chains of a puzzle; the cost per puzzle is the mean number of model calls per puzzle times 64 times the cost of one chain's call. HRM and TRM run their full 16 ACT steps at evaluation, each of 24 (HRM) or 42 (TRM) layer applications at $d = 512$ with 16 (TRM) or 1 (HRM) extra puzzle-embedding positions; TRM's MLP variant replaces attention with a SwiGLU across positions. EqR uses TRM's MLP backbone and, at its reported setting, runs 64 ACT steps for each of 128 restarts. Sotaku v2 runs 1,024 iterations of 4 layers ($d = 128$, feed-forward width 512). These are estimates of arithmetic, not measurements, and ignore early stopping: LDT stops when a puzzle is solved, which the mean call count reflects, while the others run their full budget.

Method (setting)	GFLOPs per step	TFLOPs per puzzle
Sotaku v2 (1,024 iterations)	0.14 / iteration	0.14
HRM (16 ACT steps)	13.7 / ACT step	0.22
TRM, MLP (16 ACT steps)	25.6 / ACT step	0.41
EqR (depth 64, breadth 128)	25.6 / ACT step	210
LDT, 1K train steps (65.4 calls × 64 chains)	2.28 / chain call	9.6
LDT, 2K train steps (22.2 calls)	2.28 / chain call	3.2
LDT, 4K train steps (10.1 calls)	2.28 / chain call	1.5

Training FLOPs A training step of LDT costs 4 forward passes per sample: the forward and backward pass through all 16 iterations (3 forward-pass equivalents) and the no-grad forward that advances the pool states. HRM and TRM backpropagate only through the last of their recursion cycles: TRM runs 3 H-cycles and backpropagates through one ($1\frac{2}{3}$ forward-pass equivalents per sample); HRM backpropagates through its last L- and H-module calls, 8 of its 24 layer applications, and runs one more no-grad forward for its halting target ($2\frac{2}{3}$). Sotaku v2 trains 16 iterations with gradients, preceded on 20% of batches by 32 to 512 iterations without. We do not count LDT's periodic monitoring solves, which change no weights; they would add up to about 30% to its training FLOPs. Model FLOPs utilization (MFU) is the training FLOPs divided by the measured or estimated B200 time, relative to the B200's dense BF16 peak of 2.25 PFLOP/s.

Method	Steps	Batch	PFLOPs	MFU
HRM	52,083	768	1,466	21%
TRM, MLP	32,550	1,536	2,136	5.6%
EqR	50,000	768	1,640	40%
Sotaku v2	50,000	2,048	1,265	3.8%
LDT, 1K train steps	1,000	512	4.7	1.0%
LDT, 2K train steps	2,000	512	9.4	1.1%
LDT, 4K train steps	4,000	512	18.7	1.1%

LDT trains with 78 to 114 times fewer FLOPs than HRM and TRM, but its small model uses the GPU far less fully, so its advantage in B200 time (Table 1) is smaller. EqR uses TRM's architecture and the same one-step gradient, so it costs the same FLOPs per step, yet its released code runs about 7 times as fast on the B200 as the TRM implementation we measured; MFU depends as much on the implementation as on the method.

NeurIPS Paper Checklist

1. **Claims**

   Question: Do the main claims made in the abstract and introduction accurately reflect the paper's contributions and scope?

   Answer: [Yes]

   Justification: The abstract and introduction clearly state the paper's contribution, which is the introduction and evaluation of a novel lattice deduction transformer. The claims made accurately reflect the experimental results showing improved performance.

   Guidelines:

   - The answer [N/A] means that the abstract and introduction do not include the claims made in the paper.
   - The abstract and/or introduction should clearly state the claims made, including the contributions made in the paper and important assumptions and limitations. A [No] or [N/A] answer to this question will not be perceived well by the reviewers.
   - The claims made should match theoretical and experimental results, and reflect how much the results can be expected to generalize to other settings.
   - It is fine to include aspirational goals as motivation as long as it is clear that these goals are not attained by the paper.

2. **Limitations**

   Question: Does the paper discuss the limitations of the work performed by the authors?

   Answer: [Yes]

   Justification: Limitations are discussed in the conclusion section.

   Guidelines:

   - The answer [N/A] means that the paper has no limitation while the answer [No] means that the paper has limitations, but those are not discussed in the paper.
   - The authors are encouraged to create a separate "Limitations" section in their paper.
   - The paper should point out any strong assumptions and how robust the results are to violations of these assumptions (e.g., independence assumptions, noiseless settings, model well-specification, asymptotic approximations only holding locally). The authors should reflect on how these assumptions might be violated in practice and what the implications would be.
   - The authors should reflect on the scope of the claims made, e.g., if the approach was only tested on a few datasets or with a few runs. In general, empirical results often depend on implicit assumptions, which should be articulated.
   - The authors should reflect on the factors that influence the performance of the approach. For example, a facial recognition algorithm may perform poorly when image resolution is low or images are taken in low lighting. Or a speech-to-text system might not be used reliably to provide closed captions for online lectures because it fails to handle technical jargon.
   - The authors should discuss the computational efficiency of the proposed algorithms and how they scale with dataset size.
   - If applicable, the authors should discuss possible limitations of their approach to address problems of privacy and fairness.
   - While the authors might fear that complete honesty about limitations might be used by reviewers as grounds for rejection, a worse outcome might be that reviewers discover limitations that aren't acknowledged in the paper. The authors should use their best judgment and recognize that individual actions in favor of transparency play an important role in developing norms that preserve the integrity of the community. Reviewers will be specifically instructed to not penalize honesty concerning limitations.

3. **Theory assumptions and proofs**

   Question: For each theoretical result, does the paper provide the full set of assumptions and a complete (and correct) proof?

Answer: [N/A]

Justification: The paper focuses on empirical evaluation of a new transformer architecture and does not present new theoretical results requiring formal assumptions or proofs.

Guidelines:

- The answer [N/A] means that the paper does not include theoretical results.
- All the theorems, formulas, and proofs in the paper should be numbered and cross-referenced.
- All assumptions should be clearly stated or referenced in the statement of any theorems.
- The proofs can either appear in the main paper or the supplemental material, but if they appear in the supplemental material, the authors are encouraged to provide a short proof sketch to provide intuition.
- Inversely, any informal proof provided in the core of the paper should be complemented by formal proofs provided in appendix or supplemental material.
- Theorems and Lemmas that the proof relies upon should be properly referenced.

4. **Experimental result reproducibility**

   Question: Does the paper fully disclose all the information needed to reproduce the main experimental results of the paper to the extent that it affects the main claims and/or conclusions of the paper (regardless of whether the code and data are provided or not)?

   Answer: [Yes]

   Justification: All the methodology is detailed in section 4, while the experimental benchmarks and hardware and resource settings are detailed in section 5.

   Guidelines:

   - The answer [N/A] means that the paper does not include experiments.
   - If the paper includes experiments, a [No] answer to this question will not be perceived well by the reviewers: Making the paper reproducible is important, regardless of whether the code and data are provided or not.
   - If the contribution is a dataset and/or model, the authors should describe the steps taken to make their results reproducible or verifiable.
   - Depending on the contribution, reproducibility can be accomplished in various ways. For example, if the contribution is a novel architecture, describing the architecture fully might suffice, or if the contribution is a specific model and empirical evaluation, it may be necessary to either make it possible for others to replicate the model with the same dataset, or provide access to the model. In general. releasing code and data is often one good way to accomplish this, but reproducibility can also be provided via detailed instructions for how to replicate the results, access to a hosted model (e.g., in the case of a large language model), releasing of a model checkpoint, or other means that are appropriate to the research performed.
   - While NeurIPS does not require releasing code, the conference does require all submissions to provide some reasonable avenue for reproducibility, which may depend on the nature of the contribution. For example
     (a) If the contribution is primarily a new algorithm, the paper should make it clear how to reproduce that algorithm.
     (b) If the contribution is primarily a new model architecture, the paper should describe the architecture clearly and fully.
     (c) If the contribution is a new model (e.g., a large language model), then there should either be a way to access this model for reproducing the results or a way to reproduce the model (e.g., with an open-source dataset or instructions for how to construct the dataset).
     (d) We recognize that reproducibility may be tricky in some cases, in which case authors are welcome to describe the particular way they provide for reproducibility. In the case of closed-source models, it may be that access to the model is limited in some way (e.g., to registered users), but it should be possible for other researchers to have some path to reproducing or verifying the results.

5. **Open access to data and code**

Question: Does the paper provide open access to the data and code, with sufficient instructions to faithfully reproduce the main experimental results, as described in supplemental material?

Answer: [Yes]

Justification: We make the implementation available in the supplementary material.

Guidelines:

- The answer [N/A] means that paper does not include experiments requiring code.
- Please see the NeurIPS code and data submission guidelines (`https://neurips.cc/public/guides/CodeSubmissionPolicy`) for more details.
- While we encourage the release of code and data, we understand that this might not be possible, so [No] is an acceptable answer. Papers cannot be rejected simply for not including code, unless this is central to the contribution (e.g., for a new open-source benchmark).
- The instructions should contain the exact command and environment needed to run to reproduce the results. See the NeurIPS code and data submission guidelines (`https://neurips.cc/public/guides/CodeSubmissionPolicy`) for more details.
- The authors should provide instructions on data access and preparation, including how to access the raw data, preprocessed data, intermediate data, and generated data, etc.
- The authors should provide scripts to reproduce all experimental results for the new proposed method and baselines. If only a subset of experiments are reproducible, they should state which ones are omitted from the script and why.
- At submission time, to preserve anonymity, the authors should release anonymized versions (if applicable).
- Providing as much information as possible in supplemental material (appended to the paper) is recommended, but including URLs to data and code is permitted.

6. **Experimental setting/details**

   Question: Does the paper specify all the training and test details (e.g., data splits, hyperparameters, how they were chosen, type of optimizer) necessary to understand the results?

   Answer: [Yes]

   Justification: Section 5 details the experimental settings used and the appendix details all the hyperparameters.

   Guidelines:

   - The answer [N/A] means that the paper does not include experiments.
   - The experimental setting should be presented in the core of the paper to a level of detail that is necessary to appreciate the results and make sense of them.
   - The full details can be provided either with the code, in appendix, or as supplemental material.

7. **Experiment statistical significance**

   Question: Does the paper report error bars suitably and correctly defined or other appropriate information about the statistical significance of the experiments?

   Answer: [N/A]

   Justification: No rigorous statistical analysis was done for the experiments, meaning error bars are not part of the results.

   Guidelines:

   - The answer [N/A] means that the paper does not include experiments.
   - The authors should answer [Yes] if the results are accompanied by error bars, confidence intervals, or statistical significance tests, at least for the experiments that support the main claims of the paper.
   - The factors of variability that the error bars are capturing should be clearly stated (for example, train/test split, initialization, random drawing of some parameter, or overall run with given experimental conditions).

- The method for calculating the error bars should be explained (closed form formula, call to a library function, bootstrap, etc.)
- The assumptions made should be given (e.g., Normally distributed errors).
- It should be clear whether the error bar is the standard deviation or the standard error of the mean.
- It is OK to report 1-sigma error bars, but one should state it. The authors should preferably report a 2-sigma error bar than state that they have a 96% CI, if the hypothesis of Normality of errors is not verified.
- For asymmetric distributions, the authors should be careful not to show in tables or figures symmetric error bars that would yield results that are out of range (e.g., negative error rates).
- If error bars are reported in tables or plots, the authors should explain in the text how they were calculated and reference the corresponding figures or tables in the text.

8. **Experiments compute resources**

   Question: For each experiment, does the paper provide sufficient information on the computer resources (type of compute workers, memory, time of execution) needed to reproduce the experiments?

   Answer: [Yes]

   Justification: The exact hardware and resource limits used in the experiments are detailed in section 5.

   Guidelines:

   - The answer [N/A] means that the paper does not include experiments.
   - The paper should indicate the type of compute workers CPU or GPU, internal cluster, or cloud provider, including relevant memory and storage.
   - The paper should provide the amount of compute required for each of the individual experimental runs as well as estimate the total compute.
   - The paper should disclose whether the full research project required more compute than the experiments reported in the paper (e.g., preliminary or failed experiments that didn't make it into the paper).

9. **Code of ethics**

   Question: Does the research conducted in the paper conform, in every respect, with the NeurIPS Code of Ethics `https://neurips.cc/public/EthicsGuidelines`?

   Answer: [Yes]

   Justification: We have reviewed the NeurIPS Code of Ethics and confirm that the research conducted in this paper adheres to its principles, ensuring integrity in our methods and reporting.

   Guidelines:

   - The answer [N/A] means that the authors have not reviewed the NeurIPS Code of Ethics.
   - If the authors answer [No], they should explain the special circumstances that require a deviation from the Code of Ethics.
   - The authors should make sure to preserve anonymity (e.g., if there is a special consideration due to laws or regulations in their jurisdiction).

10. **Broader impacts**

    Question: Does the paper discuss both potential positive societal impacts and negative societal impacts of the work performed?

    Answer: [N/A]

    Justification: This is foundational methods research on neural reasoning over structured combinatorial puzzles (Sudoku, snowflake Sudoku, mazes). We do not see a direct path from this work to specific positive or negative societal applications, and the paper does not deploy the model in any user-facing setting.

    Guidelines:

- The answer [N/A] means that there is no societal impact of the work performed.
- If the authors answer [N/A] or [No], they should explain why their work has no societal impact or why the paper does not address societal impact.
- Examples of negative societal impacts include potential malicious or unintended uses (e.g., disinformation, generating fake profiles, surveillance), fairness considerations (e.g., deployment of technologies that could make decisions that unfairly impact specific groups), privacy considerations, and security considerations.
- The conference expects that many papers will be foundational research and not tied to particular applications, let alone deployments. However, if there is a direct path to any negative applications, the authors should point it out. For example, it is legitimate to point out that an improvement in the quality of generative models could be used to generate Deepfakes for disinformation. On the other hand, it is not needed to point out that a generic algorithm for optimizing neural networks could enable people to train models that generate Deepfakes faster.
- The authors should consider possible harms that could arise when the technology is being used as intended and functioning correctly, harms that could arise when the technology is being used as intended but gives incorrect results, and harms following from (intentional or unintentional) misuse of the technology.
- If there are negative societal impacts, the authors could also discuss possible mitigation strategies (e.g., gated release of models, providing defenses in addition to attacks, mechanisms for monitoring misuse, mechanisms to monitor how a system learns from feedback over time, improving the efficiency and accessibility of ML).

11. **Safeguards**

    Question: Does the paper describe safeguards that have been put in place for responsible release of data or models that have a high risk for misuse (e.g., pre-trained language models, image generators, or scraped datasets)?

    Answer: [N/A]

    Justification: The presented models are trained as Sudoku and maze solvers, not at high risk for misuse. The datasets are existing datasets or procedurally generated also not at high risk for misuse.

    Guidelines:

    - The answer [N/A] means that the paper poses no such risks.
    - Released models that have a high risk for misuse or dual-use should be released with necessary safeguards to allow for controlled use of the model, for example by requiring that users adhere to usage guidelines or restrictions to access the model or implementing safety filters.
    - Datasets that have been scraped from the Internet could pose safety risks. The authors should describe how they avoided releasing unsafe images.
    - We recognize that providing effective safeguards is challenging, and many papers do not require this, but we encourage authors to take this into account and make a best faith effort.

12. **Licenses for existing assets**

    Question: Are the creators or original owners of assets (e.g., code, data, models), used in the paper, properly credited and are the license and terms of use explicitly mentioned and properly respected?

    Answer: [Yes]

    Justification: We have properly cited the original papers and repositories for the benchmark sets used in section 5.

    Guidelines:

    - The answer [N/A] means that the paper does not use existing assets.
    - The authors should cite the original paper that produced the code package or dataset.
    - The authors should state which version of the asset is used and, if possible, include a URL.

- The name of the license (e.g., CC-BY 4.0) should be included for each asset.
- For scraped data from a particular source (e.g., website), the copyright and terms of service of that source should be provided.
- If assets are released, the license, copyright information, and terms of use in the package should be provided. For popular datasets, `paperswithcode.com/datasets` has curated licenses for some datasets. Their licensing guide can help determine the license of a dataset.
- For existing datasets that are re-packaged, both the original license and the license of the derived asset (if it has changed) should be provided.
- If this information is not available online, the authors are encouraged to reach out to the asset's creators.

13. **New assets**

    Question: Are new assets introduced in the paper well documented and is the documentation provided alongside the assets?

    Answer: [Yes]

    Justification: Our implementation is well documented in the supplementary material.

    Guidelines:

    - The answer [N/A] means that the paper does not release new assets.
    - Researchers should communicate the details of the dataset/code/model as part of their submissions via structured templates. This includes details about training, license, limitations, etc.
    - The paper should discuss whether and how consent was obtained from people whose asset is used.
    - At submission time, remember to anonymize your assets (if applicable). You can either create an anonymized URL or include an anonymized zip file.

14. **Crowdsourcing and research with human subjects**

    Question: For crowdsourcing experiments and research with human subjects, does the paper include the full text of instructions given to participants and screenshots, if applicable, as well as details about compensation (if any)?

    Answer: [N/A]

    Justification: There were no crowdsourcing experiments or research with human subjects conducted.

    Guidelines:

    - The answer [N/A] means that the paper does not involve crowdsourcing nor research with human subjects.
    - Including this information in the supplemental material is fine, but if the main contribution of the paper involves human subjects, then as much detail as possible should be included in the main paper.
    - According to the NeurIPS Code of Ethics, workers involved in data collection, curation, or other labor should be paid at least the minimum wage in the country of the data collector.

15. **Institutional review board (IRB) approvals or equivalent for research with human subjects**

    Question: Does the paper describe potential risks incurred by study participants, whether such risks were disclosed to the subjects, and whether Institutional Review Board (IRB) approvals (or an equivalent approval/review based on the requirements of your country or institution) were obtained?

    Answer: [N/A]

    Justification: There were no human participants involved in the research.

    Guidelines:

    - The answer [N/A] means that the paper does not involve crowdsourcing nor research with human subjects.

- Depending on the country in which research is conducted, IRB approval (or equivalent) may be required for any human subjects research. If you obtained IRB approval, you should clearly state this in the paper.
- We recognize that the procedures for this may vary significantly between institutions and locations, and we expect authors to adhere to the NeurIPS Code of Ethics and the guidelines for their institution.
- For initial submissions, do not include any information that would break anonymity (if applicable), such as the institution conducting the review.

16. **Declaration of LLM usage**

    Question: Does the paper describe the usage of LLMs if it is an important, original, or non-standard component of the core methods in this research? Note that if the LLM is used only for writing, editing, or formatting purposes and does *not* impact the core methodology, scientific rigor, or originality of the research, declaration is not required.

    Answer: [N/A]

    Justification: LLMs appear only as zero-shot baselines in Section 5, and no LLM is part of the LDT method itself.

    Guidelines:

    - The answer [N/A] means that the core method development in this research does not involve LLMs as any important, original, or non-standard components.
    - Please refer to our LLM policy in the NeurIPS handbook for what should or should not be described.