The Role of Transformer Architecture in the Logic-as-Loss Framework

Mattia Medina Grespan and Vivek Srikumar (✉)

Kahlert School of Computing, University of Utah, Salt Lake City UT 84112, USA
{mattiamg, svivek}@cs.utah.edu

Abstract. The logic-as-loss framework has enabled transformer models to incorporate domain knowledge by encoding logical constraints as differentiable objectives, allowing neural networks to learn from explicit rules. Despite its effectiveness across diverse tasks, the relationship between neural architecture and rule internalization remains poorly understood. This study systematically investigates how transformer encoder configurations influence the ingestion of logical rules, beyond simply scaling up model capacity. We aim to identify the architectural factors that enable successful rule internalization and the inherent limitations of this process. Empirical analysis on controlled reasoning tasks reveals a capacity threshold: transformers perform poorly at rule adherence below a critical parameter count, while performance plateaus above it. A key finding is that embedding dimensionality drives rule ingestion efficacy, while increased network depth mitigates spurious solutions that satisfy rules without improving task performance. Our work highlights the role of architectural design choices for effective neuro-symbolic learning.

Keywords: Neuro-symbolic models · Logic-as-loss · Knowledge-driven learning.

1 Introduction

Integrating domain knowledge into deep learning models using logic rules is an effective strategy to address data inefficiencies [9]. One successful approach calls for encoding data and task-specific rules declaratively in predicate logic, which is then compiled to define loss functions [13,10,30,17,22]. The models trained with these losses not only fit observed data but also tend to adhere to domain constraints. We will call this the *logic-as-loss* framework, which includes two broad technical strategies: using model-counting based approaches (e.g., the semantic loss [30]), or using t-norm logic relaxations [13]. Both transform logical formulas into sub-differentiable loss functions, and have shown success across a diverse set of tasks [17,4]. In this work, we focus on the t-norm logic relaxations.

Despite empirical successes across multiple tasks, especially those involving the transformer architecture, the factors that lead to this effectiveness are relatively underexplored. In particular, while previous work compares different t-norm relaxations [10,24,7], the interplay between the neural network architectural choices and the success of the logic-as-loss agenda remains unexamined.

By training neural networks to produce only logically consistent outputs, we expect model parameters to encode the rules. *How do the architectural choices, and consequently the expressive capacity, of a model influence its ability to ingest and enforce logical constraints?* Although transformer models [26] are ubiquitous across many domains [20,6], there is no systematic analysis of how their configurations — layer depth, embedding dimensionality, or number of attention heads — affect the ability to ingest rules for multi-hop logical reasoning.

In this paper, we investigate the impact of transformer architectural choices on the logic-as-loss approach. We ask: (1) How large must a model be (i.e., how many parameters are needed) to learn complex logical constraints? (2) How do architectural factors (beyond sheer parameter count) influence this process? We employ controlled reasoning tasks to study these questions: collinearity in the plane and two Latin square based puzzles, Futoshiki and Sudoku. For each task, we examine the impact of transformer architectural parameters — embedding dimensionality, number of encoder layers, attention heads, and feedforward dimensionality — on the ability to ingest rules via losses.

Across these experiments, we find a capacity threshold below which the model struggles to learn logical constraints; once the threshold is crossed, performance stabilizes. Embedding dimensionality emerges as an especially critical design choice, while increased depth helps mitigate vacuous rule satisfaction. In certain settings, smaller models even outperform larger ones, suggesting that scaling alone does not guarantee better logic ingestion. Together, these observations underscore the importance of tailoring architectural choices — particularly embedding size and depth — to reap the benefits of the logic-as-loss approach.

Our contributions are threefold. First, we design a focused set of controlled tasks to help study the ability of neural models to learn from rules. Second, we systematically explore the influence of transformer architecture in the logic-as-loss framework, revealing how model capacity, depth, and embedding size shape performance. Finally, we discuss key insights on balancing depth versus width and present guidelines for designing models that effectively integrate domain knowledge through logical constraints.[1]

2 Logic-as-Loss Approach: Background

The logic-as-loss approach views machine learning tasks as declarative contracts for a neural network to satisfy. Models are required not only to produce correct labels for supervised examples, but also to satisfy logical constraints drawn from domain knowledge. This section gives a brief overview of the framework.

2.1 Declarative Specification with Predicate Logic

Predicate logic is the specification language in the logic-as-loss framework. Let $\mathcal{X}$ be the input domain and $\mathcal{Y}$ the finite set of task labels. A labeled dataset is

[1] The code and data for replicating our results, along with the appendix document, are archived at `https://github.com/utahnlp/logic_as_loss_with_transformers/`.

a set $D \subset \mathcal{X} \times \mathcal{Y}$ of pairs (x, y), while $U \subset \mathcal{X}$ denotes additional inputs drawn from the same distribution but without ground-truth labels.

Atomic Predicates and Rules. For every label $y \in \mathcal{Y}$ we introduce the atomic predicate

$$\mathtt{y}(x) \ : \text{“the instance } x \text{ has the label } y\text{”}.$$

Task-specific axioms or expert knowledge are expressed by rules $C(x)$ that may apply to any instance $x \in D \cup U$. Because every unlabeled input will ultimately be mapped to some $y \in \mathcal{Y}$ by the model, the atomic predicates $\mathtt{y}(x)$ and the constraints $C(x)$ are well-defined for all inputs in $D \cup U$.

Logic-as-Loss Objective. The labeled examples require the model to match ground-truth labels, while the rules propagate domain knowledge to all inputs. Learning amounts to ensuring the following logical expression holds:

$$\left(\bigwedge_{(x,y)\in D} \mathtt{y}(x) \right) \bigwedge \left(\bigwedge_{(x\in U)} C(x) \right). \tag{1}$$

2.2 Loss Relaxation with T-norm Logics

A common strategy to construct losses from the specification above employs t-norms [12] that generalize the logical connectives into the real domain. Each element in the declarative loss is relaxed by translating the connectives with their corresponding relaxation definition, enabling standard gradient-based optimization. There are infinitely many t-norms; Table 1 in Appendix A presents the three canonical t-norms typically used.

Neural models predict the probabilities for the atomic predicates. The goal of learning is to find model parameters that maximize a real-valued relaxation of the specification (1). Logic directly yields a differentiable loss function, making it possible to learn the predicate models by optimizing the total loss derived from the labeled data D and the constraints C applied to unlabeled examples:

$$L = L_D + \lambda L_C. \tag{2}$$

Here, λ is a non-negative hyper-parameter for the knowledge constraints.

T-norm losses have been successfully used in recent literature across multiple tasks [13,27,17]. While the underlying framework is agnostic to the choice of the neural network, these applications use language models that rely on the transformer family of models.

2.3 General Learning Setting for the Logic-as-Loss Framework

Let $O = \{o_1, o_2, \ldots, o_m\}$ denote a set of objects (e.g., sentences, one cell in a Sudoku game). An instance x may consist of one or more objects from the set O (e.g., $x = o_1$ or $x = (o_1, o_2, o_3)$) and represents a collection of objects (e.g.,

a document, or an entire Sudoku with multiple cells). Let $Y = \{\mathtt{y}_1, \mathtt{y}_2, \ldots, \mathtt{y}_n\}$ be the set of possible task labels that are attached to objects in instances. We associate each label y_i with a predicate $\mathtt{y}_i$ defined over objects in the set O. A single labeled observation (o, y) corresponds to the predicate $\mathtt{y}(o)$. Together, we have the labeled dataset D in (1). Additionally, logical rules specify how these predicates labels interact within an instance (e.g., $\mathtt{y}_1(o_1) \rightarrow \neg\mathtt{y}_2(o_2)$). Aggregated across instances, we get the constraints C in (1).

We can identify two learning settings that differ in how many labeled observations are available.

Setting 1: Learning with Supervision or the "Generalization" Setting. An instance x may contain both labeled objects (observations) and unlabeled ones. The observations and task rules constitute the sets D and C in (1) respectively. The constraints C indirectly supervise the unlabeled objects through their relationships with labeled objects. In this setting, we evaluate the model by its accuracy on objects whose labels were entirely withheld during training—revealed only at test time—together with the degree to which its predictions satisfy the specified logical rules.

Setting 2: Rule-Only or the "Solving" Setting. A second scenario involves no direct supervision (i.e., $D = \emptyset$). The truth value of every predicate is determined solely by instance-level logical constraints. This is akin to solving a puzzle with internally consistent labels. Here, the training process adapts the model parameters to satisfy all the rules for a single instance. Because there is no labeled data and no constraints crossing instances, we can disentangle the instances from each other. In other words, there is no notion of generalizing beyond a given instance. This scenario resembles a classical constraint satisfaction problem, rather than a standard predictive task.

These two settings illustrate how logic-as-loss can flexibly accommodate different availability of labeled data and types of domain knowledge, ranging from partially supervised tasks to purely rule-defined puzzles.

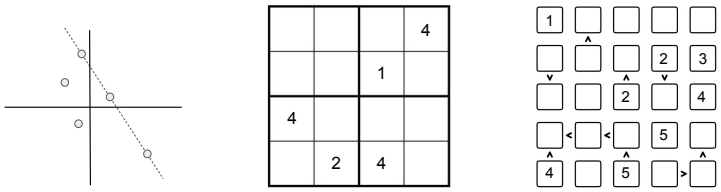
3 Investigating Logic-Aware Transformers

The logic-as-loss framework requires neural networks to produce logically consistent outputs after training. This drives the network's parameters to capture the task-specific rules introduced during training. A natural question then arises: *how do the size and complexity of a model impact its ability to learn from rules?*

We focus on transformer networks. They are not only the de facto choice for a broad range of tasks, but have also been successfully used in logic-as-loss pipelines as noted in §2. Yet, little is known about the impact of architectural design decisions on their ability to internalize and enforce logical constraints. To probe this, we use three controlled tasks. We introduce a new **collinearity** task in §3.1, which requires inferring whether a set of points on a plane are collinear. We examine two Latin square reasoning puzzles — specifically, **Sudoku**

and **Futoshiki** — in §3.2. Due to their structural complexity, these have been used in recent literature to study model reasoning capabilities [5,18,21].

By varying transformer size and architectural configurations in these tasks, we aim to uncover when and why logic-as-loss succeeds (or fails), and how model design contributes to logically consistent behavior.



(a) Collinearity (b) Sudoku (c) Futoshiki

Fig. 1: Controlled reasoning tasks used for evaluation. (a) **Collinearity**: determine whether a given set of points in the plane lies on a common straight line. (b) **Sudoku** (4×4 variant): complete the grid so that each row, column, and 2×2 block contains all digits exactly once. (c) **Futoshiki** (5×5 variant): fill the grid such that each row and column contains all digits exactly once, while also satisfying the specified inequality constraints between adjacent cells; note that no sub-block constraints apply.

3.1 Collinearity in the Real Plane

We introduce a new task of detecting collinearity among points in the real plane. Each instance is a finite *set* of points $\{p_1, p_2, \dots\} \subset \mathbb{R}^2$ (Figure 1a). We define multiple collinearity-related predicates, detailed below.

At the core of the collinearity task is the predicate $\texttt{Collinear3}(p_1, p_2, p_3)$, which is *true* if, and only if, three points p_1, p_2, p_3 lie on the same straight line. This predicate serves as the primary source of supervision: we have a labeled dataset of triplets indicating whether they are collinear or not.

Building on `Collinear3`, we define three more predicates by logical formulas that tie back to `Collinear3`.

1. **Collinearity of four points:** A group of four points is collinear if *every triplet* within that group is collinear.

$$\texttt{Collinear4}(p_1, p_2, p_3, p_4) \;\leftrightarrow \bigwedge_{1 \le i < j < k \le 4} \texttt{Collinear3}(p_i,\, p_j,\, p_k). \tag{3}$$

2. **Presence of 3-point collinearity:** A set S *has three collinear points* if it contains at least one collinear triplet.

$$\texttt{HasCollinear3}(S) \;\leftrightarrow\; \exists\, p_1, p_2, p_3 \in S : \texttt{Collinear3}(p_1,\, p_2,\, p_3). \tag{4}$$

3. **Presence of 4-point collinearity:** A set S *has four collinear points* if it contains at least one collinear subset with four points.

$$\texttt{HasCollinear4}(S) \leftrightarrow \exists\, p_1, p_2, p_3, p_4 \in S : \texttt{Collinear4}(p_1,\, p_2,\, p_3,\, p_4). \quad (5)$$

Note that *only* `Collinear3` is supervised directly from data. The rest — `Collinear4`, `HasCollinear3`, and `HasCollinear4` — receive no direct labels and instead derive their semantics through their definitions.

Joint Training via Logic-as-Loss. We use a common transformer network to embed an instance (a point set), and define separate classification heads for each predicate. Using logic-as-loss, we can derive the t-norm losses corresponding to labeled examples and each of the constraints. (Due to space constraints, we do not show the relaxations here. They can be obtained systematically as detailed in §2 and Appendix A.) The loss penalizes the network whenever `Collinear3` deviates from its labeled ground truth *or* any of the derived formulas are violated. That is, the training objective enforces: (a) correct classification of labeled triplets for `Collinear3`, and (b) consistency with logical rules (3)–(5).

This task exemplifies the setting 1 of partially supervised logic-as-loss in §2.3. By minimizing logical violations, the model learns to propagate collinearity knowledge from the supervised triplets to the more complex unlabeled predicates relying solely on rules.

3.2 Sudoku and Futoshiki

Beyond geometric reasoning, we consider two logic puzzles: *Sudoku* and *Futoshiki*. Both are defined over an $n \times n$ grid, where some cells are *given* (pre-filled) and the rest must be *predicted*.

A standard Sudoku grid is governed by three *fixed* constraints that apply to every instance: (i) each row must contain the digits $1, \ldots, n$ exactly once, (ii) each column must contain the digits $1, \ldots, n$ exactly once, and (iii) each $\sqrt{n} \times \sqrt{n}$ sub-grid (or "block") must also contain each digit exactly once (Figure 1b).

Futoshiki[2] is a Latin square puzzle variant that always enforces the row and column uniqueness constraints (i)–(ii). Unlike Sudoku, it lacks block constraints. Instead, each instance includes a set of $<$ or $>$ relations between adjacent cells (Figure 1c). These inequality constraints are dynamic: both their number and placement vary across puzzles.

The label space for these tasks comprises all possible digit assignments for each cell, formalized as predicates $\texttt{Cell}(r, c, d)$: the cell in row r and column c contains digit d. Puzzle-specific rules (e.g., "each digit must appear exactly once in every row, column, and sub-grid" for Sudoku, or Futoshiki inequalities) form the logical constraints.

For example, consider the rule that each digit must appear exactly once in every row. First, each digit must appear on each row: for each digit d and row

[2] From the Japanese word for "inequality." The standard grid size is $n = 5$, though larger sizes are common.

r, some column c contains the digit d. Second, no digit can appear more than once in each row. This is, for a digit d, a row r, and two columns c, c', the cells (r, c) and (r, c') cannot both contain the digit d. That is, we have

$$\forall d, r \, \exists c : \texttt{Cell}(r, c, d). \tag{6}$$

$$\forall d, r, c, c' \neq c : \neg[\texttt{Cell}(r, c, d) \wedge \texttt{Cell}(r, c', d)]. \tag{7}$$

Puzzle-Solving Setting. Each puzzle is an independent instance. So, given a puzzle, we can train a model to solve *only that puzzle*. The system trivially learns the given cells (as they are fully known) and relies on puzzle rules to fill in the remaining cells. Since there is no labeled supervision for the hidden cells, the model must derive their values exclusively from the logical constraints. This instantiates the purely rule-defined learning scenario, i.e., setting 2 from §2.3.

Generalization Setting. In this setting, we train on a collection of distinct Sudoku or Futoshiki instances—each defined by its own set of given cells and, in the case of Futoshiki, a unique pattern of cell-to-cell inequalities. We evaluate on a disjoint test set comprising puzzle configurations that were never seen during training. Thus, each test example is an *unseen puzzle*, differing both in its initial given values and, for Futoshiki, in its inequality layout.

1. **Given-Cells Only**: Only the puzzle's given cells are available, forcing the model to *copy* them into the solution and use the *rules* for the other cells.
2. **Full Solutions**: Each puzzle's complete solution is labeled, offering direct supervision for every cell in addition to the puzzle constraints.

By training over multiple puzzles, we seek to learn a *general procedure* for solving new puzzles in a single forward pass. This setting is considerably more challenging as it incentivizes network parameters to internalize rules and patterns across various puzzle configurations.

4 Experimental Setup

Our experiments share a common framework built around the transformer architecture. Given input sequences (e.g., points in the real plane or puzzle cells), we first apply an embedding layer and a positional encoding module, then process them with a transformer encoder. The encoder output is mapped to a set of linear classifier heads, each corresponding to an atomic predicate in the declarative objective. Following [10], we use the R-product t-norm relaxations of logic to obtain loss functions. Appendix A provides additional details.

Training proceeds in two stages. First, we optimize predicates whose ground-truth labels are available, monitoring performance on a development set for early stopping. We then introduce knowledge rules into the objective and continue training with the combined loss. Final model selection is guided by both labeled predicates performance on the development set and the degree of rule consistency over unlabeled instances. We optimize our system with AdamW with weight

decay [16] and learning rate scheduling with warmup and linear decay. Both, optimizer and scheduler are reset before the second stage of training with rules. We applied gradient clipping to stabilize training.

We explore various configurations of the standard transformer encoder architecture by modifying the input embedding dimension, number of encoder layers, attention heads, and feedforward dimensionalities. For each configuration, we further tune learning rates, batch sizes, layer normalization position, and rule coefficients in the declarative loss. We run experiments with three random seeds and report the mean performance in a hold-out test set unless otherwise stated. The best values for these hyperparameters and architectural settings differ across tasks. We describe these, and the datasets and experimental protocols in subsequent sections. Appendix B also gives more details for experiment replication.

Statistical Analysis of Architectural Parameters. To assess the impact of architectural parameters on performance, we group model results by each parameter individually (e.g., embedding dimensionality, number of layers, attention heads, and feedforward dimensionality). Within each parameter, models with identical settings form a distinct group, and we examine performance differences among these groups. Our preliminary analysis revealed that scores for the metrics evaluated across groups are similarly distributed but typically not normally distributed, as confirmed by the Shapiro–Wilk test ($p \leq 0.05$). Therefore, we apply the non-parametric Kruskal–Wallis test to identify significant overall differences in median performance across groups. If a significant difference emerges ($p \leq 0.05$), we perform pairwise comparisons using Dunn's post-hoc test with Bonferroni correction. Significant differences between adjacent parameter sizes are denoted in our results as: * ($p \leq 0.05$), ** ($p \leq 0.01$), and *** ($p \leq 0.001$).

5 Collinearity Experiments

Data Generation. We have four predicates, each representing a different collinearity concept, and we collect a separate dataset for each. We sample points on the real plane uniformly from the square $[-1, 1] \times [-1, 1]$ to construct four datasets: (a) *Collinear3*: Triplets of points, (b) *Collinear4*: Sets of four points, (c) *HasCollinear3*: Sets of five points, and, (d) *HasCollinear4*: sets of five points. Every dataset is split into 20K training examples, 4K development examples, and 4K test examples, balanced with positive and negative labels.

Model and Training. We use the experimental setup described in §4. The first stage of training focuses on `Collinear3` for 100 epochs (with early stopping based on development-set F1). *Stage 2* incorporates rule-based terms for `Collinear4`, `HasCollinear3`, and `HasCollinear4`, continuing training for 30 epochs (with reinitialized learning scheduler and optimizer) under early stopping. The best rule-constrained model is selected using the average between `Collinear3` F1 performance and the proportions of examples satisfying the rules (3), (4), and (5) in §3.1. Importantly, only `Collinear3` has direct supervision.

We conduct a grid search over batch sizes {16, 32, 64}, learning rates {10^{-3}, 10^{-4}, 10^{-5}}, and rule-loss coefficients {5, 1, 0.1, 0.01, 0.001} for each transformer configuration. We study the following architectural variations:

- **Embedding Dimensions**: $\{4, 16, 32, 64, 128\}$,
- **Encoder Layers**: $\{1, 2, 4, 6, 8\}$,
- **Attention Heads**: $\{2, 4, 8, 16\}$,
- **Feedforward Dimensionality**: $\frac{1}{2}\times$, $1\times$, or $2\times$ the embedding size.

This results in 270 distinct configurations. Further details on implementation and batching are in the Appendix C.

Evaluation. We evaluate the jointly trained model on the held-out test sets by reporting the **F1** score for each predicate classifier: `Collinear3`, `Collinear4`, `HasCollinear3`, and `HasCollinear4`. Predictions are generated by applying the corresponding linear head to each point tuple. Since only `Collinear3` is directly supervised, performance on the remaining predicates reflects the model's grasp of the rule-based definitions acquired during training.

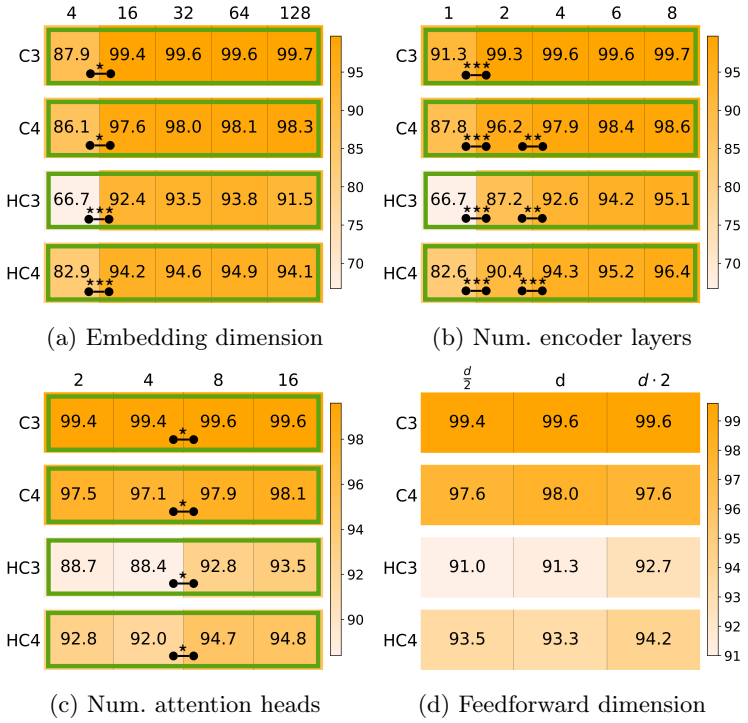
Results. Figure 2 reports median F1 scores for each predicate across the transformer architectures (see §4). We observe a significant improvement in performance when embedding dimensionality increases from 4 to 16 across all predicates, after which performance does not improve with larger embedding dimensions (Figure 2a). Similarly, models with two or more encoder layers significantly outperform single-layer models (Figure 2b). This indicates that the system requires a minimum input dimensionality and encoders to achieve optimal performance; increasing above this threshold does not help.

Configurations with the maximum number of attention heads (16) outperform those with fewer heads, but only slightly so (Figure 2c). Varying feedforward dimensionality do not exhibit statistically significant differences (Figure 2d).

We found that the median baseline F1 for `Collinear3` is 97.1%, based on ground-truth supervision before introducing logical constraints. However, after constrained training, models with limited capacity (embedding dimension 4 and depth 1, Figures 2a and 2b) perform worse than direct supervision. Yet, such models exhibit perfect rule satisfaction, suggesting they converge to spurious solutions that trivially satisfy the constraints.

Figure 3 shows F1 scores for the `HasCollinear3` predicate against model capacity (total parameter count). We observe a performance jump when models reach approximately 16 embedding dimensions and 6 encoder layers. Below this threshold, models struggle to use logical rules effectively. However, once the minimum parameter requirement is met (darker green points), performance stabilizes, with little to no improvement for larger models. Past this threshold, narrow and deep architectures outperform wide and shallow architectures, consistent with previous studies [29,23]. Furthermore, wider and shallower models with higher number of parameters are more susceptible to degenerate solutions, but increased depth mitigates this issue.

We find similar trends for the unsupervised `Collinear4` and `HasCollinear4` predicates. Due to space constraints, the details are in Appendix C.



(a) Embedding dimension (b) Num. encoder layers

(c) Num. attention heads (d) Feedforward dimension

Fig. 2: Median F1 score for each architectural parameter value for the predicates `Collinear3` (C3), `Collinear4` (C4), `HasCollinear3` (HC3), and `HasCollinear4` (HC4). Groups of values statistically different are enclosed in green rectangles, and post-hoc significant adjacent pairs are connected by black segments.

6 Puzzle Tasks

Sudoku and Futoshiki Datasets. For both puzzle tasks, we construct datasets for two learning settings: a *solving* setting, where the model learns to solve a single instance solely through rule interactions; and a *generalization* setting, where the model is trained on a set of puzzles with rule-based supervision and evaluated on its ability to solve unseen instances.

For Sudoku, we use the *4×4 sudoku* Kaggle dataset.[3] We extract 288 puzzles with unique solutions[4] and different difficulty levels: 96 easy puzzles (6–7 hidden cells), 96 medium puzzles (8–9 hidden cells), and 96 hard puzzles (10–12 hidden cells). From this set, we sample 100 puzzles (33 easy, 34 medium, 33 hard) for

[3] https://www.kaggle.com/datasets/redraiment/complete-13-million-4x4-sudoku-puzzles

[4] The total number of distinct 4x4 Sudoku solutions is 288.

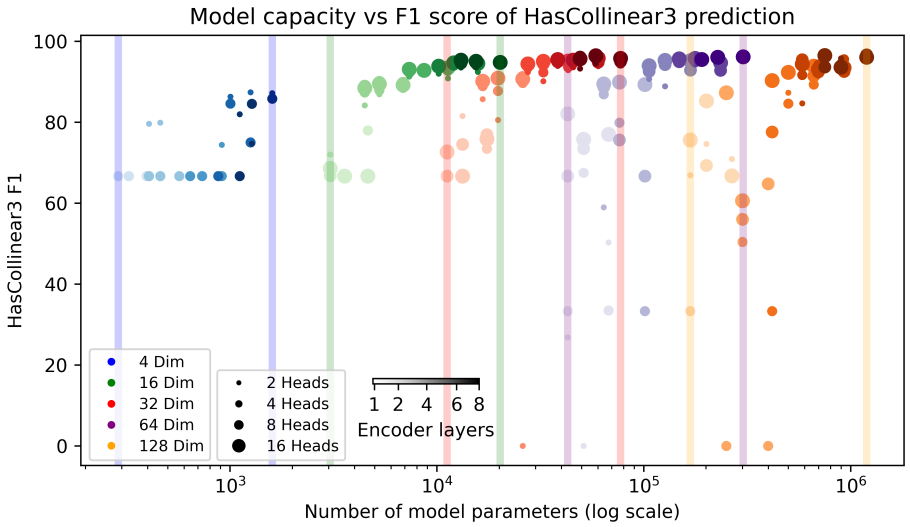


Fig. 3: `HasCollinear3` F1 performance (avg. of three runs) against the total number of model parameters. Each point represents a different transformer configuration, with color denoting the embedding dimensionality, color intensity indicating the number of encoder layers, dot size reflecting the number of attention heads, and feedforward dimensionality increasing from left to right within each color group. The maximum and average standard deviation across runs are 10.9 and 0.8 respectively.

the solving setting. For the generalization setting, the 288 puzzles are split into training (70%), development (15%), and test (15%) sets, stratified by difficulty.

For Futoshiki, we use 5×5 puzzles with unique solutions and 10 inequalities introduced in [19]. We sample 1000 boards spanning different difficulty levels: 345 easy (7–10 hidden cells), 402 medium (11-13 hidden cells), and 253 hard (14–17 hidden cells). For the solving setting, we subsample 100 puzzles (33 easy, 34 medium, 33 hard). As with Sudoku, for the generalization setting, we use a 70/15/15 stratified train/dev/test split.

Each puzzle and its corresponding solution are encoded as a sequence of one-hot vectors corresponding to a digit or an empty cell.

Solving Setting Encoding. Here, we solve a single puzzle instance. The input thus encodes only the given digits and empty cells. All Sudoku constraints and Futoshiki inequalities are enforced via the loss, so no explicit rule encoding is required in the input.

Generalization Setting Encoding. At test time the network receives a new (unseen) puzzle without any external rule module; we expect the solving procedure to be internalized in the model weights. For Sudoku, the input encoding remains

the same, containing only digits and empty cells. For Futoshiki, since inequalities vary across puzzles, we augment the one-hot input with an additional binary encoding marking the positions and orientations of each inequality sign ("$<$" or "$>$"). This extra information enables the model to adjust its reasoning according to puzzle-specific inequality constraints. Further implementation details are provided in Appendix E.

Model and Training. Each input puzzle is represented as a sequence of one-hot encoded vectors. These vectors pass through an embedding layer and we use a 2D sinusoidal positional encoding layer [28]. The resulting sequence is fed into a transformer encoder. Each position in the puzzle grid is assigned a dedicated predicate classifier head that produces the probability distribution of the cell having any of the possible digits in the puzzle (i.e., for Sudoku 4×4 each classifier head produces a probability distribution of four elements modeling the score of the corresponding cell taking values 1–4).

During training, we use puzzles one at a time (batch size 1) to compute the loss with respect to the rule constraints. In the first stage of training, the system learns to "copy" the given cells without constraints. We continue training introducing the rules of the game in the second stage (for 400 and 100 epochs, for solving and generalization settings resp.). We perform a grid search over learning rates $\{10^{-5}, 10^{-4}, 10^{-3}, 10^{-2}\}$ and rule-loss coefficients $\{10^{-5}, 10^{-4}, 10^{-3}, 10^{-2}, 10^{-1}, 1, 5\}$. We explore the following architectural variations:

- **Embedding Dimensions**: $\{16, 64, 256, 1024\}$,
- **Encoder Layers**: $\{1, 2, 6, 12\}$,
- **Attention Heads**: $\{4, 8, 16\}$,
- **Feedforward Dimensionality**: $\frac{1}{2}\times$, $1\times$, or $2\times$ the embedding size.

This results in 144 distinct configurations. Further details on implementation and batching are in Appendix D.

Evaluation. We evaluate each architecture configuration under both *solving* and *generalization* experiments. For the former, we train a model to solve each puzzle and measure the fraction of puzzles fully solved and the proportion of rule components satisfied across all puzzles. For the latter, we train models on puzzles from the train set and evaluate on unseen puzzles in the test set, considering two regimes (cf. §3.2): given-cells only and full-solution supervision. We report the ratio of correctly predicted hidden cells and fully solved puzzles from the test set for both settings, averaged over three random seeds.

6.1 Solving Setting Results

Figure 4 shows that both tasks require a minimum embedding dimensionality to reach near peak performance: 256 for Futoshiki and 64 for Sudoku. Smaller embeddings struggle to fully solve puzzles, but surpassing the threshold triggers a sharp increase in completion rates — a "phase shift" effect. Additional dimensionality yields only limited gains.

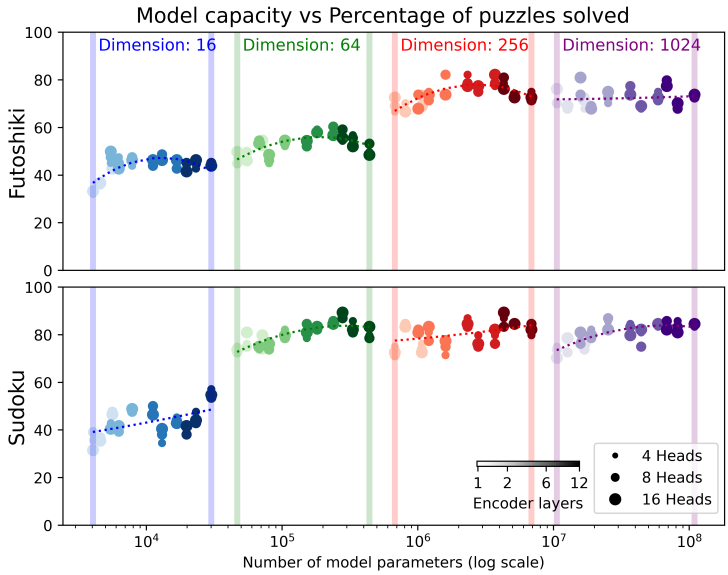


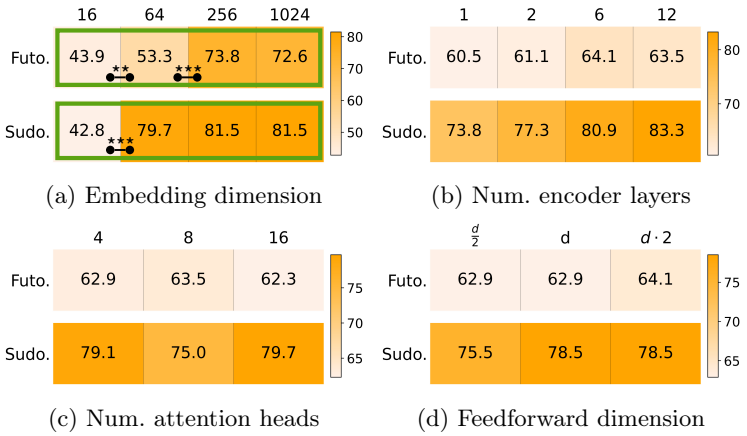
Fig. 4: Solved Futoshiki (top) and Sudoku (bottom) puzzles (three-run average) against number of model parameters. The visual encoding is as in Figure 3. The maximum and average standard deviation across runs are 7.5 and 9.0 and 3.1 and 4.5 for Futoshiki and Sudoku runs respectively.

Figure 5 reports the medians of solved puzzle rates for each architectural parameter value. Below their respective dimensionality thresholds (16 and 64), the performance for Futoshiki and Sudoku models is significantly worse than at or above the threshold, and higher embedding dimensions offer no significant improvements (Figure 5a). No statistically significant differences exist for the other architectural choices (Figures. 5b–5d).

Appendix D presents additional analyses demonstrating that this trend persists when evaluating performance in terms of hidden-cell prediction accuracy. Furthermore, when performance is analyzed separately across different puzzle difficulty levels (easy, medium, and hard), the results consistently indicate that embedding dimensionality remains the most critical architectural parameter.

6.2 Generalization Setting Results

Given-cells-only Supervision. Figure 6 shows the median fraction of correctly predicted hidden cells. For both puzzles, smaller embedding dimensionalities yield the best results. For Futoshiki, increasing dimensionality from 16 to 64 provides a statistically significant improvement of 8.2% (Figure 6a), and network depth also significantly impacts performance: architectures with 12 layers outperform (10%) those with 1 or 2 layers (Figure 6b). For Sudoku, embedding



(a) Embedding dimension (b) Num. encoder layers

(c) Num. attention heads (d) Feedforward dimension

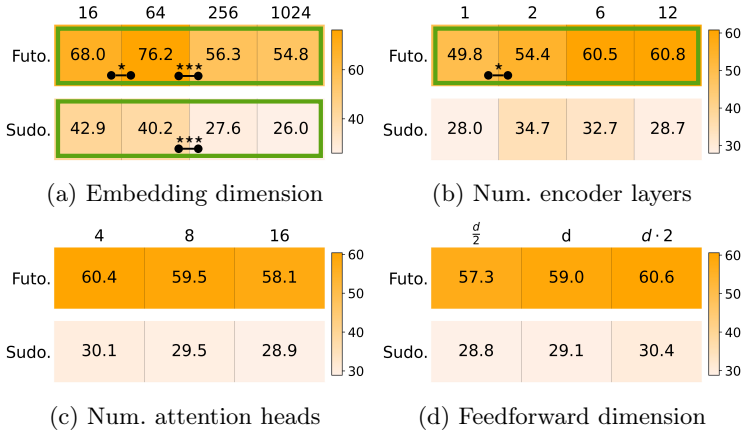
Fig. 5: Solving setting: Median percentage of puzzles solved for each architectural parameter value–Futoshiki (Futo.) above, Sudoku (Sudo.) below. Groups of values statistically different are enclosed in green rectangles, and post-hoc significant contiguous pairs are connected by black segments.

dimensionalities of 16 and 64 outperform the others. Network depth, number of attention heads, and feedforward dimensionality show no significant effects.

Curiously, models with larger embedding dimensionalities are significantly worse. Evaluating them on the training sets reveals clear overfitting. We conjecture that smaller networks more readily lock onto puzzle constraints, leading to faster convergence to consistent solutions. In contrast, larger networks likely have more complex optimization landscapes, are more sensitive to hyperparameter choices, and more prone to convergence to local minima. While our experiments focus on architectural factors, we note that the degree of overfitting may also be influenced by optimization choices such as learning rate, regularization, and stopping criteria—variables we did not systematically ablate in this study.

Figure 7 shows hidden-cell prediction performance as a function of total model parameters. Futoshiki models fare better than the Sudoku, despite the former's reputation as being more difficult. We conjecture that this is the case due to our richer input encoding for Futoshiki puzzles, which incorporates inequality positions. Importantly, these models do not see any labeled data during training, and rely only on the logical rules in the loss function to generalize to unseen puzzles. Yet, some architectural configurations can fully solve up to 75% of Futoshikis and 12% of Sudokus from the test set. Appendix E has more details.

Full-Solution Supervision. For the fully-supervised (full-solution) setting, results follow analogous trends, but exhibit consistently higher performance due to the additional explicit supervision. Here, rules function as auxiliary guidance on top of direct puzzle solutions. Comparing performance before (stage one) and after



(a) Embedding dimension (b) Num. encoder layers

(c) Num. attention heads (d) Feedforward dimension

Fig. 6: Given-cells-only setting: Median percent of correctly predicted hidden cell for each architectural parameter value–Futoshiki (Futo.) above, Sudoku (Sudo.) below. Groups of values statistically different are enclosed in green rectangles, and post-hoc significant contiguous pairs are connected by black segments.

rule incorporation (stage two), we observe consistent improvements due to rules, especially pronounced in deeper networks. Due to space constraints, detailed figures and additional analyses are provided in Appendix E.

7 Discussion

Unlike other work on neuro-symbolic models that seek to improve reasoning performance [15,19,1,31], we do not introduce a model or a framework that surpasses state-of-the-art results. Rather, our goal is to analyze the role and impact of transformer architectural choices when these architectures serve as the underlying neural component in logic-as-loss frameworks.

In this study, we design rule-governed synthetic tasks—such as planar collinearity and Latin square puzzles—where each instance is fully defined by a known set of logical constraints. Success in these tasks depends entirely on the model's ability to adhere to the rules, providing a noise-free setting to analyze how Transformer design choices interact with the *logic-as-loss* objective. While this controlled setup limits direct real-world applicability, it offers a clear lens for studying rule internalization. Extending this analysis to more naturalistic domains, where constraints are only partially known, is an important direction for future work. Additionally, exploring how architectural factors interact with specific classes of logical rules (e.g., monotonic vs. non-monotonic) is an interesting avenue for further research.

Our empirical findings consistently highlight the input embedding dimensionality as a critical architectural parameter, significantly impacting model performance across all tasks (green squares in Figures 2a, 6a, 5a). Specifically, in the

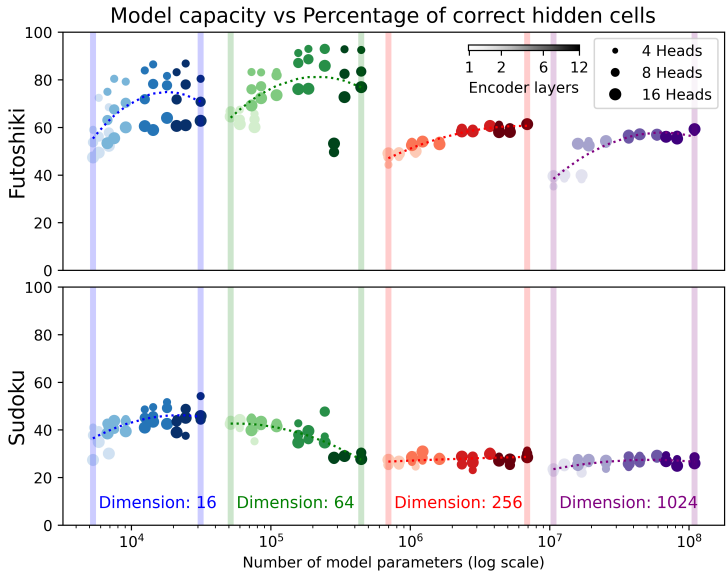


Fig. 7: Given-cells-only setting: Average percent (over three runs) of correctly predicted hidden cells for Futoshiki (top) and Sudoku (bottom), plotted against the total number of model parameters (log scale). The visual encoding is as in Figure 3. The maximum and average standard deviation across runs are 8.1 and 10.0 and 1.1 and 1.5 for Futoshiki and Sudoku experiments respectively.

collinearity task and puzzle-solving scenarios, we identify a clear threshold or "phase shift" effect: performance significantly improves once embedding dimensionality exceeds a critical value, beyond which additional capacity yields no substantial gains. This observation suggests that increasing embedding size beyond a necessary threshold is computationally inefficient without performance benefits. For generalization scenarios, smaller-capacity models appeared to incorporate the training rules more efficiently, often outperforming larger networks, which exhibited higher susceptibility to overfitting and greater sensitivity to hyperparameter tuning.

We further observe that network depth can positively impact performance within fixed embedding dimensions. Particularly, in the collinearity task, deeper networks were less prone to degenerate, trivial solutions. This result motivates further exploration of network depth alongside other strategies from the literature aimed at mitigating spurious or degenerate solutions [11,14].

While our analysis focused specifically on logic-as-loss using t-norm relaxations, prior studies have observed similar learning behaviors in probabilistic logic-relaxation techniques [2,3,8]. Moreover, theoretical results indicate fundamental similarities between model-counting approaches such as semantic loss and t-norm based methods within the logic-as-loss framework [25,24]. Thus, we

anticipate that our architectural insights may generalize broadly across various logic-relaxation methods, although empirical validation remains necessary.

8 Conclusion

We analyzed how architectural choices in transformers influence their ability to incorporate logical constraints using the logic-as-loss framework. We used controlled reasoning tasks — collinearity detection, Sudoku, and Futoshiki — to identify a clear pattern: above a certain embedding dimensionality threshold, further capacity increments provided limited performance improvements. Additionally, deeper networks mitigated degenerate solutions, highlighting the advantage of depth over width for robust rule ingestion. Our findings emphasize the critical role of architectural design in transformer-based logic-as-loss applications.

Acknowledgments. We thank Ana Marasović, Kyle Richardson, Jeff M. Phillips, and members of the UtahNLP group for their valuable insights. We are also grateful to the reviewers for their helpful comments, corrections, and suggestions for related work. This work was partly supported by NSF grants #2217154 and #2411319. Additionally, the support and resources from the Center for High Performance Computing at the University of Utah are gratefully acknowledged.

Disclosure of Interests. The authors have no competing interests to declare that are relevant to the content of this article.

References

1. Ahmed, K., Chang, K.W., Van den Broeck, G.: A pseudo-semantic loss for autoregressive models with logical constraints. In: NeurIPS (2023)
2. Ahmed, K., Chang, K.W., Van den Broeck, G.: Semantic strengthening of neuro-symbolic learning. In: AISTATS (2023)
3. Ahmed, K., Teso, S., Chang, K.W., Van den Broeck, G., Vergari, A.: Semantic probabilistic layers for neuro-symbolic learning. In: Proceedings of the 36th International Conference on Neural Information Processing Systems (2022)
4. Cătălina Stoian, M., Giunchiglia, E., Lukasiewicz, T.: Exploiting T-norms for Deep Learning in Autonomous Driving. arXiv (2024)
5. Defresne, M., Barbe, S., Schiex, T.: Scalable coupling of deep learning with logical reasoning. In: IJCAI (2023)
6. Dosovitskiy, A., Beyer, L., Kolesnikov, A., Weissenborn, D., Zhai, X., Unterthiner, T., Dehghani, M., Minderer, M., Heigold, G., Gelly, S., Uszkoreit, J., Houlsby, N.: An image is worth 16x16 words: Transformers for image recognition at scale. In: ICLR (2021)
7. Flinkow, T., Pearlmutter, B.A., Monahan, R.: Comparing differentiable logics for learning with logical constraints. Science of Computer Programming (2025)
8. Flügel, S., Glauer, M., Mossakowski, T., Neuhaus, F.: A fuzzy loss for ontology classification. In: Neural-Symbolic Learning and Reasoning: 18th International Conference, NeSy 2024, Barcelona, Spain, September 9–12, 2024, Proceedings, Part I (2024)

9. Giunchiglia, E., Stoian, M.C., Lukasiewicz, T.: Deep learning with logical constraints. In: IJCAI (2022)
10. Grespan, M.M., Gupta, A., Srikumar, V.: Evaluating relaxations of logic for neural networks: A comprehensive study. In: IJCAI (2021)
11. He, H.Y., Dai, W.Z., Li, M.: Reduced implication-bias logic loss for neuro-symbolic learning. Machine Learning (2024)
12. Klement, E.P., Mesiar, R., Pap, E.: Triangular Norms, vol. 8. Springer Science & Business Media (2013)
13. Li, T., Gupta, V., Mehta, M., Srikumar, V.: A logic-driven framework for consistency of neural models. In: EMNLP-IJCNLP (2019)
14. Li, Z., Liu, Z., Yao, Y., Xu, J., Chen, T., Ma, X., Lu, J.: Learning with logical constraints but without shortcut satisfaction. In: ICLR (2023)
15. Li, Z., Guo, J., Jiang, Y., Si, X.: Learning reliable logical rules with satnet. In: NeurIPS (2023)
16. Loshchilov, I., Hutter, F.: Decoupled weight decay regularization. In: ICLR (2019)
17. Medina Grespan, M., Broadbent, M., Zhang, X., Axford, K., Kious, B., Imel, Z., Srikumar, V.: Logic-driven indirect supervision: An application to crisis counseling. In: ACL (2023)
18. Nam, A.J., Abdool, M., Maxfield, T., McClelland, J.L.: Achieving and understanding out-of-distribution generalization in systematic reasoning in small-scale transformers. arXiv (2022)
19. Nandwani, Y., Jain, V., Mausam, Singla, P.: Neural models for output-space invariance in combinatorial problems. In: ICLR (2022)
20. OpenAI: Gpt-4 technical report. Tech. rep., OpenAI (2023)
21. Palm, R.B., Paquet, U., Winther, O.: Recurrent relational networks. In: NeurIPS. pp. 3368–3378 (2018)
22. Richardson, K., Srikumar, V., Sabharwal, A.: Understanding the Logic of Direct Preference Alignment through Logic. In: ICML (2025)
23. Tay, Y., Dehghani, M., Rao, J., Fedus, W., Abnar, S., Chung, H.W., Narang, S., Yogatama, D., Vaswani, A., Metzler, D.: Scale efficiently: Insights from pretraining and finetuning transformers. In: ICLR (2022)
24. van Krieken, E., Acar, E., van Harmelen, F.: Analyzing differentiable fuzzy logic operators. Artificial Intelligence (2022)
25. Van Krieken, E., Minervini, P., Ponti, E.M., Vergari, A.: On the independence assumption in neurosymbolic learning. In: Proceedings of the 41st International Conference on Machine Learning (2024)
26. Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A.N., Kaiser, L.u., Polosukhin, I.: Attention is all you need. In: NeurIPS (2017)
27. Wang, H., Chen, M., Zhang, H., Roth, D.: Joint constrained learning for event-event relation extraction. In: EMNLP (2020)
28. Wang, Z., Liu, J.: Translating math formula images to LaTeX sequences using deep neural networks with sequence-level training. IJDAR (2021)
29. Warner, B., Chaffin, A., Clavié, B., Weller, O., Hallström, O., Taghadouini, S., Gallagher, A., Biswas, R., Ladhak, F., Aarsen, T., Cooper, N., Adams, G., Howard, J., Poli, I.: Smarter, better, faster, longer: A modern bidirectional encoder for fast, memory efficient, and long context finetuning and inference. arXiv (2024)
30. Xu, J., Zhang, Z., Friedman, T., Liang, Y., Van den Broeck, G.: A semantic loss function for deep learning with symbolic knowledge. In: ICML (2018)
31. Yang, Z., Ishay, A., Lee, J.: Learning to solve constraint satisfaction problems with recurrent transformer. In: ICLR (2023)