Unlocking Out-of-Distribution Generalization in Transformers via Latent Space Reasoning

Anonymous authors
Paper under double-blind review

Abstract

Systematic, compositional generalization beyond the training distribution remains a core challenge in machine learning—and a critical bottleneck for the emergent reasoning abilities of modern language models. This work investigates out-of-distribution (OOD) generalization in Transformer networks using a GSM8K-style modular arithmetic on computational graphs task as a testbed. We introduce and explore a set of four architectural mechanisms aimed at enhancing OOD algorithmic generalization: *(i)* input-adaptive recurrence; *(ii)* algorithmic supervision; *(iii)* anchored latent representations via a discrete bottleneck; and *(iv)* an explicit error-correction mechanism. Collectively, these mechanisms yield an architectural approach for native and scalable latent space reasoning in Transformer networks with robust algorithmic generalization capabilities. We complement these empirical results with a detailed mechanistic interpretability analysis that reveals how these mechanisms give rise to robust OOD generalization abilities.

1 Introduction

Systematic algorithmic generalization stands as a critical milestone and a grand challenge in machine learning research [1–4]. This ability is fundamental to human cognition, stemming from our capacity for *systematic compositionality*—algebraically producing novel combinations from known components and making strong generalizations from limited data [5–7]. Achieving such generalization necessitates learning universal, scalable problem-solving algorithms. Even in humans, acquiring such algorithmic understanding often requires explicit step-by-step supervision. Once an algorithm is learned, however, humans can generalize its application far beyond the domain of previously encountered stimuli or problems [8, 9].

The reasoning capabilities of artificial intelligence systems have advanced rapidly in recent years, built upon the foundation of large language models. In particular, chain-of-thought (CoT) techniques have been central to enhancing these capabilities [10–13], especially in domains like mathematics [14–17]. CoT enables a model to receive supervision on learning a reference problem-solving procedure during training and allows the model to emulate this procedure at test-time. This progress presents a unique opportunity to make significant strides on foundational challenges related to reasoning in artificial intelligence.

However, despite these advancements, out-of-distribution (OOD) generalization—particularly the type of *length generalization* involved in algorithmic reasoning (i.e., generalizing from simpler or smaller problem instances to larger or more complex ones)—has remained a central challenge and limitation for Transformer-based [18] language models [19–24]. While chain-of-thought techniques alleviate this to some degree by enabling the learning of more complex algorithmic procedures, the ability to generalize far outside the training distribution remains a significant obstacle [25, 26].

In this work, we investigate the architectural and methodological mechanisms that underpin algorithmic OOD generalization in Transformer networks. To facilitate a systematic investigation, we focus our study on a simple yet scalable mathematical reasoning task: performing modular arithmetic on computational graphs. This task allows us to study OOD and algorithmic generalization in a controlled manner—with complexity directly parameterized by graph size and depth—while also capturing the core essence of established mathematical reasoning benchmarks like GSM8K [14], which are central to evaluating the reasoning capabilities of large language models. Furthermore, this task possesses a compositional nature; it

can be solved by learning a core set of skills and scaling up their application to solve larger and more complex problem instances. We use this task to explore the following guiding question: *What are the architectural mechanisms and inductive biases needed for robust and systematic OOD algorithmic generalization in Transformers?*

We find that while standard CoT training techniques enable good in-distribution performance and a limited degree of OOD generalization, the learned solutions are not robust or universal, and their performance rapidly degrades as test inputs grow in complexity beyond the training regime. We propose and explore a set of four simple architectural and methodological mechanisms, built upon the Transformer architecture, to facilitate the learning of robust and generalizable algorithmic solutions: *(i)* input-adaptive recurrence; *(ii)* algorithmic supervision; *(iii)* anchored latent representations via a discrete bottleneck; and *(iv)* an explicit error-correction mechanism. When combined, these mechanisms yield an architectural approach for native and scalable *latent space reasoning* in Transformer networks, demonstrating robust algorithmic generalization capabilities. In particular, on our mathematical reasoning task, our method achieves perfect generalization on inputs that are several times larger than those seen during training. We complement our architectural proposal and empirical results with a mechanistic interpretability analysis to reveal *how* these architectural proposals enable sharp OOD generalization, what circuits they learn, and why those circuits facilitate robust OOD generalization.

Related Work. This work is related to the literature on out-of-distribution generalization, chain-of-thought, and Transformer architectures. We will mention related work as we develop our methods, and provide a dedicated detailed discussion in Section A.

2 Problem Setup

2.1 Task Description: Modular Arithmetic on Computational Graphs

We formally introduce the task of *modular arithmetic on computational graphs* as follows.

Task Description. A *computation graph* is a directed acyclic graph (DAG) representing a network of mathematical computations, where nodes correspond to variables and edges describe the dependencies between them. As illustrated in Figure 1 with an example, the *leaf nodes* in this DAG are directly assigned numerical values (e.g., $x_7 \leftarrow 20$). All other *non-leaf nodes* are defined as functions of their parent nodes in the computation graph. In particular, the value of each non-leaf node is computed by applying one or more specified operations to the values of its parent nodes. In our experiments, we consider *modular arithmetic* operations (addition, multiplication, or subtraction), with the prime number $p = 23$ as the modular base. For example, in Figure 1 we have $x_{23} \leftarrow x_7 + x_{42} \pmod p$ and $x_{101} \leftarrow x_{23} \times x_{91} \pmod p$. In the following, we let N and L denote the total number of nodes and the number of leaf nodes, respectively. We consider graphs with up to 128 nodes, and let $\mathcal{V} = \{x_1, \dots, x_{128}\}$ denote the set of variable names.

Data Generation Process. A problem instance in this task is specified by the *values of the leaf nodes* and *a computation graph* depicting the computations that determine the values of all non-leaf nodes. In particular, given parameters N and L, an input instance is generated as follows:

(i) Randomly generate a DAG with N nodes, L of which are leaf nodes.
(ii) Randomly assign a variable name from $\mathcal{V}$ to each node.
(iii) Randomly assign numerical values to the leaf nodes from $\mathcal{N} = \{0, 1, \dots, 22\}$.
(iv) For each non-leaf node, randomly assign operations from $\mathcal{O} = \{+, -, \times\}$ to define its computation based on its parent nodes

The instance generated by (i)–(iv) is stored as a token sequence, where each variable name, numerical value, and operation is assigned a unique token. A special *separation token* [sep] is used to separate different formulas. For example, the instance depicted in Figure 1 is represented as the following *token sequence*:

$$\begin{aligned}
&\langle 20\rangle\langle\rightarrow\rangle\langle x_7\rangle\ [\texttt{sep}]\ \langle 2\rangle\langle\rightarrow\rangle\langle x_{42}\rangle\ [\texttt{sep}]\ \langle 6\rangle\langle\rightarrow\rangle\langle x_{88}\rangle\ [\texttt{sep}]\ \langle 14\rangle\langle\rightarrow\rangle\langle x_{115}\rangle\\
&\langle x_7\rangle\langle +\rangle\langle x_{42}\rangle\langle\rightarrow\rangle\langle x_{23}\rangle\ [\texttt{sep}]\ \langle x_{42}\rangle\langle +\rangle\langle x_{88}\rangle\langle\rightarrow\rangle\langle x_{91}\rangle\ [\texttt{sep}]\ \langle x_{88}\rangle\langle\times\rangle\langle x_{115}\rangle\langle\rightarrow\rangle\langle x_{55}\rangle\\
&\langle x_{23}\rangle\langle\times\rangle\langle x_{91}\rangle\langle\rightarrow\rangle\langle x_{101}\rangle\ [\texttt{sep}]\ \langle x_{91}\rangle\langle -\rangle\langle x_{88}\rangle\langle +\rangle\langle x_{55}\rangle\langle\rightarrow\rangle\langle x_{30}\rangle
\end{aligned} \tag{1}$$

Target Output & Evaluation Metric. Given a generated problem instance, the task is to compute the value of every node in the computation graph; these values are uniquely determined by steps (i)–(iv) above. We consider the model output to be correct only if *all* node values are computed correctly (i.e., the input graph is fully solved).

Out-of-Distribution Generalization. Our primary focus in this work is to investigate the ability of Transformer networks to learn general problem-solving procedures or algorithms that enable *out-of-distribution* (OOD) generalization. The complexity of each problem instance can be explicitly parameterized by graph size, enabling precise measurement of a model's ability to generalize to inputs more complex than those encountered during training. In particular, in this mathematical reasoning task, OOD generalization is evaluated by training Transformer models on problem instances with $N \leq 32$ nodes and testing them on instances of varying sizes, up to $N = 128$ (a fourfold increase). Such generalization requires the ability to process larger inputs and adaptively scale computation time during testing, beyond what was encountered in the training regime.

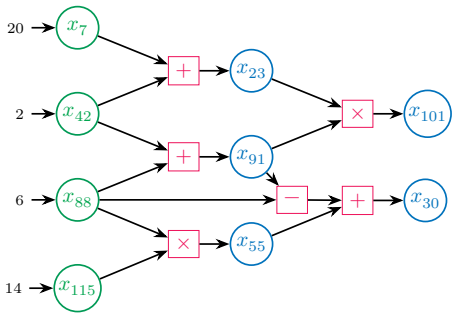


Figure 1. An illustration of a problem instance. The goal is to compute the values of all nodes in the graph. For example, here, $x_{23} = 20 + 2 = 22$ and $x_{55} = 6 \times 14 = 15$.

Representativeness and Generality of the Computation-Graph Task. In addition to providing fine-grained control over instance complexity, this computation-graph formulation captures structural properties shared by a broad class of algorithmic reasoning tasks. Any program over a finite primitive operation set can be represented as a directed acyclic computation graph in which nodes encode operations and edges specify data dependencies. Depth in such graphs corresponds to the number of sequential computational steps required to solve an instance, making depth extrapolation a direct proxy for the central challenge in algorithmic reasoning: maintaining stable computation as compositional depth grows. Although our instantiation uses modular arithmetic as the primitive operations, the underlying abstraction—symbolic computation over DAGs—is general. Moreover, this synthetic task retains the essential combinatorial structure of mathematical-reasoning benchmarks such as GSM8K [14], while eliminating incidental complexities of natural language and enabling precise analysis of the learned computation, as explored in Section 4.

2.2 Limitations of Standard Transformers with CoT Training

To establish a baseline and motivate the need for alternative approaches, we evaluate standard Transformer architectures on our synthetic task using two primary training paradigms.

End-to-End Training. The first baseline is *End-to-End* training, where the Transformer models are trained to directly output the final values of all nodes given the problem input, without explicit intermediate steps. The input token sequences are in the form of (1), and we employ various Transformer models with diverse architectures. See Section B for details.

Chain-of-Thought (CoT) Training. The second baseline is based on autoregressive *Chain-of-Thought (CoT)* training [10, 14, 15, 27, 28], a prevalent technique for enabling multi-step reasoning in LLMs. Instead of directly outputting the final answer, CoT trains a model to generate a sequence of intermediate reasoning steps (the "thought process") that culminates in the solution. For our task, CoT intermediate steps consist of explicit demonstrations of the step-by-step computation of nodes within a given computation graph. In particular, in *CoT* training, the Transformer model receives an input prompt consisting of the token representation of the computation graph (as in (1)), followed by a special ⟨CoT⟩ token. This special token signals the beginning of the CoT reasoning, which outlines the computation of each node in topological order. Each step in the trajectory involves: (1) recalling the equation defining the node's value, (2) recalling the values of its dependent nodes, and (3) performing the arithmetic computation. For example, computing node $\langle x_{101} \rangle$ from Figure 1 would appear in the CoT as:

$$[\ldots\texttt{Input Prompt}\ldots]\langle\texttt{CoT}\rangle[\ldots]\langle x_{101}\rangle = \langle x_{23}\rangle\,\langle\times\rangle\,\langle x_{91}\rangle = \langle 22\rangle\,\langle\times\rangle\,\langle 8\rangle = \langle 15\rangle$$

Here, the [...Input Prompt...] gives the description of the problem instance, and [...] denotes the preceding portion of the chain-of-thought trajectory up to node $\langle x_{91} \rangle$, which in particular includes the computation of the values of $\langle x_{23} \rangle$ and $\langle x_{91} \rangle$. An example of a full CoT example from the training data is provided in Section B.2.

Implementation. We train causal Transformer models from scratch using both *End-to-End* and *CoT* supervision on randomly generated problem instances with graph sizes $N \leq 32$. At inference time, models are prompted with the input and generation is performed using *greedy decoding*. End-to-End models directly output all node values given the input, while CoT models autoregressively generate the solution, including the full CoT trajectory. We evaluate performance based on the proportion of instances where the model computes *all* node values correctly, with a particular focus on OOD generalization to new, randomly generated graphs of varying sizes up to $N = 128$. For each method, we perform an extensive hyperparameter search (including layers, model dimension, and positional encoding) to ensure that we compare against the best-achievable performance for each method. For example, prior work observed that positional encoding methods and other hyperparameter choices can have a significant impact on out-of-distribution generalization performance [20, 29]. A detailed experimental setup for these baseline experiments is provided in Section B.

Observed OOD Generalization Deficiencies. We find that *Chain-of-Thought* training enables models to solve larger graphs compared to those trained *End-to-End* without chain-of-thought supervision (Figure 3). While the best-performing CoT models exhibit *a limited degree of OOD generalization* to moderately larger graphs ($N \leq 32 \rightsquigarrow N \approx 40$), this capability rapidly deteriorates as graph sizes exceed the training regime. In the next section, we propose a series of architectural mechanisms that address these generalization challenges.

3 Reasoning in Latent Space with Algorithmic Supervision

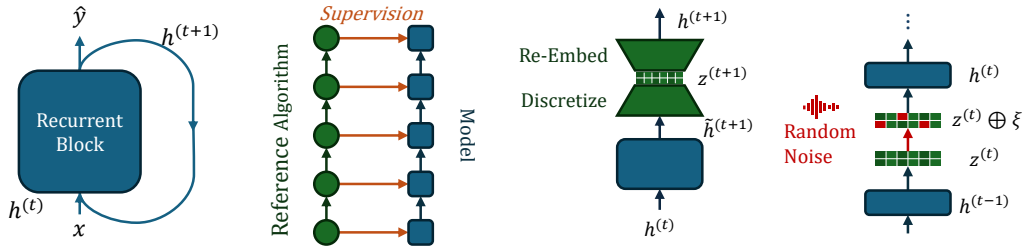
3.1 Mechanisms for Effective OOD Generalization.

Effective OOD generalization on complex reasoning tasks hinges on a model's ability to learn and emulate an underlying scalable *algorithm*. This requires the model to, implicitly or explicitly, execute an iterative procedure that adapts to input complexity. Designing *inductive biases* to support the discovery of such scalable, compositional solutions is a central challenge in machine learning [3, 30–32]. Chain-of-thought (CoT) techniques attempt this by having the model sequentially generate a token representation of a computational process. However, this restriction to a token-based, autoregressive format often yields brittle "algorithms" that fail to generalize robustly, especially as longer CoT sequences are needed for more complex inputs. These well-documented length generalization issues [19–22, 25, 26] underscore CoT's limitations in effectively emulating truly scalable algorithmic procedures. This work, therefore, proposes alternative mechanisms to facilitate the learning of such iterative algorithms directly within a model's latent processing.

Our proposal features recurrent Transformer blocks, algorithmic supervision, discretization in latent space, and a self-correction scheme (depicted in Figure 2). Collectively, these mechanisms constitute an architecture enabling native latent-space reasoning, leading to effective OOD generalization. In the following, we present the four proposed mechanisms and the essence of their implementation, deferring certain implementation details to Section C.

Algorithm to Emulate. To solve this task, a natural algorithmic solution that is well-aligned with the Transformer architecture is to compute the values in the computation graph one layer at a time. This can be realized through a recursive process that iteratively applies the same computational modules. Specifically, each iteration of the algorithm computes values one layer deeper in the computation graph by fetching the necessary dependent values for nodes at the current layer and then performing the required modular arithmetic. In particular, for the example in Figure 1, in the first iteration, we evaluate variables $\{x_7, x_{42}, x_{88}, x_{115}\}$. In the second iteration, we evaluate $\{x_{23}, x_{91}, x_{55}\}$. In the last iteration, we evaluate $\{x_{101}, x_{30}\}$. Note that each iteration involves the same type of computation, providing a succinct and scalable recursive problem-solving algorithm.

Mechanism 1: Recurrence & Input-Adaptive Computation. The iterative and recursive structure of the target layer-by-layer algorithm naturally motivates a recurrent architecture. We employ a recurrent Transformer block [33] with the goal that each application emulates one algorithmic iteration—that is, computing values for one additional layer of the computation graph. An input instance is represented as a sequence of n tokens



(a) Mechanism 1: Recurrence & Adaptive Computation **(b)** Mechanism 2: Algorithmic Supervision **(c)** Mechanism 3: Anchored Discrete Latent Space **(d)** Mechanism 4: Error Correction

Figure 2. A Depiction of the proposed mechanisms for out-of-distribution generalization.

$X = (x_1, \dots, x_n)$, as described in (1). This is embedded to form a sequence of embedding vectors $E_1^{(0)}, \dots, E_n^{(0)}$, and recurrently processed with the recurrent transformer block

$$(E_1^{(t+1)}, \dots, E_n^{(t+1)}) \leftarrow \text{RecurrentTransformerBlock}(E_1^{(t)}, \dots, E_n^{(t)}), \; t = 1, 2, \dots, T. \tag{2}$$

The output is linearly read out from the final embedding states $E_1^{(T)}, \dots, E_n^{(T)}$. Crucially, the number of recurrent iterations, T, is not fixed but adapts to input complexity, scaling linearly with the depth of the computation graph. This input-adaptive recurrence allows the model to dynamically scale its computation time proportionate to the problem's requirements, a key capability for OOD generalization to larger graphs. Unlike CoT methods that scale computation by generating progressively longer linear sequences of tokens, recurrence introduces inductive biases favoring recursive solution structures, which are inherently more scalable. The use of recurrence to adaptively scale computation time is a well-established concept for tackling tasks with variable complexity [33–39].

Mechanism 2: Latent State Algorithmic Supervision. While recurrence provides the capacity for iterative computation, it does not inherently guarantee that the model will learn the desired layer-by-layer algorithmic procedure. To instill this structure, we introduce latent state algorithmic supervision. Unlike CoT, which supervises intermediate computation in token space, our mechanism provides supervision directly within the model's latent representation space at each recurrent step, steering the internal states to align with the step-by-step execution of our target algorithm. Specifically, at each recurrent iteration t, a shared *linear readout layer* is used to predict node values from their current latent embeddings $E_i^{(t)}$. The training loss applied to these predictions at each recurrent iteration aligns the model with the target layer-by-layer algorithm, taking the form:

$$\text{AlgorithmAlignmentLoss} = \sum_{t=1}^{T} \sum_{i \in [n]} \mathbf{1}\{\text{Depth}(x_i) \leq t\} \cdot \ell\left(W_{\texttt{value}} E_i^{(t)}, \text{Value}(x_i)\right), \tag{3}$$

where $\text{Depth}(x_i)$ is the node's depth in the computation graph, $\text{Value}(x_i)$ is its ground-truth value, and ℓ is the cross-entropy loss. Thus, the algorithm alignment loss supervises the model such that at iteration t, it computes the values of all nodes in the input at computational depth less than or equal to t. For example, in Figure 1, supervision at $t = 1$ applies to leaf nodes (e.g., x_7), while at $t = 2$ it extends to include second-layer nodes (e.g., x_{23}), and so on. This iterative supervision encourages the model to progressively build up the solution, computing the graph one effective layer deeper with each recurrent step.

We note that the only information required to implement the AlgorithmAlignmentLoss beyond the end-to-end label is the depth of each node in the graph, signifying the order in which nodes ought to be computed. This is a weaker supervision oracle than the CoT baseline, which requires traces that decompose each variable's computation into selecting parent variables, retrieving their values, and performing the corresponding arithmetic operation.

Mechanism 3: Anchoring Latent Representation via Discretization. Recurrent models can suffer from representational drift across recurrent iterations during extended out-of-distribution computation, arising from error accumulation when computation scales beyond the training regime. To mitigate this and ensure stable processing across many

iterations, we introduce a discretization mechanism that *anchors* the model's latent representation as computation scales through recurrence. Specifically, after each iteration, the model's continuous hidden states are projected into a structured discrete symbolic space and then immediately re-embedded to form the input for the next recurrent step. This makes it so that the input at each iteration lies in the same structured space, despite scaling computation beyond the training regime.

We implement this anchoring using a structured tokenization and embedding scheme, enabling each token's internal state to evolve recurrently while remaining grounded in a shared discrete space. In our task, the discrete latent space is structured as a product of four factors: token syntax, variable identity, numerical value, and operation type. Please refer to Section C.1 for a concrete example. This yields a latent representation that is discrete, shared across steps, and scalable to extended computation. To map the discrete states to distributed embeddings, we train separate embeddings for each factor and combine the factor embeddings by summation. At each iteration, we first apply the RecurrentTransformerBlock, as in Equation (2), forming the core computation of the recurrent step. The processed distributed representations are then discretized via argmax decoding across each symbolic factor, projecting the latent representation to a common structured space. We then re-embed the discrete state to form the vectorized input for the next iteration.

$$
\begin{aligned}
(\tilde{E}_1^{(t+1)}, \dots, \tilde{E}_n^{(t+1)}) &\leftarrow \text{RecurrentTransformerBlock}(E_1^{(t)}, \dots, E_n^{(t)}) \\
z_{i,\texttt{factor}}^{(t+1)} &\leftarrow \arg\max\{W_{\texttt{factor}} \tilde{E}_i^{(t+1)}\} \quad \text{for each } \texttt{factor} \in \mathcal{F} \\
E_{i,\texttt{factor}}^{(t+1)} &\leftarrow \text{FactorEmbed}(z_{i,\texttt{factor}}^{(t+1)}) \quad \text{for each } \texttt{factor} \in \mathcal{F} \\
E_i^{(t+1)} &\leftarrow E_{i,\texttt{syntax}}^{(t+1)} + E_{i,\texttt{variable}}^{(t+1)} + E_{i,\texttt{operation}}^{(t+1)} + E_{i,\texttt{value}}^{(t+1)}.
\end{aligned}
\tag{4}
$$

The use of discrete latent states combines with Mechanism 1, input-adaptive recurrence, to enable a simple and natural mechanism for determining the number of iterations. The model continues iterating until it reaches a *fixed point* satisfying $\mathbf{z}^{(t+1)} = \mathbf{z}^{(t)}$. Because the model is recurrent and the latent states take values in a discrete space, reaching such a state implies $\mathbf{z}^{(t+k)} = \mathbf{z}^{(t)}$ for all $k \geq 0$, which avoids the "overthinking" behavior that has been observed in recurrent models in prior work, where performance degrades when iterating far beyond the training regime [e.g., 37]. Moreover, discreteness allows a direct and robust halting criterion based on *exact* equality of successive latent states, offering a simpler and more stable criterion than prior learned-halting approaches [e.g., 34, 35].

Mechanism 4: Learning to Self-Correct. Finally, we introduce a self-correction scheme to enhance the robustness of the learned algorithm, especially as the number of computational steps increases, which makes the process more susceptible to error propagation. This mechanism aims to equip the model with the ability to recover from such intermediate mistakes. During training, we inject small, structured perturbations into the latent state (e.g., corrupting a subset of node values or intermediate states) and train the recurrent computation to repair them, encouraging attractor-like, self-correcting dynamics. Specifically, at each recurrent iteration, with a small probability, we randomly corrupt a selection of the value components within the model's discrete latent states. This training regimen forces the model to learn to detect when a previously computed value is incorrect (due to our induced corruption or its own misstep) and then to correct this error in a subsequent computational step before proceeding with the task.

3.2 Experimental Results & Discussion

Combining these mechanisms yields an architecture capable of effectively generalizing far beyond the training distribution to much larger and more complex inputs. To evaluate the effects of the different mechanisms we propose, we study a collection of methods, each implementing a different subset of these mechanisms. These methods are listed in Table 1.

Enabling Robust Algorithmic OOD Generalization. Figure 3 depicts the OOD generalization performance of our methods, ablating across the ingredients described above, as well as the aforementioned *Chain-of-Thought* and *End-to-End* baselines. As previously mentioned, we find that the *End-to-End* models (both recurrent and feedforward) fail to effectively learn the task (with respect to our stringent "fully solved" metric) beyond small graph sizes, even in-distribution. The recurrent models slightly outperform the feedforward

Table 1. *Guide to Implementation of Proposed Mechanisms in Baselines.* The leftmost column lists the method names, matching the figure legends. ● indicates that a mechanism is implemented, ○ means it is not, and ◐ signifies partial implementation.

Method / Mechanism	*Mechanism 1*	*Mechanism 2*	*Mechanism 3*	*Mechanism 4*
Feedforward End-to-End	○	○	○	○
Recurrent End-to-End	◐	○	○	○
Chain-of-Thought	◐	◐	○	○
Continuous Latent Space Supervision	●	●	○	○
Discrete Latent Space Supervision	●	●	●	○
Discrete Latent Space Supervision ↺	●	●	●	●

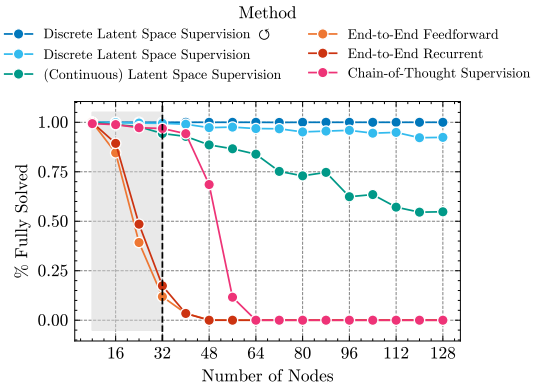


Figure 3. Out-of-Distribution generalization performance of different methods on the mathematical reasoning task.

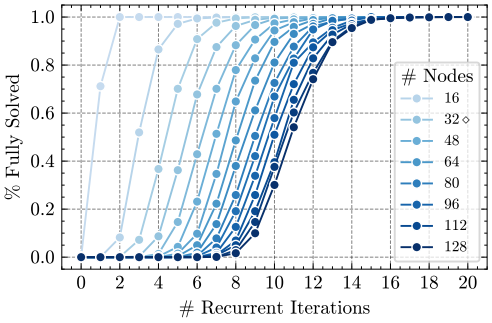


Figure 4. Effective out-of-distribution generalization via input-adaptive scaling of computation time. This depicts *Discrete Latent Space Supervision* ↺

models. *Chain-of-Thought* supervision enables a significant improvement, yielding near-perfect performance in-distribution ($N \leq 32$), and a limited degree of out-of-distribution generalization. To assess our proposed mechanisms for robust OOD generalization in Transformers, we evaluate three classes of models incorporating different subsets of those ingredients. We find that this enables a dramatic improvement in OOD generalization, with performance improving further as more ingredients are incorporated. When all proposed ingredients are incorporated, i.e., *Discrete Latent Space Supervision* ↺[1], the model robustly achieves *near-perfect performance* across all OOD splits we examined.

The Importance of Anchored Discrete Representations. In Figure 3, *Continuous Latent Space Supervision* denotes a recurrent model where the continuous latent states receive step-by-step algorithmic supervision, but the latent states are not discretized in between recurrent block iterations as they are in *Discrete Latent Space Supervision*. We see that, while this outperforms the *Chain-of-Thought* baseline, which is limited to linear reasoning paths, its out-of-distribution performance slowly degrades as we test on progressively larger inputs, which require increasing recurrent depth and computation time. We attribute this to accumulating noise in the continuous vector representations—a phenomenon exacerbated when scaling test-time compute for larger problem instances—which eventually causes representations to drift from the semantically meaningful manifold learned during training. In *Discrete Latent Space Supervision*, the model receives step-by-step algorithmic supervision as with its continuous counterpart, but now we additionally discretize the latent representation, then re-embed using a common embedder that is shared across recurrent iterations. This has the effect of "anchoring" the latent states to a common, semantically-consistent representation space, allowing the model to scale up computational depth without accumulating noise. We observe that this yields significantly improved OOD generalization.

Error-Correction Leads to Greater Robustness in Scaling. In *Discrete Latent Space Supervision* ↺, we introduce explicit supervision for error correction by randomly corrupting the model's latent space with some small probability during training. While the model may make occasional errors, it is able to correct them in the next recurrent iteration, thereby yielding near-perfect OOD generalization. Interestingly, we find that error correction re-

[1]Here, ↺ denotes self-correction.

quires more layers in the recurrent block in order to succeed. An intuitive explanation is that error correction requires greater computational depth *per step*: the model must first identify and correct errors from prior steps before executing the current step's computation.

Robust Test-time Scaling. On many tasks, the computation time required to solve a problem instance is proportional to its size or complexity. Consequently, solving problems larger than those encountered during training necessitates scaling computation time beyond the training regime. In our setting, where the model's reasoning process is latent, we achieve this by increasing the number of recurrent iterations. Figure 4 depicts the proportion of input instances solved as a function of the number of recurrent iterations. Increasing the number of iterations enables solving incrementally larger and harder problem instances. Our architectural mechanisms enable this robust scaling beyond the training regime.

Details, Extensions & Further Ablations. In the appendices, we provide further discussion and present additional experimental results. Here, we briefly highlight a few aspects of these extensions. Across all methods, we find that hyperparameter choice can be critical. In particular, we find that the choice of positional encoding and model depth is especially important. In the above results, we always report the best model within each method after a hyperparameter search, the details of which are provided in the appendix. Additionally, for the chain-of-thought baselines, we explore multiple schemes for the design of the reasoning chains and present the best results here.

Now that we have demonstrated the effectiveness of the proposed architectural mechanisms for robust OOD generalization, we next conduct a mechanistic interpretability analysis to probe the precise computational circuits learned by each component of our model.

4 Mechanistic Interpretability

In this section, we aim to answer the following questions via a detailed study of the model's inner workings: *(i)* What algorithm does the trained model implement? *(ii)* Why is the trained model able to generalize to OOD data? To answer these questions, we first propose hypotheses on the functionality of each model block: first-layer attention, second-layer attention, and the final MLP. For each of these hypotheses, we conduct controlled experiments where we apply causal interventions to specific parts of the input and isolate the effect on model activations to identify the function of each component. Our methodology builds on prior work on causal interpretability in neural networks [40–42], but is tailored specifically to interpreting *recurrent* transformer models. We provide complete details of our experimental methodology in the appendix.

Induction Head & Modular Addition Mechanism

To understand the algorithm implemented by the trained model, we analyze in detail the recurrent Transformer model trained with our proposed *Discrete Latent Space Supervision* method on the mathematical reasoning task. The recurrent Transformer model is configured with two layers, 16 attention heads, and a hidden state dimension of 256. For more details on the model configuration, please refer to Section E. We summarize our mechanism analysis results in Figure 5, where we reveal an induction head mechanism operating within the two-layer attention block and a modular addition mechanism in the final feedforward layer. To better understand the model's behavior, let us take an example equation

$$[\texttt{sep}]\ \langle\texttt{var0}\rangle\ \langle+\rangle\ \langle\texttt{var1}\rangle\ \langle+\rangle\ \langle\texttt{var2}\rangle\ \langle=\rangle\ \langle\texttt{rhs}\rangle\,.$$

We can break down the model's computation into three main components at the Right-Hand Side (RHS) position: The first layer attention heads copy the "`variable`" factored embeddings of variables $\langle\texttt{var0}\rangle$, $\langle\texttt{var1}\rangle$, and $\langle\texttt{var2}\rangle$ to the RHS position, which let the model know the variable names at the RHS position. The second layer attention heads use the copied variable names to retrieve the computed values of variables $\langle\texttt{var0}\rangle$, $\langle\texttt{var1}\rangle$, and $\langle\texttt{var2}\rangle$ from the previous equations through an induction-head mechanism. The last feedforward layer computes the sum of the values of the variables on the LHS and outputs the result to the RHS position.

First Layer Attention Performs Variable Copying. The attention heads in the first layer are grouped by the variable position they attend to, reflecting an attention pattern that is dependent on relative position, as illustrated in Figure 6 (left). For the token embeddings of $\langle\texttt{var0}\rangle$, $\langle\texttt{var1}\rangle$, and $\langle\texttt{var2}\rangle$, which comprise four separate factored embedding types (`syntax`, `variable`, `operation`, and `value`), the value and output projection matrices of

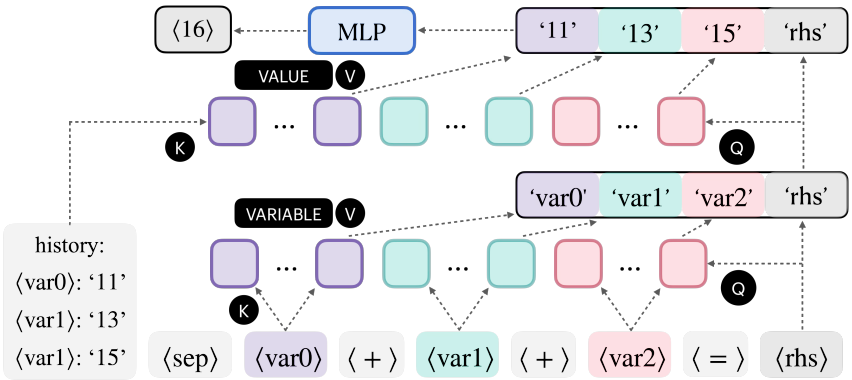


Figure 5. Illustration of the two-layer model performing the modular addition task. The colored squares represent attention heads, grouped by the variable positions they attend to. Black rectangles indicate the embedding components chosen by the value projection matrix. ⟨·⟩ denotes tokens, and '·' denotes embedding components.

each head group *select a subspace* of these token embeddings containing only the `variable` embeddings. This is evident in Figure 6 (right), which plots the norm amplification for different factored embedding types. More details on the norm amplification calculation can be found in Section E. This shows that the first layer attention copies the *variable names* of its parents, which will later be used to obtain their values in the second layer.

Second Layer Attention Implements Variable-Dependent Induction Head Mechanism. The second layer's attention heads then retrieve the corresponding values of variables ⟨`var0`⟩, ⟨`var1`⟩, and ⟨`var2`⟩ from the previous equations through an induction-head mechanism [43]. Specifically, all the attention heads are also grouped by which variable value they are retrieving. For example, let us suppose that the first head group is responsible for retrieving the value of ⟨`var0`⟩. Then, the attention heads within this group will find the first occurrence of ⟨`var0`⟩, which will be the RHS of some previous equation. This particular position is the first time the value of ⟨`var0`⟩ is computed. And these attention heads will then copy the "`value`" factored embedding of ⟨`var0`⟩ also to the current RHS position. In summary, the variable names copied in the first layer are used as queries to retrieve these variables' values, searching over the RHS of previous equations.

Feedforward Layer Performs Modular Addition. The last feedforward layer implements a modular addition mechanism, where the model computes the sum of the values of the variables on the LHS and outputs the result to the RHS position. There are many works that have studied how the feedforward layer implements a modular addition mechanism in the context of Transformer networks [44–46] using a Fourier-based approach. We provide additional evidence for this mechanism in Section E. However, as this is not the main focus of our work, we refer interested readers to above mentioned works for more details.

OOD Generalization of the Trained Model. The model's robust OOD generalization stems from its architectural mechanisms, which guide it towards learning a universal and scalable algorithm. In particular, the algorithm implements a variable-dependent induction head mechanism that is invariant to length, leveraging both relative-positional and variable-dependent attention patterns, which enables the model to operate over contexts of arbitrary lengths. Thus, despite being trained on graphs with limited size, the input-adaptive recurrence, intermediate supervision, and discretization mechanisms enable the model to learn a scalable algorithm capable of solving problems of increased complexity.

5 Discussion

5.1 Architectural ingredients for depth-invariant algorithmic generalization

This work identifies a compact set of architectural ingredients that enable strong out-of-distribution algorithmic generalization: latent recurrent computation (Mechanism 1), algorithmic alignment supervision (Mechanism 2), discrete latent representations (Mechanism 3), and error-correcting dynamics (Mechanism 4). Together, these mechanisms support scalable and robust iterative computation in latent space, enabling the model to scale its

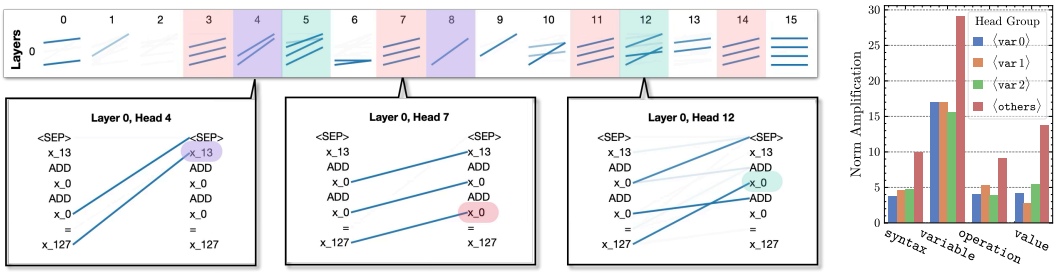


Figure 6. Left. An illustration of the functionality of attention heads in the first layer. Head 4 and 8 attend to the first variable position, Head 5 and 12 attend to the second variable position, Head 3, 7, 11, 14 attend to the third variable position, and the remaining heads attend to the RHS position or do not show a clear attention pattern. **Right.** Norm amplification of each factor's embeddings passed through the combined attention OV matrix by head groups. ⟨`others`⟩ exhibits significantly higher norm amplification, primarily because head 15 performs a self-copy operation at the RHS position.

effective computational depth in proportion to input complexity. Our results show that these ingredients form a minimal recipe for learning depth-invariant algorithms in neural networks. Unlike token-level Chain-of-Thought methods, which rely on explicit reasoning traces, latent recurrence performs reasoning internally and avoids the constraints of autoregressive token generation. The approach complements CoT-style supervision by providing architectural inductive bias rather than depending solely on data-driven pattern learning.

5.2 Limitations and future work

The main limitation of this study is that we validate our proposal on a single task domain. This design choice enabled a level of experimental control and mechanistic analysis difficult to achieve on broader benchmarks: we could precisely manipulate instance complexity, fully isolate the contribution of each mechanism, and perform mechanistic analysis. Such deeply controlled studies are often essential in the early stages of developing new algorithmic reasoning methods. Nevertheless, it remains to evaluate the full architecture—and its ablated variants—on additional tasks. Extending these mechanisms to richer settings such as program execution, dynamic programming lattices, or real-world mathematical reasoning benchmarks is a promising direction for future work.

5.3 Beyond modular arithmetic on computation graphs

We selected modular arithmetic on computation graphs because it instantiates core structural properties of algorithmic computation while allowing fine-grained control over compositional depth. Any program over a finite set of primitive operations can be represented as a directed acyclic computation graph; depth then corresponds to the number of sequential computational steps required for evaluation. Our graphs include branching, reconvergence, and reuse of intermediate values, reflecting the compositional challenges inherent in general algorithmic tasks (e.g., expression evaluation, symbolic manipulation, and dynamic programming). The task is therefore not specific to arithmetic: it represents a canonical instance of a much broader class of DAG-structured reasoning problems. Future work may explore variants involving different primitive operation sets, richer control-flow patterns, or partially observed graph structures, allowing systematic investigation of how each mechanism contributes across a spectrum of algorithmic tasks.

5.4 Generality of the recurrent latent-space architecture with discrete states

The four mechanisms introduced in this work operate at the level of generic computation graphs and do not depend on the specifics of modular arithmetic. Input-adaptive recurrence (Mechanism 1) applies whenever the target algorithm requires variable computational depth, which is a common characteristic of algorithmic computation. Algorithm-alignment supervision (Mechanism 2) assumes only that the underlying computation admits an iterative decomposition, which is true for any DAG-structured algorithm; moreover, as shown in Section D, this structure can be learned implicitly without oracle-provided depths. Discrete latent states (Mechanism 3) fit naturally in domains where intermediate values lie in a finite

symbolic space and provide the stability needed for long-chain computation and fixed-point halting. Finally, error correction (Mechanism 4) is broadly applicable: perturbing intermediate states to encourage self-correcting dynamics is a task-agnostic strategy for preventing error accumulation. Together, these mechanisms define a general architectural template for depth-invariant algorithmic reasoning.

5.5 Integration with language models

A natural question is how latent recurrent reasoning with discrete states might integrate with large-scale language models. Autoregressive Transformers perform reasoning through token generation, whereas the present architecture performs reasoning in latent space through iterative refinement. Combining these paradigms may yield hybrid models in which latent recurrence supports algorithmic stability and input-adaptive computation, while continuous representations and autoregressive decoding preserve the flexibility and generality of modern language models. Potential avenues include embedding a discrete latent recurrent module inside a larger pretrained model, introducing hybrid continuous–discrete latent spaces, or enabling latent computation as an internal reasoning step between forward passes. As large language models are increasingly applied to tasks requiring deeper compositional generalization, the mechanisms studied in this work may provide useful inductive biases for stabilizing long-chain reasoning.

References

[1] Jordan B Pollack. "Recursive distributed representations". In: *Artificial Intelligence* (1990) (cited on pages 1, 15).

[2] Richard Socher, Brody Huval, Christopher D Manning, and Andrew Y Ng. "Semantic compositionality through recursive matrix-vector spaces". In: *Proceedings of the 2012 joint conference on empirical methods in natural language processing and computational natural language learning*. 2012 (cited on pages 1, 15).

[3] Brenden Lake and Marco Baroni. "Generalization without systematicity: On the compositional skills of sequence-to-sequence recurrent networks". In: *International conference on machine learning*. PMLR. 2018 (cited on pages 1, 4).

[4] Petar Veličković and Charles Blundell. "Neural algorithmic reasoning". In: *Patterns* (2021) (cited on page 1).

[5] Noam Chomsky. "Syntactic structures". Mouton de Gruyter, 1957 (cited on page 1).

[6] Jerry A Fodor and Zenon W Pylyshyn. "Connectionism and cognitive architecture: A critical analysis". In: *Cognition* (1988) (cited on pages 1, 15).

[7] Brenden M Lake, Tomer D Ullman, Joshua B Tenenbaum, and Samuel J Gershman. "Building machines that learn and think like people". In: *Behavioral and brain sciences* (2017) (cited on pages 1, 15).

[8] John R Anderson. "Acquisition of cognitive skill." In: *Psychological review* (1982) (cited on page 1).

[9] Mark K Singley and John Robert Anderson. "The transfer of cognitive skill". Harvard University Press, 1989 (cited on page 1).

[10] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. "Chain-of-thought prompting elicits reasoning in large language models". In: *Advances in neural information processing systems* (2022) (cited on pages 1, 3, 16).

[11] Takeshi Kojima, Shixiang Shane Gu, Machel Reid, Yutaka Matsuo, and Yusuke Iwasawa. "Large language models are zero-shot reasoners". In: *Advances in neural information processing systems* (2022) (cited on pages 1, 16).

[12] Hyung Won Chung et al. "Scaling Instruction-Finetuned Language Models". 2022. arXiv: 2210.11416 [cs.LG] (cited on page 1).

[13] Hanmeng Liu, Zhiyang Teng, Leyang Cui, Chaoli Zhang, Qiji Zhou, and Yue Zhang. "LogiCoT: Logical Chain-of-Thought Instruction Tuning". In: *Findings of the Association for Computational Linguistics: EMNLP 2023*. Ed. by Houda Bouamor, Juan Pino, and Kalika Bali. Singapore: Association for Computational Linguistics, Dec. 2023 (cited on pages 1, 16).

[14] Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Mark Chen, Heewoo Jun, Łukasz Kaiser, Matthias Plappert, Jerry Tworek, Jacob Hilton, Reiichiro Nakano, Christopher Hesse, and John Schulman. "Training Verifiers to Solve Math Word Problems". arXiv:2110.14168 [cs]. Nov. 2021 (cited on pages 1, 3).

[15] Aitor Lewkowycz, Anders Johan Andreassen, David Dohan, Ethan Dyer, Henryk Michalewski, Vinay Venkatesh Ramasesh, Ambrose Slone, Cem Anil, Imanol Schlag, Theo Gutman-Solo, Yuhuai Wu, Behnam Neyshabur, Guy Gur-Ari, and Vedant Misra. "Solving Quantitative Reasoning Problems with Language Models". In: *Advances in Neural Information Processing Systems*. Ed. by Alice H. Oh, Alekh Agarwal, Danielle Belgrave, and Kyunghyun Cho. 2022 (cited on pages 1, 3, 16).

[16] Hunter Lightman, Vineet Kosaraju, Yura Burda, Harri Edwards, Bowen Baker, Teddy Lee, Jan Leike, John Schulman, Ilya Sutskever, and Karl Cobbe. "Let's Verify Step by Step". 2023. arXiv: 2305.20050 [cs.LG] (cited on page 1).

[17] Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Xiao Bi, Haowei Zhang, Mingchuan Zhang, Y. K. Li, Y. Wu, and Daya Guo. "DeepSeekMath: Pushing the Limits of Mathematical Reasoning in Open Language Models". 2024. arXiv: 2402.03300 [cs.CL] (cited on page 1).

[18] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. "Attention is all you need". In: *Advances in neural information processing systems* (2017) (cited on pages 1, 15, 17).

[19] Cem Anil, Yuhuai Wu, Anders Andreassen, Aitor Lewkowycz, Vedant Misra, Vinay Ramasesh, Ambrose Slone, Guy Gur-Ari, Ethan Dyer, and Behnam Neyshabur. "Exploring Length Generalization in Large Language Models". In: *Advances in Neural Information Processing Systems* (Dec. 6, 2022) (cited on pages 1, 4, 15).

[20] Amirhossein Kazemnejad, Inkit Padhi, Karthikeyan Natesan Ramamurthy, Payel Das, and Siva Reddy. "The Impact of Positional Encoding on Length Generalization in Transformers". In: *Advances in Neural Information Processing Systems* (Dec. 15, 2023) (cited on pages 1, 4, 15, 17).

[21] Samy Jelassi, Stéphane d'Ascoli, Carles Domingo-Enrich, Yuhuai Wu, Yuanzhi Li, and François Charton. "Length Generalization in Arithmetic Transformers". 2023. arXiv: 2306.15400 [cs.LG] (cited on pages 1, 4, 15).

[22] Hattie Zhou, Arwen Bradley, Etai Littwin, Noam Razin, Omid Saremi, Joshua M. Susskind, Samy Bengio, and Preetum Nakkiran. "What Algorithms can Transformers Learn? A Study in Length Generalization". In: *The Twelfth International Conference on Learning Representations*. 2024 (cited on pages 1, 4, 15).

[23] Jonathan Thomm, Giacomo Camposampiero, Aleksandar Terzic, Michael Hersche, Bernhard Schölkopf, and Abbas Rahimi. "Limits of transformer language models on learning to compose algorithms". In: *Advances in Neural Information Processing Systems* (2024) (cited on page 1).

[24] Nouha Dziri, Ximing Lu, Melanie Sclar, Xiang Lorraine Li, Liwei Jiang, Bill Yuchen Lin, Sean Welleck, Peter West, Chandra Bhagavatula, Ronan Le Bras, et al. "Faith and fate: Limits of transformers on compositionality". In: *Advances in Neural Information Processing Systems* (2023) (cited on page 1).

[25] Kaya Stechly, Karthik Valmeekam, and Subbarao Kambhampati. "Chain of thoughtlessness? an analysis of cot in planning". In: *The Thirty-eighth Annual Conference on Neural Information Processing Systems*. 2024 (cited on pages 1, 4, 15).

[26] Yongchao Zhou, Uri Alon, Xinyun Chen, Xuezhi Wang, Rishabh Agarwal, and Denny Zhou. "Transformers Can Achieve Length Generalization But Not Robustly". Feb. 14, 2024. arXiv: 2402.09371 [cs]. Pre-published (cited on pages 1, 4, 15).

[27] Hyung Won Chung, Le Hou, Shayne Longpre, Barret Zoph, Yi Tay, William Fedus, Yunxuan Li, Xuezhi Wang, Mostafa Dehghani, Siddhartha Brahma, et al. "Scaling instruction-finetuned language models". In: *Journal of Machine Learning Research* (2024) (cited on pages 3, 16).

[28] Tian Ye, Zicheng Xu, Yuanzhi Li, and Zeyuan Allen-Zhu. "Physics of Language Models: Part 2.1, Grade-School Math and the Hidden Reasoning Process". July 29, 2024. arXiv: 2407.20311 [cs]. Pre-published (cited on page 3).

[29] Róbert Csordás, Kazuki Irie, and Juergen Schmidhuber. "The Devil is in the Detail: Simple Tricks Improve Systematic Generalization of Transformers". In: *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*. 2021 (cited on page 4).

[30] Jonathan Baxter. "A model of inductive bias learning". In: *Journal of artificial intelligence research* (2000) (cited on pages 4, 15).

[31] David Barrett, Felix Hill, Adam Santoro, Ari Morcos, and Timothy Lillicrap. "Measuring abstract reasoning in neural networks". In: *International conference on machine learning*. PMLR. 2018 (cited on pages 4, 15).

[32] Anirudh Goyal and Yoshua Bengio. "Inductive biases for deep learning of higher-level cognition". In: *Proceedings of the Royal Society A* (2022) (cited on pages 4, 15).

[33] Mostafa Dehghani, Stephan Gouws, Oriol Vinyals, Jakob Uszkoreit, and Łukasz Kaiser. "Universal Transformers". Mar. 5, 2019. arXiv: 1807.03819 [cs, stat]. Pre-published (cited on pages 4, 5, 15).

[34] Alex Graves. "Adaptive Computation Time for Recurrent Neural Networks". Feb. 21, 2017. arXiv: 1603.08983 [cs]. Pre-published (cited on pages 5, 6, 15).

[35] Andrea Banino, Jan Balaguer, and Charles Blundell. "PonderNet: Learning to Ponder". Sept. 2, 2021. arXiv: 2107.05407. Pre-published (cited on pages 5, 6, 15).

[36] Avi Schwarzschild, Eitan Borgnia, Arjun Gupta, Furong Huang, Uzi Vishkin, Micah Goldblum, and Tom Goldstein. "Can You Learn an Algorithm? Generalizing from Easy to Hard Problems with Recurrent Networks". Nov. 2, 2021. arXiv: 2106.04537 [cs]. Pre-published (cited on pages 5, 15).

[37] Arpit Bansal, Avi Schwarzschild, Eitan Borgnia, Zeyad Emam, Furong Huang, Micah Goldblum, and Tom Goldstein. "End-to-End Algorithm Synthesis with Recurrent Networks: Logical Extrapolation Without Overthinking". Oct. 14, 2022. arXiv: 2202.05826 [cs]. Pre-published (cited on pages 5, 6, 15).

[38] Ying Fan, Yilun Du, Kannan Ramchandran, and Kangwook Lee. "Looped Transformers for Length Generalization". Sept. 25, 2024. arXiv: 2409.15647. Pre-published (cited on pages 5, 15).

[39] Jonas Geiping, Sean McLeish, Neel Jain, John Kirchenbauer, Siddharth Singh, Brian R. Bartoldson, Bhavya Kailkhura, Abhinav Bhatele, and Tom Goldstein. "Scaling up Test-Time Compute with Latent Reasoning: A Recurrent Depth Approach". Feb. 17, 2025. arXiv: 2502.05171 [cs]. Pre-published (cited on pages 5, 15).

[40] Atticus Geiger, Hanson Lu, Thomas F Icard, and Christopher Potts. "Causal Abstractions of Neural Networks". In: *Advances in Neural Information Processing Systems*. Ed. by A. Beygelzimer, Y. Dauphin, P. Liang, and J. Wortman Vaughan. 2021 (cited on pages 8, 16).

[41] Kevin Meng, David Bau, Alex Andonian, and Yonatan Belinkov. "Locating and editing factual associations in gpt". In: *Advances in neural information processing systems* (2022) (cited on pages 8, 16).

[42] Atticus Geiger, Zhengxuan Wu, Christopher Potts, Thomas Icard, and Noah Goodman. "Finding alignments between interpretable causal variables and distributed neural representations". In: *Causal Learning and Reasoning*. PMLR. 2024 (cited on pages 8, 16).

[43] Catherine Olsson et al. "In-context Learning and Induction Heads". In: *Transformer Circuits Thread* (2022) (cited on pages 9, 16).

[44] Neel Nanda, Lawrence Chan, Tom Lieberum, Jess Smith, and Jacob Steinhardt. "Progress measures for grokking via mechanistic interpretability". In: *The Eleventh International Conference on Learning Representations*. 2023 (cited on pages 9, 16).

[45] Yuandong Tian. "Composing Global Optimizers to Reasoning Tasks via Algebraic Objects in Neural Nets". 2024. arXiv: 2410.01779 [cs.LG] (cited on pages 9, 16).

[46] Darshil Doshi, Aritra Das, Tianyu He, and Andrey Gromov. "To grok or not to grok: Disentangling generalization and memorization on corrupted algorithmic datasets". In: *Bulletin of the American Physical Society* (2024) (cited on page 9).

[47] Dieuwke Hupkes, Verna Dankers, Mathijs Mul, and Elia Bruni. "Compositionality decomposed: How do neural networks generalise?" In: *Journal of Artificial Intelligence Research* (2020) (cited on page 15).

[48] Jeffrey L Elman. "Finding structure in time". In: *Cognitive science* (1990) (cited on page 15).

[49] Michael I Jordan. "Serial order: A parallel distributed processing approach". In: *Advances in psychology*. Elsevier, 1997 (cited on page 15).

[50] Sepp Hochreiter and Jürgen Schmidhuber. "Long short-term memory". In: *Neural computation* (1997) (cited on page 15).

[51] Kyunghyun Cho, Bart van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. "Learning Phrase Representations using RNN Encoder–Decoder for Statistical Machine Translation". In: *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. Ed. by Alessandro Moschitti, Bo Pang, and Walter Daelemans. Doha, Qatar: Association for Computational Linguistics, Oct. 2014 (cited on page 15).

[52] Ilya Sutskever, Oriol Vinyals, and Quoc V Le. "Sequence to sequence learning with neural networks". In: *Advances in neural information processing systems* (2014) (cited on page 15).

[53] Liu Yang, Kangwook Lee, Robert Nowak, and Dimitris Papailiopoulos. "Looped Transformers Are Better at Learning Learning Algorithms". Mar. 16, 2024. arXiv: 2311.12424 [cs]. Pre-published (cited on page 15).

[54] Allen Newel and Herbert A Simon. "Computer science as empirical inquiry: Symbols and search". In: *Communications of the ACM* (1976) (cited on page 15).

[55] Artur SD'Avila Garcez, Luis C Lamb, and Dov M Gabbay. "Neural-symbolic cognitive reasoning". Springer Science & Business Media, 2008 (cited on page 15).

[56] Ruslan Salakhutdinov and Geoffrey Hinton. "Deep boltzmann machines". In: *Artificial intelligence and statistics*. PMLR. 2009 (cited on page 15).

[57] Aaron Courville, James Bergstra, and Yoshua Bengio. "A spike and slab restricted Boltzmann machine". In: *Proceedings of the fourteenth international conference on artificial intelligence and statistics*. JMLR Workshop and Conference Proceedings. 2011 (cited on page 15).

[58] Eirikur Agustsson, Fabian Mentzer, Michael Tschannen, Lukas Cavigelli, Radu Timofte, Luca Benini, and Luc V Gool. "Soft-to-hard vector quantization for end-to-end learning compressible representations". In: *Advances in neural information processing systems* (2017) (cited on page 15).

[59] Aaron van den Oord, Oriol Vinyals, and Koray Kavukcuoglu. "Neural Discrete Representation Learning". May 30, 2018. arXiv: 1711.00937 [cs]. Pre-published (cited on page 15).

[60] Gail Weiss, Yoav Goldberg, and Eran Yahav. "Thinking Like Transformers". July 19, 2021. arXiv: 2106.06981 [cs]. Pre-published (cited on page 15).

[61] Paul Smolensky, Roland Fernandez, Zhenghao Herbert Zhou, Mattia Opper, and Jianfeng Gao. "Mechanisms of Symbol Processing for In-Context Learning in Transformer Networks". 2024. arXiv: 2410.17498 [cs.AI] (cited on page 15).

[62] Maxwell Nye, Anders Johan Andreassen, Guy Gur-Ari, Henryk Michalewski, Jacob Austin, David Bieber, David Dohan, Aitor Lewkowycz, Maarten Bosma, David Luan, Charles Sutton, and Augustus Odena. "Show Your Work: Scratchpads for Intermediate Computation with Language Models". 2021. arXiv: 2112.00114 [cs.LG] (cited on page 16).

[63] Nelson Elhage et al. "A Mathematical Framework for Transformer Circuits". In: *Transformer Circuits Thread* (2021) (cited on page 16).

[64] Nelson Elhage et al. "Toy Models of Superposition". In: *Transformer Circuits Thread* (2022) (cited on page 16).

[65] Trenton Bricken et al. "Towards Monosemanticity: Decomposing Language Models With Dictionary Learning". In: *Transformer Circuits Thread* (2023) (cited on page 16).

[66] Emmanuel Ameisen et al. "Circuit Tracing: Revealing Computational Graphs in Language Models". In: *Transformer Circuits Thread* (2025) (cited on page 16).

[67] Jianlin Su, Yu Lu, Shengfeng Pan, Ahmed Murtadha, Bo Wen, and Yunfeng Liu. "RoFormer: Enhanced Transformer with Rotary Position Embedding". 2023. arXiv: 2104.09864 [cs.CL] (cited on page 17).

[68] Pengcheng He, Xiaodong Liu, Jianfeng Gao, and Weizhu Chen. "DeBERTa: Decoding-enhanced BERT with Disentangled Attention". 2021. arXiv: 2006.03654 [cs.CL] (cited on page 17).

A Related Work

Our work is related to several strands of fundamental machine learning research, including issues of out-of-distribution generalization, architectural mechanisms such as recurrence and discretization, chain-of-thought and intermediate supervision methods, and work on mechanistic interpretability techniques.

Out-of-Distribution Generalization. Out-of-distribution (OOD) generalization, along with related capabilities such as compositionality and systematicity, poses a fundamental challenge in machine learning research [1, 2, 30, 31, 47]. These capabilities are crucial for developing AI systems that can reliably apply learned knowledge to novel scenarios, a hallmark of robust intelligence [6, 7, 32]. A particularly important type of OOD generalization, especially for algorithmic reasoning tasks, is *length generalization*—the ability to generalize from simpler or shorter training instances to significantly longer and more structurally complex instances. This has proven to be a key limitation of Transformer-based [18] language models [19–22]. While chain-of-thought techniques alleviate this to some degree by enabling the learning of more complex algorithmic procedures, the ability to generalize far outside the training distribution remains a significant obstacle [25, 26].

Recurrence. Recurrence forms a foundational architectural principle in neural networks, particularly for tasks that involve sequential data or inherently iterative processes [48–50]. These architectures are designed to emulate step-by-step computations by maintaining and updating an internal state, making them well-aligned with problems that have a recursive or layered solution structure. Sequence-to-sequence recurrent architectures for sequence transduction and neural machine translation advanced the state of the art [51, 52], and were instrumental to the development of attention mechanisms and the Transformer architecture [18]. While standard Transformers do not possess a recurrent structure, recurrent variants of the Transformer architecture were explored soon after its introduction [33]. Whereas standard recurrent neural networks apply their recurrence across time or sequence length, recurrent Transformer architectures are *parallel in time* due to the parallel attention mechanism, but recurrent across computational depth—that is, the same Transformer layer is applied iteratively to the sequence as a whole. The recurrent inductive biases have been demonstrated to confer certain advantages in generalization [38, 53]. In our work, recurrence is a key architectural mechanism encoding important inductive biases that aid the discovery of scalable recursive algorithms for solving the underlying mathematical problem.

Adaptive Computation. A critical challenge is handling inputs with varying complexity, where a fixed amount of computation may be inefficient or insufficient. This motivates the concept of adaptive computation, wherein a model can dynamically adjust its computation time, for example by varying the number of recurrent iterations, based on the demands of the input. An important work in this domain is the *Adaptive Computation Time (ACT)* mechanism proposed by Graves [34] for recurrent neural networks, which explicitly models and learns how many computational steps are needed as a function of the input. A version of the ACT mechanism is incorporated in the recurrent Transformer architecture proposed by Dehghani et al. [33]. However, a drawback of such mechanisms is their complexity and difficulty of training. Although efforts have been made to explore simpler adaptive computation methods [35], an even simpler approach is explored by Schwarzschild et al. [36] and Bansal et al. [37], where the halting time is not explicitly modeled by the network, and instead the number of recurrent iterations is scaled at inference time based on the size of the input. This simpler approach can be easier to train, and has been shown to improve out-of-distribution generalization. More recently, Geiping et al. [39] explored the viability of this approach as a way to perform test-time scaling in large language models. In our work, we similarly scale computation time by proportionately scaling the number of recurrent iterations in order to solve more complex problem instances, generalizing far beyond the training distribution.

Discreteness in Neural Networks. Symbolic AI systems derive their power from manipulating discrete symbols according to well-defined rules, which enables robust, precise, and interpretable reasoning [6, 54]. Given this rich tradition of using discrete symbolic states in artificial intelligence, many works have subsequently explored incorporating such discrete latent representations into neural networks [55–59]. Additionally, discreteness is often a central characteristic of *constructions* of Transformer networks for specific tasks. For example, Weiss, Goldberg, and Yahav [60] develops a programming language that represents Transformer-based computation with discrete internal mechanisms. Additionally, Smolen-

sky et al. [61] constructs a Transformer network for a compositional in-context learning task, which features discreteness in both its latent states and attention mechanism. In our work, we explore the use of discrete latent states as a means of *anchoring* the latent representation to a common, depth-invariant space to enable scaling computation far beyond the training distribution while avoiding representational shift across computational depth.

Chain-of-Thought & Algorithmic Supervision. Chain-of-thought techniques have been central to enhancing the reasoning capabilities of large language models. Early usage of the term "chain-of-thought" referred to prompting techniques that condition a model to generate a sequence of intermediate steps before arriving at the final answer [10, 11, 62]. For example, Wei et al. [10] demonstrated that prompting the LLM with a few CoT exemplars caused the model to generate an analogous step-by-step solution, which significantly improved performance on a range of arithmetic, commonsense, and symbolic reasoning tasks. Kojima et al. [11] showed that LLMs can be "zero-shot" reasoners in the sense that simply asking the model to reason step-by-step, without providing in-context learning CoT exemplars, can be sufficient to elicit chain-of-thought-style reasoning and improve performance. Modern usage of the term "chain-of-thought" has extended beyond prompting methods, as it now forms a key component of the *training* pipeline of LLMs, wherein a model is explicitly trained on demonstrations of step-by-step solutions to problems of interest, such as mathematical reasoning [13, 15, 27]. In some situations, chain-of-thought training can be interpreted as providing explicit supervision to align the model to a particular algorithm or procedure for solving a problem, as opposed to simply providing supervision via input-output examples. In our work, we explore traditional chain-of-thought training techniques as baselines, as well as incorporate algorithmic supervision to the internal states of our proposed method.

Mechanistic Interpretability. In our work, we carry out a mechanistic interpretability analysis to probe *how* the model has learned to solve the task and *why* it can do so robustly, generalizing far outside the training distribution. In recent years, there has been a resurgence in work on interpretability, with new techniques being introduced that aim to understand modern large language models [41, 43, 63–66]. Elhage et al. [63] is an influential work in this area of research, introducing a conceptual framework and new terminology that continues to be used in subsequent work. A key early achievement in this line of work is the discovery of "induction head" circuits in large language models [43], which perform a two-step copying operation that is crucial for in-context learning. In our work, we identify a similar mechanism in our recurrent models that is used to copy previously computed variable values. This involves first retrieving the parent variables' names in the first layer, then using these variable names to retrieve their values in the second layer, which are computed elsewhere in the sequence of latent states. Such work is often described as *circuit analysis*, where the goal is to identify sub-networks that are responsible for particular functions. A key method for validating hypotheses about the functions of different model components is *causal interventions* like activation patching or ablations [40–42], which involves systematically modifying parts of the model or input to observe effects on behavior or internal states. We use related causal intervention techniques in our own mechanistic interpretability analysis in this work. Finally, the work by Nanda et al. [44] and Tian [45] is relevant as it specifically investigated how Transformers perform arithmetic, reverse-engineering a modular addition algorithm learned by the feedforward network in a Transformer layer—a phenomenon we also observe in our models.

B Experimental Details on Chain-of-Thought & End-to-End Baselines

This section provides further experimental details on the chain-of-thought and end-to-end baselines.

B.1 End-to-End Baselines

The end-to-end models in our experiments are causal encoder-only Transformer models with a fixed depth and/or number of iterations that are trained with end-to-end supervision only. That is, they receive supervision on the final solution, but do not receive fine-grained supervision on the intermediate steps to explicitly align the models to a universal algorithmic problem-solving procedure.

Within the end-to-end baselines, we consider feedforward models and recurrent models. Feedforward models have a fixed number of layers and independently-learned parameters at each layer. Recurrent models, on the other hand, have a recurrent block consisting of some number of Transformer layers, which is applied recurrently for a fixed number of iterations.

Recognizing the importance of positional encoding for length generalization [20], we explore several positional encoding methods for each class of methods that we evaluate. In particular, we evaluate learned absolute positional embeddings [18] (AbPE), Rotary Positional Encoding [67] (RoPE), No Positional Encoding [20] (NoPE), and the relative positional-encoding method proposed by [68] (DeBERTa).

We perform a hyperparameter search across each of these factors, varying the number of recurrent iterations T, the number of layers per recurrent block L, the hidden state dimension D, and the positional encoding method. As described in the main text, we train on a dataset of examples with up to 32 nodes, and evaluate on examples varying in size from 8 nodes to 128 nodes. Figure 7 depicts the average OOD performance as measured by the "% Fully Solved" metric for each baseline model configuration. The results in the main text correspond to the best-performing end-to-end models according to this metric. In particular, the best-performing recurrent model is RoPE-T4L2H16D256, and the best-performing feedforward model is DeBERTa-T1L8H16D256. Note that the naming scheme describes the positional encoding method, the number of recurrent steps T, the number of layers L in the Transformer block, the number of attention heads H, and the model dimension D. $T = 1$ corresponds to a "feedforward" model with no recurrence.

Figure 8 depicts additional experimental results for the end-to-end baseline experiments.

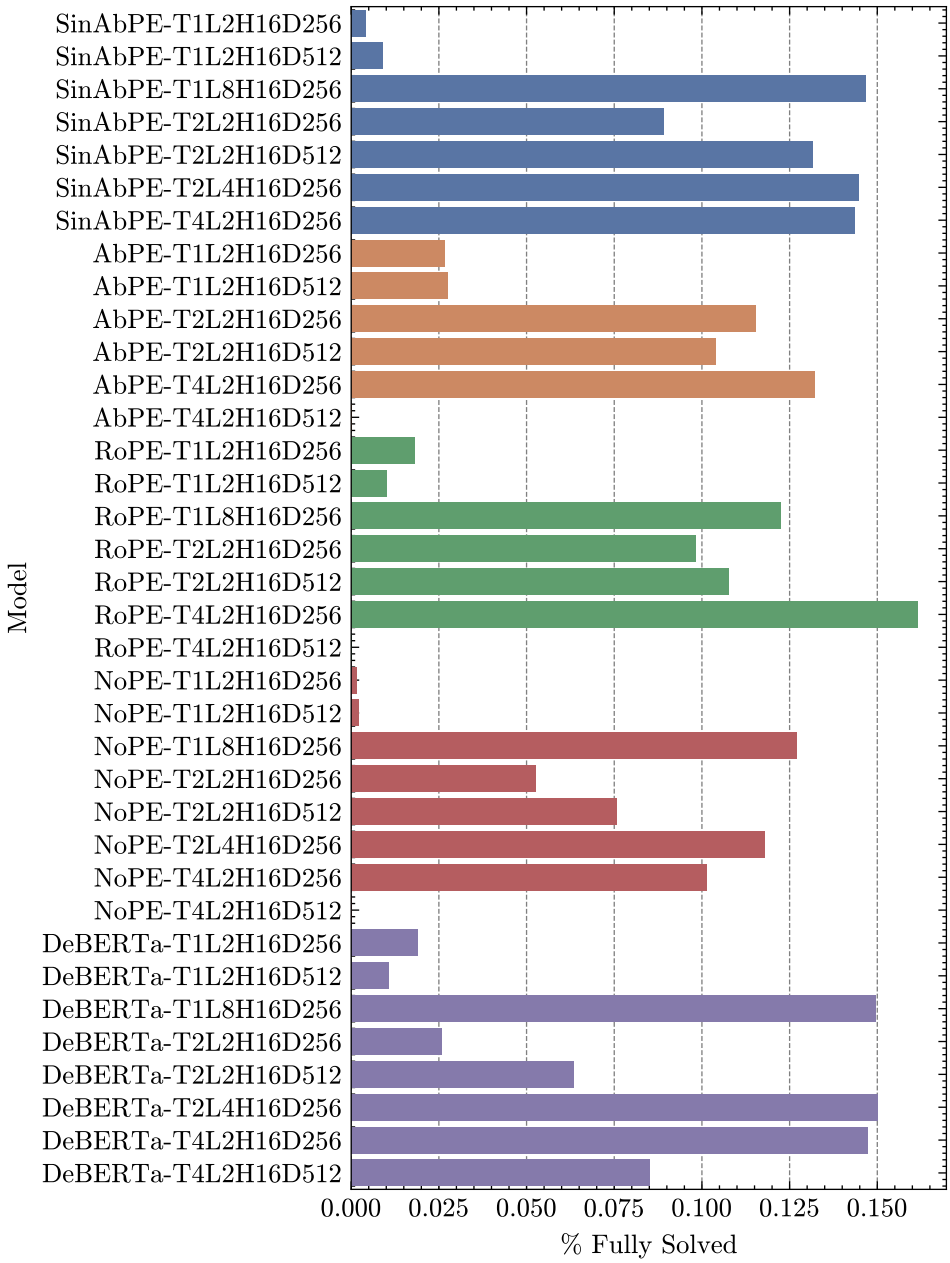
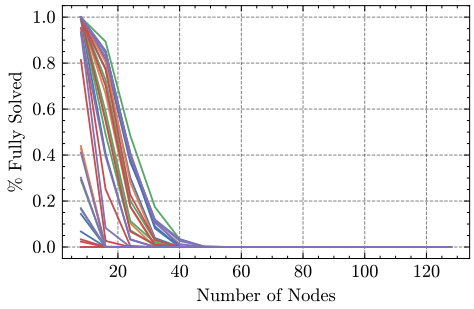
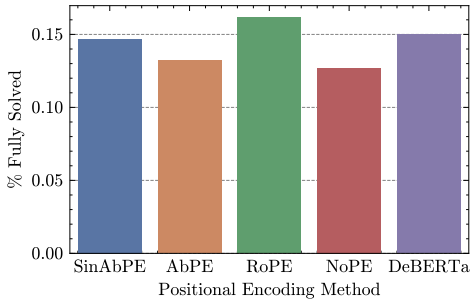


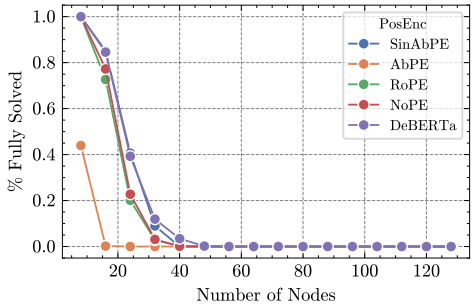
Figure 7. A comparison of average OOD generalization performance of different feedforward and recurrent baselines, varying architectural hyperparameters. This is computed as the average of the "% Fully Solved" metric computed on inputs of varying size from $N = 8$ to $N = 128$.



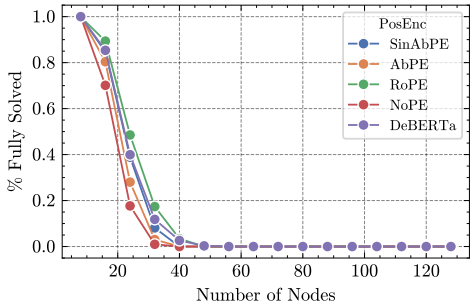
(a) Each line corresponds to an experimental run. Lines are color-coded by positional encoding, but other architectural hyperparameters vary and are not represented.



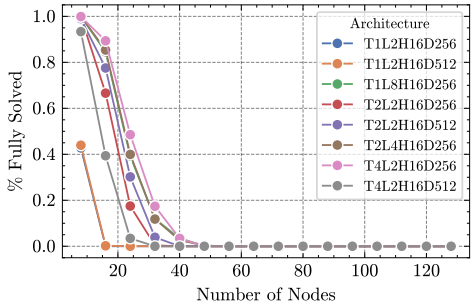
(b) Average "% Fully Solved" across test splits for the best model of each positional encoding method. The relative positional encoding methods, RoPE and DeBERTa perform best.



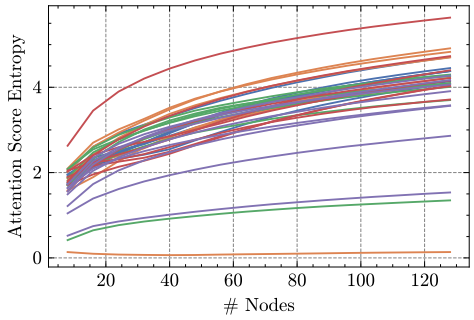
(c) % Fully solved by graph size for best model of each positional encoding method in the *feedforward* baselines.



(d) % Fully solved by graph size for best model of each positional encoding method in the *recurrent* baselines.



(e) % Fully solved by graph size for the best model of each architectural configuration. Recurrent models slightly outperform feedforward models. Computational depth (i.e., $T \cdot L$) is crucial, with shallow models performing poorly even on the smallest in-distribution inputs.



(f) Average attention score entropy by input size. Attention scores disperse as the input size increases.

Figure 8. Further experimental results for end-to-end baselines. All end-to-end models struggle to generalize beyond the training distribution, regardless of architectural hyperparameters.

B.2 Chain-of-Thought Baselines

The chain-of-thought baselines in our experiments are causal Transformer language models that are trained with a next-token prediction objective on sequence data that includes a step-by-step solution of the problem instance. The models are evaluated by prompting them with the problem instance and autoregressively generating the entire chain-of-thought via a greedy decoding procedure.

We begin by providing more details on the construction of the chain-of-thought trajectories for these baselines, then provide further details on the experimental setup and present additional results.

B.2.1 Chain-of-Thought Trajectories

We experiment with a few different types of chain-of-thought trajectories, providing different levels and styles of supervision on the intermediate computation.

As described in the main text, the first part of the sequence is always the description of the input problem, which matches the format of the other methods we consider: a sequence of equations that define a computational graph to be solved. This is then followed by a special $\langle \texttt{CoT} \rangle$ token which indicates the end of the input and the beginning of the chain-of-thought. The chain-of-thought involves solving each variable in the input in linear order, one-by-one.

We experiment with two types of CoT trajectories that vary the level of detail. The first provides supervision on the values only. The CoT simply recalls that variables one-by-one and computes their values, without recalling the equation that defined them.

$$[\ldots \texttt{Input Prompt} \ldots] \langle \texttt{CoT} \rangle [\ldots] \langle x_{101} \rangle \langle = \rangle \langle 4 \rangle$$

The second type of CoT trajectory involves first recalling the equation that defined the variable, then recalling the values of the variables in the equation, and then computing the value of the desired variable. This requires a longer chain-of-thought but provides richer supervision.

$$[\ldots \texttt{Input Prompt} \ldots] \langle \texttt{CoT} \rangle [\ldots] \langle x_{101} \rangle \langle = \rangle \langle x_{23} \rangle \langle + \rangle \langle x_{91} \rangle \langle = \rangle \langle 22 \rangle \langle + \rangle \langle 5 \rangle \langle = \rangle \langle 4 \rangle$$

Below, we provide an example of a *full* CoT trajectory on an input with $N = 32$ nodes.

$\langle 2\rangle\langle=\rangle\langle x_{3}\rangle$ [sep] $\langle 2\rangle\langle=\rangle\langle x_{30}\rangle$ [sep] $\langle 18\rangle\langle=\rangle\langle x_{12}\rangle$ [sep] $\langle 14\rangle\langle=\rangle\langle x_{11}\rangle$ [sep]
$\langle 15\rangle\langle=\rangle\langle x_{20}\rangle$ [sep] $\langle 8\rangle\langle=\rangle\langle x_{23}\rangle$ [sep] $\langle x_{30}\rangle\langle=\rangle\langle x_{9}\rangle$ [sep] $\langle x_{23}\rangle\langle+\rangle\langle x_{3}\rangle\langle=\rangle\langle x_{22}\rangle$ [sep]
$\langle x_{20}\rangle\langle\times\rangle\langle x_{23}\rangle\langle=\rangle\langle x_{27}\rangle$ [sep] $\langle x_{3}\rangle\langle+\rangle\langle x_{22}\rangle\langle=\rangle\langle x_{0}\rangle$ [sep] $\langle x_{3}\rangle\langle+\rangle\langle x_{22}\rangle\langle\times\rangle\langle x_{11}\rangle\langle=\rangle\langle x_{26}\rangle$ [sep]
$\langle x_{20}\rangle\langle-\rangle\langle x_{22}\rangle\langle+\rangle\langle x_{23}\rangle\langle=\rangle\langle x_{13}\rangle$ [sep] $\langle x_{22}\rangle\langle=\rangle\langle x_{24}\rangle$ [sep] $\langle x_{12}\rangle\langle\times\rangle\langle x_{23}\rangle\langle-\rangle\langle x_{0}\rangle\langle=\rangle\langle x_{17}\rangle$ [sep]
$\langle x_{11}\rangle\langle\times\rangle\langle x_{26}\rangle\langle=\rangle\langle x_{28}\rangle$ [sep] $\langle x_{13}\rangle\langle-\rangle\langle x_{11}\rangle\langle+\rangle\langle x_{23}\rangle\langle=\rangle\langle x_{21}\rangle$ [sep] $\langle x_{17}\rangle\langle-\rangle\langle x_{3}\rangle\langle=\rangle\langle x_{25}\rangle$ [sep]
$\langle x_{30}\rangle\langle\times\rangle\langle x_{17}\rangle\langle-\rangle\langle x_{23}\rangle\langle=\rangle\langle x_{6}\rangle$ [sep] $\langle x_{17}\rangle\langle=\rangle\langle x_{16}\rangle$ [sep] $\langle x_{11}\rangle\langle+\rangle\langle x_{21}\rangle\langle=\rangle\langle x_{7}\rangle$ [sep]
$\langle x_{28}\rangle\langle+\rangle\langle x_{17}\rangle\langle-\rangle\langle x_{21}\rangle\langle=\rangle\langle x_{14}\rangle$ [sep] $\langle x_{7}\rangle\langle=\rangle\langle x_{15}\rangle$ [sep] $\langle x_{7}\rangle\langle=\rangle\langle x_{31}\rangle$ [sep]
$\langle x_{12}\rangle\langle+\rangle\langle x_{3}\rangle\langle+\rangle\langle x_{14}\rangle\langle=\rangle\langle x_{5}\rangle$ [sep] $\langle x_{14}\rangle\langle=\rangle\langle x_{19}\rangle$ [sep] $\langle x_{23}\rangle\langle-\rangle\langle x_{5}\rangle\langle\times\rangle\langle x_{7}\rangle\langle=\rangle\langle x_{29}\rangle$ [sep]
$\langle x_{5}\rangle\langle=\rangle\langle x_{18}\rangle$ [sep] $\langle x_{25}\rangle\langle+\rangle\langle x_{23}\rangle\langle-\rangle\langle x_{19}\rangle\langle=\rangle\langle x_{4}\rangle$ [sep] $\langle x_{14}\rangle\langle\times\rangle\langle x_{29}\rangle\langle-\rangle\langle x_{5}\rangle\langle=\rangle\langle x_{2}\rangle$ [sep]
$\langle x_{29}\rangle\langle\times\rangle\langle x_{28}\rangle\langle-\rangle\langle x_{7}\rangle\langle=\rangle\langle x_{1}\rangle$ [sep] $\langle x_{3}\rangle\langle\times\rangle\langle x_{23}\rangle\langle\times\rangle\langle x_{18}\rangle\langle=\rangle\langle x_{8}\rangle$ [sep]
$\langle x_{8}\rangle\langle-\rangle\langle x_{28}\rangle\langle-\rangle\langle x_{0}\rangle\langle=\rangle\langle x_{10}\rangle$ $\langle$CoT$\rangle$ $\langle x_{3}\rangle\langle=\rangle\langle 2\rangle$ [sep] $\langle x_{30}\rangle\langle=\rangle\langle 2\rangle$ [sep]
$\langle x_{12}\rangle\langle=\rangle\langle 18\rangle$ [sep] $\langle x_{11}\rangle\langle=\rangle\langle 14\rangle$ [sep] $\langle x_{20}\rangle\langle=\rangle\langle 15\rangle$ [sep] $\langle x_{23}\rangle\langle=\rangle\langle 8\rangle$ [sep]
$\langle x_{9}\rangle\langle=\rangle\langle x_{30}\rangle\langle=\rangle\langle 2\rangle$ [sep] $\langle x_{22}\rangle\langle=\rangle\langle x_{23}\rangle\langle+\rangle\langle x_{3}\rangle\langle=\rangle\langle 10\rangle$ [sep] $\langle x_{27}\rangle\langle=\rangle\langle x_{20}\rangle\langle\times\rangle\langle x_{23}\rangle\langle=\rangle\langle 5\rangle$ [sep]
$\langle x_{0}\rangle\langle=\rangle\langle x_{3}\rangle\langle+\rangle\langle x_{22}\rangle\langle=\rangle\langle 12\rangle$ [sep] $\langle x_{26}\rangle\langle=\rangle\langle x_{3}\rangle\langle+\rangle\langle x_{22}\rangle\langle\times\rangle\langle x_{11}\rangle\langle=\rangle\langle 7\rangle$ [sep]
$\langle x_{13}\rangle\langle=\rangle\langle x_{20}\rangle\langle-\rangle\langle x_{22}\rangle\langle+\rangle\langle x_{23}\rangle\langle=\rangle\langle 13\rangle$ [sep] $\langle x_{24}\rangle\langle=\rangle\langle x_{22}\rangle\langle=\rangle\langle 10\rangle$ [sep]
$\langle x_{17}\rangle\langle=\rangle\langle x_{12}\rangle\langle\times\rangle\langle x_{23}\rangle\langle-\rangle\langle x_{0}\rangle\langle=\rangle\langle 17\rangle$ [sep] $\langle x_{28}\rangle\langle=\rangle\langle x_{11}\rangle\langle\times\rangle\langle x_{26}\rangle\langle=\rangle\langle 6\rangle$ [sep]
$\langle x_{21}\rangle\langle=\rangle\langle x_{13}\rangle\langle-\rangle\langle x_{11}\rangle\langle+\rangle\langle x_{23}\rangle\langle=\rangle\langle 7\rangle$ [sep] $\langle x_{25}\rangle\langle=\rangle\langle x_{17}\rangle\langle-\rangle\langle x_{3}\rangle\langle=\rangle\langle 15\rangle$ [sep]
$\langle x_{6}\rangle\langle=\rangle\langle x_{30}\rangle\langle\times\rangle\langle x_{17}\rangle\langle-\rangle\langle x_{23}\rangle\langle=\rangle\langle 3\rangle$ [sep] $\langle x_{16}\rangle\langle=\rangle\langle x_{17}\rangle\langle=\rangle\langle 17\rangle$ [sep]
$\langle x_{7}\rangle\langle=\rangle\langle x_{11}\rangle\langle+\rangle\langle x_{21}\rangle\langle=\rangle\langle 21\rangle$ [sep] $\langle x_{14}\rangle\langle=\rangle\langle x_{28}\rangle\langle+\rangle\langle x_{17}\rangle\langle-\rangle\langle x_{21}\rangle\langle=\rangle\langle 16\rangle$ [sep]
$\langle x_{15}\rangle\langle=\rangle\langle x_{7}\rangle\langle=\rangle\langle 21\rangle$ [sep] $\langle x_{31}\rangle\langle=\rangle\langle x_{7}\rangle\langle=\rangle\langle 21\rangle$ [sep] $\langle x_{5}\rangle\langle=\rangle\langle x_{12}\rangle\langle+\rangle\langle x_{3}\rangle\langle+\rangle\langle x_{14}\rangle\langle=\rangle\langle 13\rangle$ [sep]
$\langle x_{19}\rangle\langle=\rangle\langle x_{14}\rangle\langle=\rangle\langle 16\rangle$ [sep] $\langle x_{29}\rangle\langle=\rangle\langle x_{23}\rangle\langle-\rangle\langle x_{5}\rangle\langle\times\rangle\langle x_{7}\rangle\langle=\rangle\langle 10\rangle$ [sep]
$\langle x_{18}\rangle\langle=\rangle\langle x_{5}\rangle\langle=\rangle\langle 13\rangle$ [sep] $\langle x_{4}\rangle\langle=\rangle\langle x_{25}\rangle\langle+\rangle\langle x_{23}\rangle\langle-\rangle\langle x_{19}\rangle\langle=\rangle\langle 7\rangle$ [sep]
$\langle x_{2}\rangle\langle=\rangle\langle x_{14}\rangle\langle\times\rangle\langle x_{29}\rangle\langle-\rangle\langle x_{5}\rangle\langle=\rangle\langle 9\rangle$ [sep] $\langle x_{1}\rangle\langle=\rangle\langle x_{29}\rangle\langle\times\rangle\langle x_{28}\rangle\langle-\rangle\langle x_{7}\rangle\langle=\rangle\langle 16\rangle$ [sep]
$\langle x_{8}\rangle\langle=\rangle\langle x_{3}\rangle\langle\times\rangle\langle x_{23}\rangle\langle\times\rangle\langle x_{18}\rangle\langle=\rangle\langle 1\rangle$ [sep] $\langle x_{10}\rangle\langle=\rangle\langle x_{8}\rangle\langle-\rangle\langle x_{28}\rangle\langle-\rangle\langle x_{0}\rangle\langle=\rangle\langle 6\rangle$

B.2.2 Experimental Details & Additional Results

We perform a hyperparameter search varying: the number of recurrent iterations T, the number of layers per recurrent block L, the hidden state dimension D, and the positional encoding method. As described in the main text, we train on a dataset of examples with up to 32 nodes, and evaluate on examples varying in size from 8 nodes to 128 nodes. Figure 9 depicts the average OOD performance as measured by the "% Fully Solved" metric for each baseline model configuration. The results in the main text correspond to the best-performing CoT-supervised model according to this metric, which is the RoPE-T4L2H16D256 model.

Figure 11 depicts additional experimental results for the end-to-end baseline experiments. We highlight a few observations here:

- Figure 10 shows that some models are able to recall the equation structure correctly in their CoT, but are unable to robustly compute the values correctly. This suggests that a common source of error in the CoT baselines is the arithmetic computation, rather than copying equations from the input.
- As with the end-to-end baselines, the positional encoding method was critical for performance and length generalization. Among the methods we evaluated, we found NoPE to perform best, generalizing well to 40 nodes when trained on $N \leq 32$ nodes. The other positional encoding methods fail to generalize beyond the training regime. No method generalized robustly beyond 40 nodes.
- As with the end-to-end baselines, the computational depth of the model had a significant effect on performance. In particular, 4 layer models failed to learn the task well, but 8-layer models achieved good in-distribution performance and a limited degree of out-of-distribution generalization.

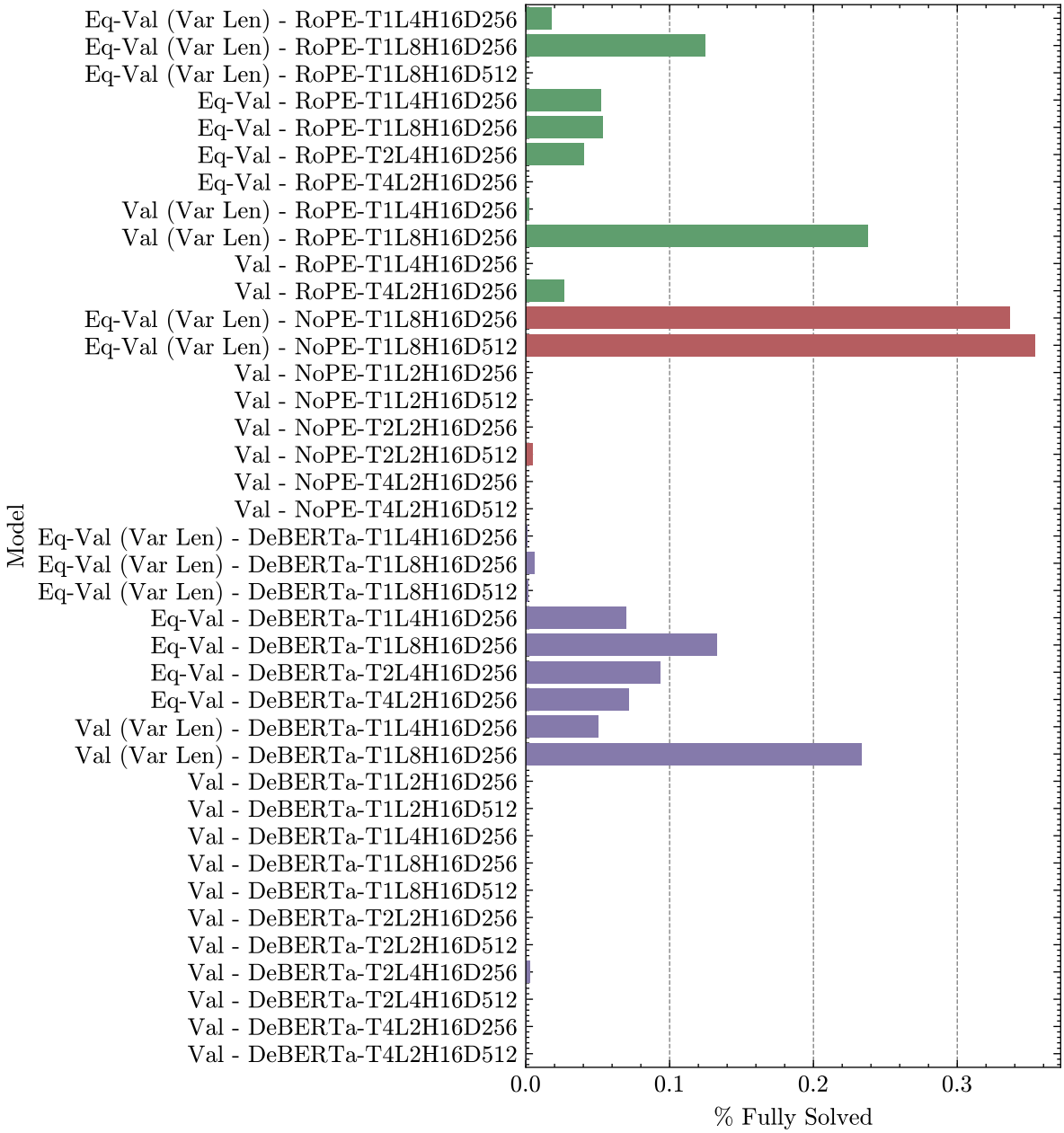


Figure 9. A comparison of average OOD generalization performance of different CoT-supervised baselines, varying architectural hyperparameters. The metric is full sequence accuracy, which measures the proportion of inputs where every node's value is computed correctly. The naming scheme matches the previous section, but adds a prefix describing the format of the CoT trajectories. "Val" means that the CoT trajectory directly computes the values of each variable, whereas "Eq-Val" first recalls the equations and then computes the values. Here, "(Var Len)" indicates runs where the input problem size is variable and randomly sampled in $N \leq 32$, rather than being only $N = 32$.

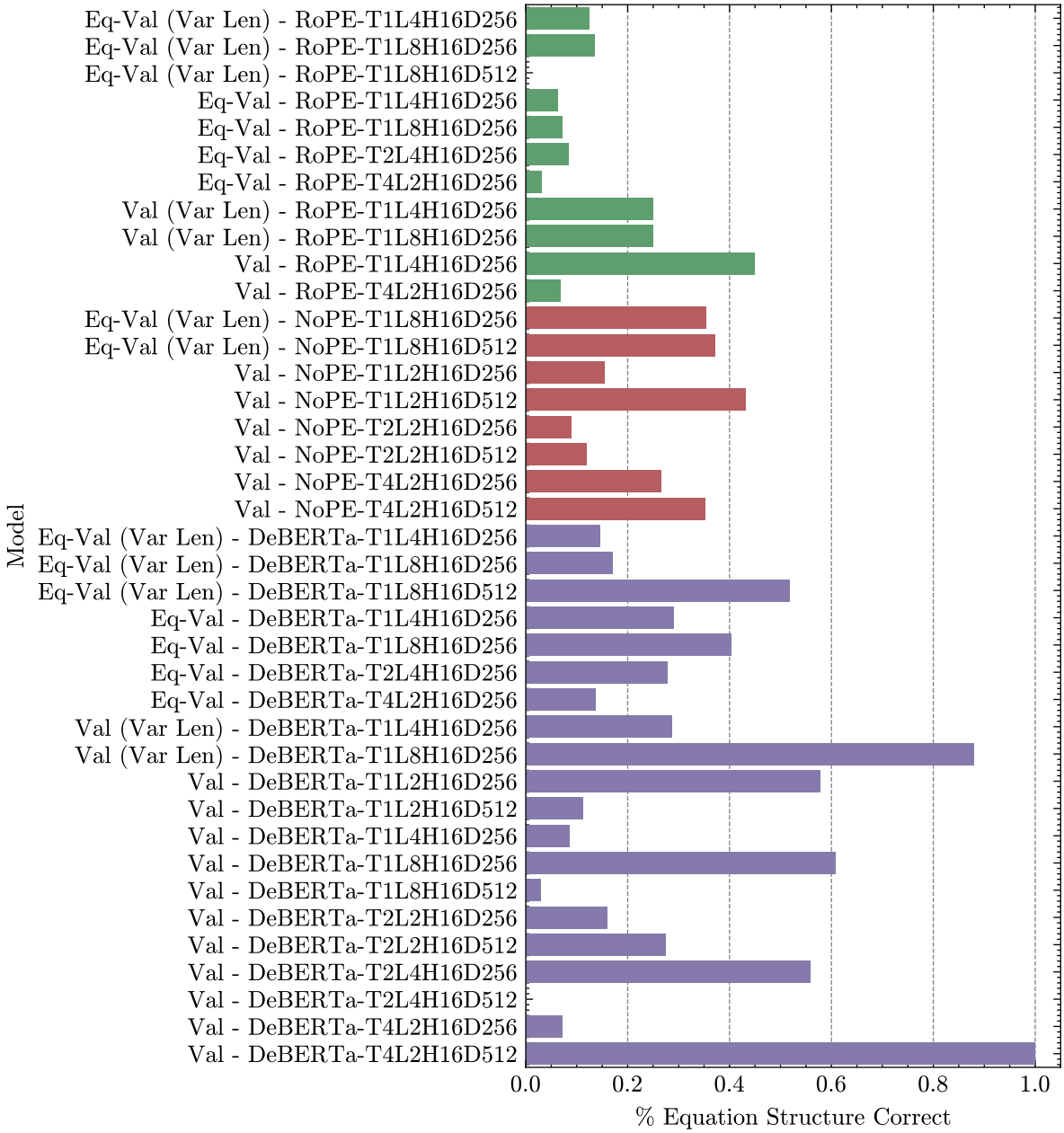
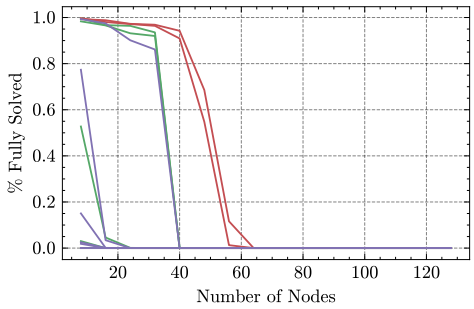
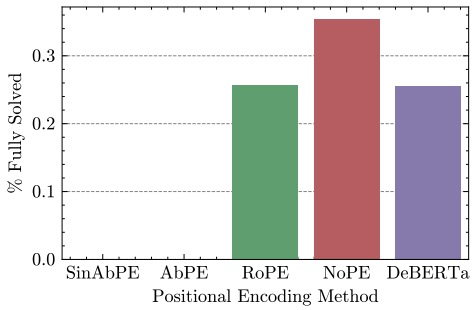


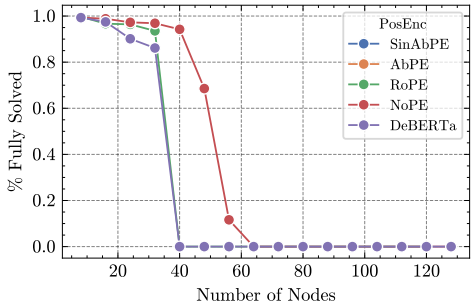
Figure 10. A comparison of average OOD generalization performance of different CoT-supervised baselines, varying architectural hyperparameters. The metric is "% Equation Structure Correct", which measures the proportion of inputs where the autoregressively generated CoT has the correct equation structure (without checking whether the values computed are correct).



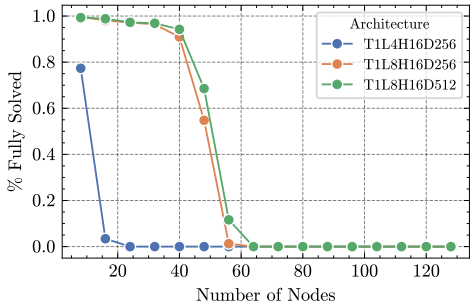
(a) Each line corresponds to an experimental run. Lines are color-coded by positional encoding, but other architectural hyperparameters vary and are not represented.



(b) Average OOD performance across test splits for the best model of each positional encoding method.



(c) % Fully solved by graph size for the best model of each positional encoding method. We find NoPE to achieve the best out-of-distribution generalization performance, generalizing well to 40 nodes when trained on $N \leq 32$ nodes. The other positional encoding methods fail to generalize beyond the training regime.



(d) % Fully solved by graph size for the best model of each architectural configuration. Computational depth (i.e., $T \cdot L$) is crucial for good performance, with shallow models performing poorly even in-distribution on larger inputs.

Figure 11. Further experimental results for CoT baselines. While chain-of-thought supervision yields improved performance over end-to-end models, out-of-distribution generalization capabilities are limited.

C Details on Latent State Supervision

C.1 Latent State Embedding Structure

The input to the model is presented as a sequence of equations defining the value of each node in the computation graph. The vocabulary of the input includes variable names (e.g., x_{42}), numerical values (e.g., 17), operations (e.g., $+$), and special symbols like equality $\langle = \rangle$ or equation separation $[\texttt{sep}]$.

To provide the model with supervision on each part of the input, we employ a special tokenization and embedding scheme. We use a factored structure to tokenize each symbol in the input into 4-component tokens: syntax, variable, operation, and value. For example, the input $\langle 17 \rangle \langle = \rangle \langle x_{42} \rangle [\texttt{sep}] \ldots$, is tokenized as follows before the first iteration:

		syntax	variable	operation	value
$\langle 17 \rangle$	$\rightarrow$	value	N/A	N/A	17
$\langle = \rangle$	$\rightarrow$	$\langle = \rangle$	N/A	N/A	N/A
$\langle x_{42} \rangle$	$\rightarrow$	variable	x_{42}	N/A	empty
$[\texttt{sep}]$	$\rightarrow$	$[\texttt{sep}]$	N/A	N/A	N/A

The *syntax* factor can be `value`, `variable`, `operation`, or the special symbols $\langle = \rangle$ or $[\texttt{sep}]$. The *variable* factor is the variable names $\{x_0, \ldots, x_{127}\}$. The *operation* factor is the set of arithmetic operations (e.g, $+, -, \times$). The *value* factor is the set of numerical values (i.e., $\{0, \ldots, 22\}$). We also include an `N/A` symbol for the *variable*, *operation*, and *value* factors. For example, symbols with value syntax do not have a variable factor, etc. We also include a special `empty` symbol for the value factor of variable tokens. In the input to the model, the variable tokens have empty value factors because their values have not been computed yet. As the model processes the input, it iteratively computes the values of different variables and fills in their value factor.

We train a separate embedder for each factor, and map the input to vector embeddings by embedding each factor and adding the embeddings.

C.2 Latent State Supervision

The *Continuous Latent Space Supervision*, *Discrete Latent Space Supervision*, and *Discrete Latent Space Supervision* ↺ methods share the same latent state supervision scheme. We train these recurrent models to learn to solve the input problem be computing node values one layer deeper in the computation graph with each recurrent iteration. We do this by defining a loss function at each iteration that penalizes predictions only for variables with depth less than or equal to the current iteration.

For each $\texttt{factor} \in \{\texttt{syntax}, \texttt{variable}, \texttt{operation}, \texttt{value}\}$, we learn a linear read-out layer $W_{\texttt{factor}} \in \mathbb{R}^{d_{\text{model}} \times |\mathcal{V}_{\texttt{factor}}|}$ that maps the vector state at the end of the recurrent iteration to a prediction of each factor. Here, $\mathcal{V}_{\texttt{factor}}$ denotes the vocabulary for the given factor (e.g., for the *value* factor, this is $\{0, \ldots, 22, \texttt{N/A}, \texttt{empty}\}$).

We provide the model with supervision on its latent states by defining a loss for each factor and at each recurrent iteration. In particular, the loss function for the value factor is defined such that the model is trained to predict the values of all variables that occur at depth $\leq t$ in the computation graph. In particular, for an input sequence $X = (x_1, \ldots, x_n)$, the value factor loss at iteration t is defined as

$$\text{Loss}(\texttt{factor} = \texttt{value}, \text{iteration} = t) = \sum_{\substack{i \in [n] \\ \text{Depth}(x_i) \leq t}} \ell\left(W_{\texttt{value}} E_i^{(t)}, \text{Value}(x_i)\right). \tag{5}$$

where $\text{Depth}(x_i)$ is the depth of the variable x_i in the input computation graph, $\text{Value}(x_i)$ is its computed value, and $E_i^{(t)} \in \mathbb{R}^{d_{\text{model}}}$ is the vector embedding of x_i at recurrent iteration t. Here, ℓ is the cross-entropy loss.

The overall loss used to train the models is the sum of the individual factor losses at each iteration.

$$\text{Loss} = \sum_{\texttt{factor}} \sum_{t} \text{Loss}(\texttt{factor} = \texttt{value}, \text{iteration} = t). \tag{6}$$

C.3 Discretization of Intermediate States

The training procedure described above applies to the *Continuous Latent Space Supervision*, *Discrete Latent Space Supervision*, and *Discrete Latent Space Supervision* ↺ methods in the same way. In the methods with a discrete latent bottleneck, we introduce an additional architectural mechanism where the read-out layers are used not only for computing the loss on the intermediate iterations, but also for mapping the latent representation to a discrete space.

In particular, letting $E_i^{(t)}$ be the embedding of the i-th token after t recurrent iterations, we use argmax decoding of the linear read-outs to map the embedding to a discrete prediction for each factor. This discrete state is then re-embedded using the same learned embedding module to form the vectorized input $E_i^{(t+1)}$ at the next iteration. In particular, at iteration t, the model's forward pass is defined as follows

$$
\begin{aligned}
(\tilde{E}_1^{(t+1)}, \dots, \tilde{E}_n^{(t+1)}) &\leftarrow \text{RecurrentTransformerBlock}(E_1^{(t)}, \dots, E_n^{(t)}) \\
z_{i,\texttt{factor}}^{(t+1)} &\leftarrow \arg\max\{W_{\texttt{factor}}\, \tilde{E}_i^{(t+1)}\} \quad \texttt{factor} \in \{\texttt{syntax}, \texttt{variable}, \texttt{operation}, \texttt{value}\} \\
E_{i,\texttt{factor}}^{(t+1)} &\leftarrow \text{FactorEmbed}(z_{i,\texttt{factor}}^{(t+1)}) \quad \texttt{factor} \in \{\texttt{syntax}, \texttt{variable}, \texttt{operation}, \texttt{value}\} \\
E_i^{(t+1)} &\leftarrow E_{i,\texttt{syntax}}^{(t+1)} + E_{i,\texttt{variable}}^{(t+1)} + E_{i,\texttt{operation}}^{(t+1)} + E_{i,\texttt{value}}^{(t+1)}.
\end{aligned}
\tag{7}
$$

This discretization enables us to train the model with a type of *teacher-forcing* across recurrent iterations. That is, we can teacher-force the inputs $z_i^{(t)}$ at each iteration t. This enables more efficient training.

C.4 Self-Correction Mechanism

In a reasoning task, each reasoning step depends crucially on the prior steps in the reasoning path. If a mistake is made at any stage, all subsequent computation is affected, and the error is often fatal. As the size of the problem and the number of computational steps scale, the likelihood of an error occurring at *some point* in the reasoning process becomes large, limiting the ability to generalize indefinitely to more complex problems. To address this challenge, a reasoning model must be able to detect and correct errors as they occur in order to recover when a mistake is made in its previous computation.

We train the model to detect and correct errors by randomly corrupting the model's latent state. That is, at each iteration, with some small probability, we corrupt a random selection of the value components of the models' discrete states. To achieve good loss, the model must learn to detect when a previously-computed value is incorrect and correct it before proceeding.

C.5 Experiment Details & Additional Results

As with the baselines, we explore the effect of different architectural hyperparameters, such as positional encoding and the depth of the recurrent block, on model performance. Figure 12 depicts the average OOD perfromance as measured by the "% Fully Solved" metric for each model configuration in the *Discrete Latent Space Supervision*, and *Discrete Latent Space Supervision* ↺ methods. The results in the main text correspond to the best-performing models according to this metric. In particular, the best-performing *Discrete Latent Space Supervision* model is DeBERTa-L2H16D256, and the best-performing *Discrete Latent Space Supervision* ↺ model is DeBERTa-L4H16D384.

Figure 13 depicts additional experimental results for the *Discrete Latent Space Supervision*, and *Discrete Latent Space Supervision* ↺ methods. We highlight a few observations here:

- The positional encoding method is critical for length generalization. The DeBERTa positional encoding method (a relative positional encoding method) performed the best by far.
- 2 layers for the recurrent block were sufficient for the *Discrete Latent Space Supervision* method. However, the recorrection mechanism of *Discrete Latent Space Supervision* ↺ required a deeper recurrent block. We saw no significant improvement for the re-correction mechanism with 2 layers, but with 4 layers, the re-correction mechanism kicked in and enabled near-perfect OOD generalization.

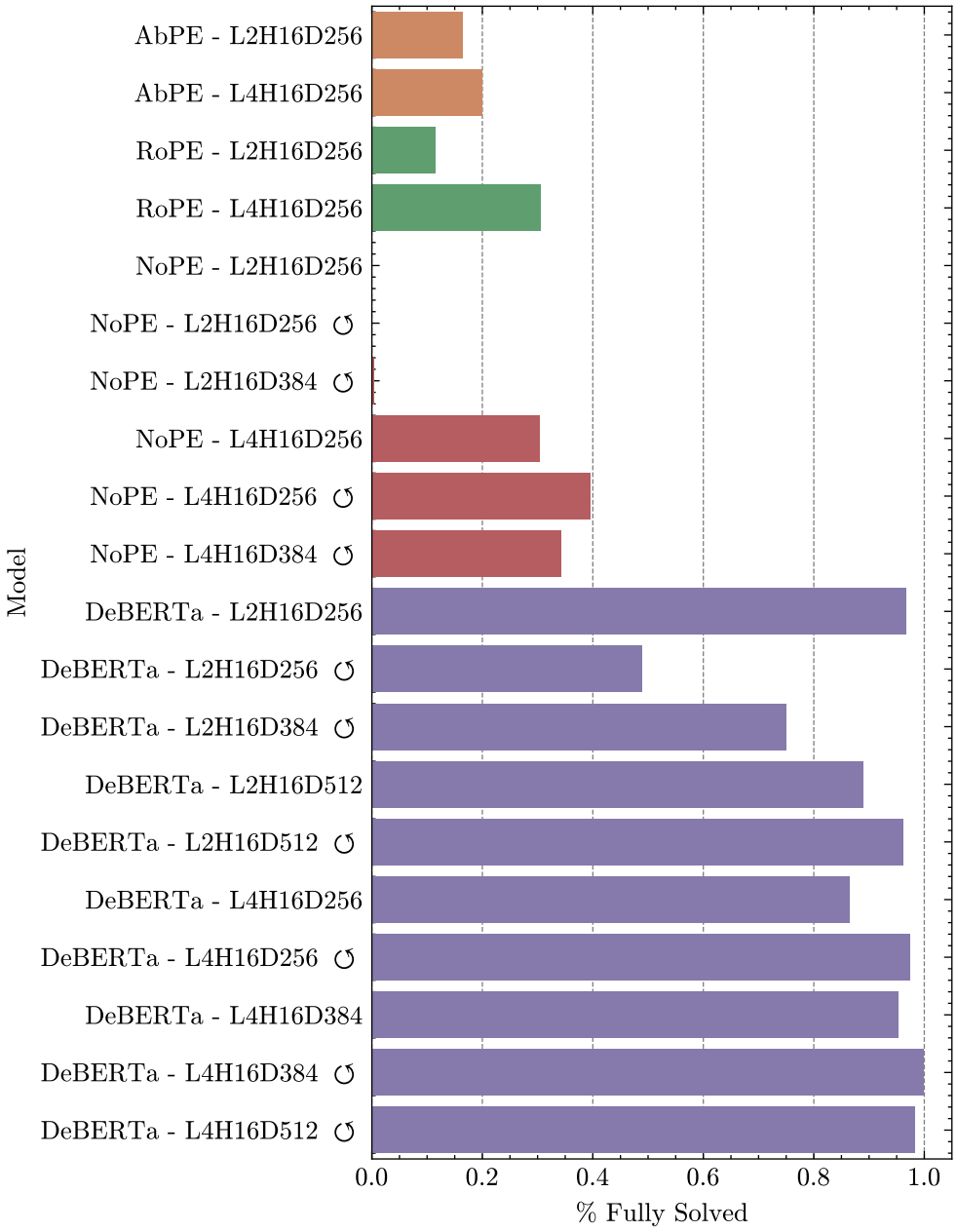
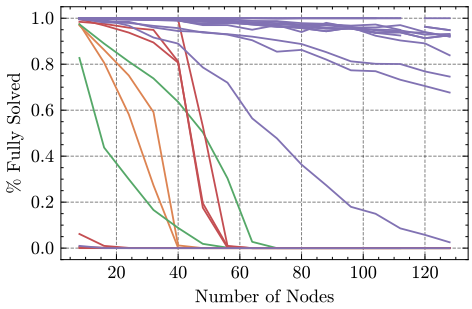
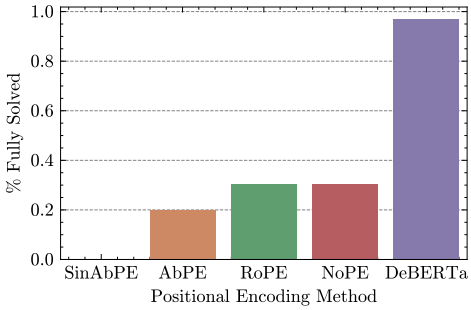


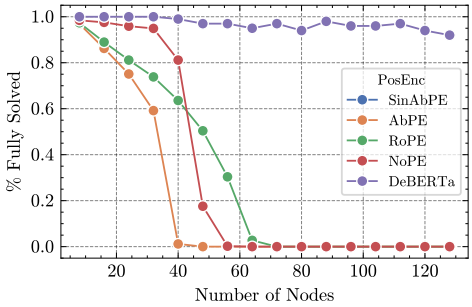
Figure 12. Average "% Fully Solved", across # nodes between 8 and 128, with training on ≤ 32 nodes,



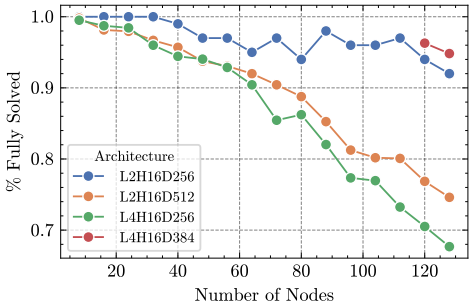
(a) Each line corresponds to an experimental run. Lines are color-coded by positional encoding, but other architectural hyperparameters vary and are not represented.



(b) Average OOD performance across test splits for the best model of each positional encoding method.



(c) % Fully solved by graph size for best model of each positional encoding method.



(d) % Fully solved by graph size for best model of each architectural configuration.

Figure 13. Further experimental results for methods exploring our proposed architectural mechanisms.

D Learning to infer an effective depth structure without any oracle decomposition

In the main experiments, the algorithm-alignment loss (**??**) leverages oracle-provided depth information to expose a coarse layer-by-layer decomposition of the computation graph. This auxiliary supervision biases the model toward learning an iterative algorithm rather than a direct input–output mapping. While such supervision is widely available for synthetic algorithmic tasks—and is strictly weaker than the step-level traces required by token-space CoT training—it is natural to ask whether the model can *infer* the effective depth structure without any oracle decomposition.

This section presents a simple modification that removes the need for depth-based masking while preserving the benefits of iterative latent computation.

D.1 Method: Iterative Refinement with ⟨EMPTY⟩-Discounted Loss

The key idea is to encourage the model to identify, during each recurrent iteration, which portions of the computation can already be solved and which should remain unsolved until later iterations. To make this possible without supervision on intermediate states, we augment the latent and prediction spaces with a special ⟨EMPTY⟩ token representing a "not yet solved" value, as in the model in the main paper.

At each recurrent iteration, the model predicts a distribution

$$p = \text{softmax}(\text{logits})$$

over all possible entries, including ⟨EMPTY⟩. Recall that in the main model, we apply a different loss for each iteration to align the learned algorithm with the input graph's depth structure. Here, we instead simply apply the same final-label loss at every iteration, but with a modified loss function.

For target label y and discount factor $\alpha \in (0, 1)$, we use the discounted loss:

$$\mathcal{L}(p, y) = CE(p, y)\,(1 - \alpha \cdot p[\langle\texttt{EMPTY}\rangle]), \tag{8}$$

where $CE(p, y)$ is the standard cross-entropy. The loss behaves as follows:

- If the model assigns probability mass to an *incorrect non-⟨EMPTY⟩* value, the penalty remains the full cross-entropy loss.
- If the model assigns probability mass to ⟨EMPTY⟩, the loss is discounted by a factor proportional to α.

This creates a "safe" prediction state for positions the model is not yet ready to solve: predicting ⟨EMPTY⟩ is preferable to guessing an incorrect value, but still penalized so that the model eventually resolves all nodes. Over iterations, the recurrent update learns to fill in increasingly many positions as information accumulates in the latent state. Crucially, no oracle decomposition or depth mask is required.

As in the main experiments, we train with a fixed number of recurrent steps during training and allow the model to run until convergence at evaluation time.

D.2 Results

Here, we present preliminary results with this method. The model achieves strong in-distribution performance and exhibits robust out-of-distribution generalization, approaching the performance of the full model that uses oracle depth supervision.

Qualitatively, the learned latent dynamics mirror those observed under oracle decomposition: early iterations populate easily solvable nodes, while later iterations resolve deeper nodes and correct earlier errors. Quantitatively, the models show high accuracy at depths far beyond the training regime, confirming that the latent recurrent architecture is capable of discovering the effective computational schedule without explicit supervision.

These findings indicate that explicit depth supervision is not required for the architecture to learn a depth-invariant algorithm. The ⟨EMPTY⟩-discounted loss provides a simple and effective way for the model to learn an iterative refinement strategy, broadening the applicability of the approach to domains where intermediate states are unavailable or costly to obtain.

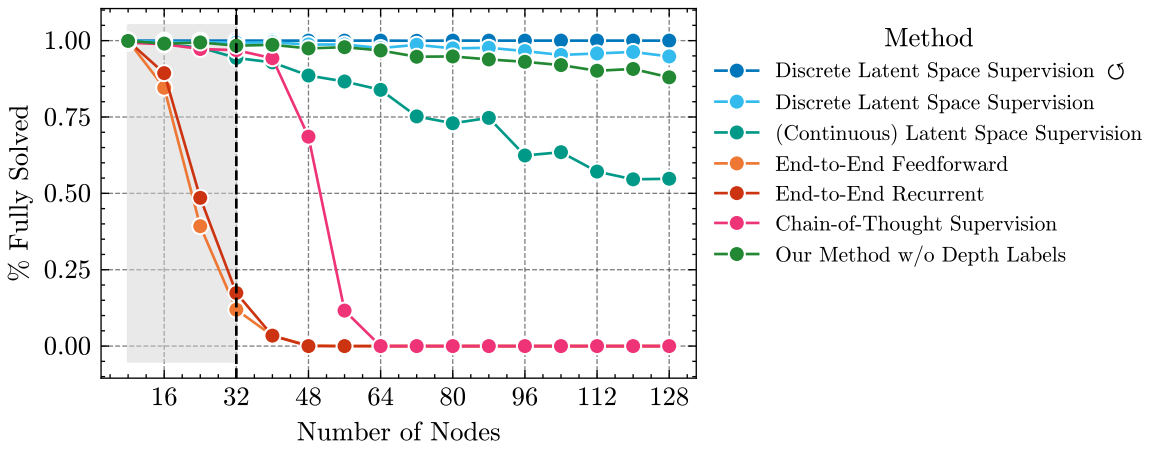


Figure 14. Our method achieves strong out-of-distribution generalization performance even without access to a depth decomposition oracle during training. Performance far exceeds the end-to-end and CoT baselines, and approaches the full method described in the main paper.

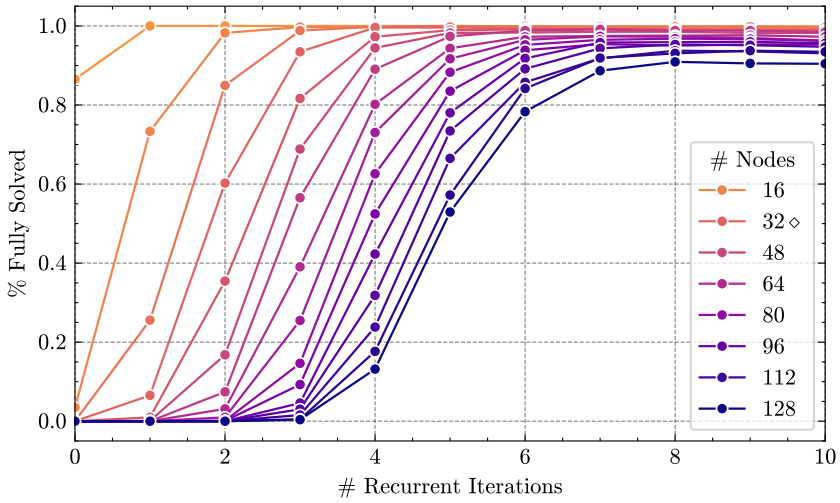


Figure 15. Proportion of input instances that are fully solved by number of iterations for graphs of varying sizes.

E Details of Mechanistic Interpretability Analysis

In this section, we provide additional experimental evidence to support our claim on the mechanism learned by the model together with the error analysis of the model's predictions.

Notice: The following analysis is conducted only for showing the computation happening at the Right-Hand Side (RHS) position in each equation, as it is the place where the model is expected to compute the final result.

Model Configuration. We use DeBERTa-L2H16D256 trained with our proposed *Discrete Latent Space Supervision* method (without the re-correction mechanism) on the mathematical reasoning task. Specifically, the recurrent Transformer model is configured with two layers, 16 attention heads, a hidden dimension of 256. We use DeBERTa's relative positional encoding method. The training data is the same as the one used in the main text. We choose this model setup because it is the best-performing configuration according to the "% Fully Solved" metric displayed in Figure 12 for a two-layer model. In particular, we cherry-pick the best-performing model trained with the same configuration with different random seeds, which has a "% Fully Solved" score of 99.98% on the OOD test set. We use this model to conduct the mechanism analysis for better interpretability. We train on modular-23 addition task with maximum graph size 32. The total number of variables in the training data is 128. The testing data used for mechanism analysis has the maximum graph size 128.

Additional Definitions and Notations. In the following, we frequently use the following definitions and notations:

- **Head output**: For a given attention head h, we define the head output for a query vector $q_h \in \mathbb{R}^{d_h}$ for head dimension d_h as

$$\text{Head Output}(h) = \text{softmax}(q_h K_h^\top / \sqrt{d_k}) V_h W_O^{(h)},$$

  where K_h and V_h are the key and value matrices of the head h, respectively, and $W_O^{(h)}$ is the output projection matrix of the head h. In standard attention mechanism, each head'squery, key and value vector is obtained by applying a linear transformation to the attention input specified by $W_Q^{(h)}$, $W_K^{(h)}$, and $W_V^{(h)}$, respectively. The above definition can be applied to define the head output for *any* query position. However, since our mechanism analysis focuses exclusively on the RHS position, we consistently define the head output as the output of the attention head at the RHS position. Here, we don't include the bias of the head output projection in the definition of the head output, as the bias applied to the final attention output is not specified to individual heads.

- **OV combined matrix**: For a group of attention heads $\mathcal{H} \subseteq [16]$, we define the OV combined matrix as

$$W_{OV}^{(\mathcal{H})} = \sum_{h \in \mathcal{H}} W_V^{(h)} W_O^{(h)},$$

  where $W_O^{(h)} \in \mathbb{R}^{d_h \times d}$ and $W_V^{(h)} \in \mathbb{R}^{d \times d_h}$ are the output projection matrix and the value projection matrix of the attention head $h \in \mathcal{H}$, respectively.

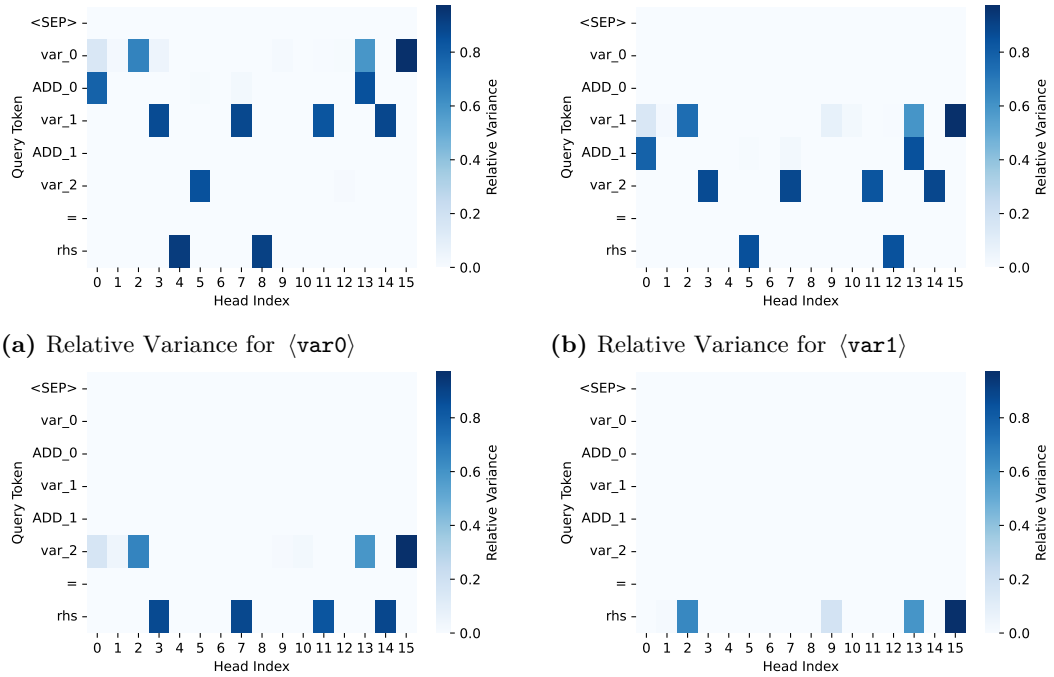
E.1 First Layer Attention: Variable Copying

In the followiing, we will give a detailed analysis of the first layer attention mechanism.

E.1.1 Group Structure in the First Layer Attention

Group Structure in the First Layer Attention. The attention heads in the first layer exhibit a clear grouping pattern based on which variable position they attend to. To rigorously demonstrate this grouping structure, we conduct controlled experiments by systematically varying the input data. On a testing example with number of nodes 128, we append a new **probe equation** to the end of the sequence with the following format:

$$[\texttt{sep}]\ \langle \texttt{var0} \rangle\ \langle + \rangle\ \langle \texttt{var1} \rangle\ \langle + \rangle\ \langle \texttt{var2} \rangle\ \langle = \rangle\ \langle \texttt{rhs} \rangle\,. \tag{9}$$



(a) Relative Variance for ⟨var0⟩

(b) Relative Variance for ⟨var1⟩

(c) Relative Variance for ⟨var2⟩

(d) Relative Variance for ⟨rhs⟩

Figure 16. Relative variance heatmaps when we vary the value of ⟨var0⟩, ⟨var1⟩, ⟨var2⟩, and ⟨rhs⟩. Each row corresponds to a query position and each column corresponds to an attention head.

To identify the group structure in the first layer attention and detect which heads belong to which groups, we measure each head's **relative variance** when we vary the value of each of the four variables ⟨var0⟩, ⟨var1⟩, ⟨var2⟩, and ⟨rhs⟩. See below for more details.

Experiment Design for Group Structure Detection. To detect which heads attend to ⟨var0⟩, we fix ⟨var1⟩, ⟨var2⟩, and ⟨rhs⟩ while randomly sampling different variables $\langle x_i \rangle$ with $i = 1, \ldots, 128$ for ⟨var0⟩. Note that the variable $\langle x_i \rangle$ must be computed in the preceding equations. Otherwise, the model cannot compute the value of $\langle x_i \rangle$ and the probe equation is invalid. As our testing data has all variables computed in the preceding equations, we collect 128 samples that only differ in the value of ⟨var0⟩. We then compute the relative variance of each attention head's output at each position within the probe equation across the 128 samples. Note that relative variance is a measure of how much the head's output varies in response to changes in ⟨var0⟩, and we give the rigorous definition in the next paragraph. The analysis can also be conducted for the other variable positions, and the results are reported in Figure 16.

Relative Variance Calculation. Let us take n different sequences, e.g., the 128 sequences in the above experiment design. We only consider one RHS position for the probe equation in each sequence. For a given attention head h, we define the relative variance over the n sequences as

$$\text{Relative Variance}(h) = \frac{\text{tr}(\text{Cov}(\text{Head Output}(h)))}{\mathbb{E}[\|\text{Head Output}(h)\|_2^2]}. \tag{10}$$

Here, the covariance matrix for a sequence of vectors $v_1, \ldots, v_n$ is defined as

$$\text{Cov}(v_1, \ldots, v_n) = \frac{1}{n} \sum_{i=1}^{n} (v_i - \bar{v})(v_i - \bar{v})^\top,$$

where $\bar{v}$ is the mean of the sequence over the n sequences, and $\mathbb{E}[\cdot]$ is the empirical expectation over the n sequences. Intuitively, the relative variance measures how much the head's output varies relative to its overall magnitude. *A higher relative variance indicates that the attention head's output has a larger variance relative to its overall magnitude.* Since we only change $\langle$`var0`$\rangle$ in the above example, a larger relative variance for a head means that the head's output is primarily influenced by $\langle$`var0`$\rangle$.

Illustration of Figure 16. In Figure 16, we plot the relative variance heatmaps for all 16 attention heads when we vary the variable names of $\langle$`var0`$\rangle$, $\langle$`var1`$\rangle$, $\langle$`var2`$\rangle$, and $\langle$`rhs`$\rangle$. Each column corresponds to a different attention head, and each row corresponds to a different query position. As our goal is to understand the mechanism at the $\langle$`rhs`$\rangle$ position, we focus on the last row corresponding to the $\langle$`rhs`$\rangle$ query position in the figures. Each subfigure plots the relative variance heatmap for altering one particular variable. A higher relative variance in one subfigure indicates that the attention head's output is more sensitive to changes in the corresponding variable. Based on these results, we observe a clear group structure in the first layer's attention heads: Heads 4 and 8's relative variance is high only when we change the value of $\langle$`var0`$\rangle$, while the relative variance of the other heads is low. This facts suggests that heads 4 and 8 attend primarily to $\langle$`var0`$\rangle$. Similary, heads 5 and 12 attend primarily to $\langle$`var1`$\rangle$, and heads 3, 7, 11, and 14 attend primarily to $\langle$`var2`$\rangle$. The last subfigure plots the relative variance heatmap for the $\langle$`rhs`$\rangle$ position. We observe that heads 2, 9, 13, and 15 attend to the RHS position, and the remaining heads do not exhibit a distinct pattern according to the relative variance heatmap. Notice that the above head groups are all disjoint. This further indicates that each head is specialized for a specific variable position. This result is also backed up by the trace of the attention logits of the first layer attention heads as shown in Figure 6.

Summary of the Group Structure. We observe that the attention heads in the first layer exhibit a clear grouping pattern based on which variable position they attend to. Therefore, we know that the first layer attention must be copying something from the LHS variables to the RHS position. In the following, we will conduct further analysis to identify what information is being copied.

E.1.2 First Layer Attention Copying the Variable Identity

Here, by saying "copying the variable identity", we mean that the attention head is copying the factored embedding of `variable` among the four factored embeddings $\{$`syntax`, `variable`, `operation`, `value`$\}$. In the previous experiment, we have identified that the first layer attention heads are grouped into four groups, each of which attends to a specific variable position. Now, we aim to identify which of the four factored embeddings is being copied by these groups.

Norm Amplification Analysis. To achieve our goal, we analyze the norm amplification for each type of factored embeddings when passed through the combined OV matrix of different head groups. We define the norm amplification for a matrix W_{OV} on input x as:

$$\text{Norm Amplification}(W_{OV}, x) = \frac{\|W_{OV}x\|_2}{\|x\|_2}. \tag{11}$$

Note that the above definition can be applied to any matrix W_{OV} and input x with the conformal dimensions. For our analysis, we will consider W_{OV} as the combined OV matrix of the attention heads in a group. Specifically, let $\mathcal{H} \subseteq [16]$ be a group of attention heads, and let $W_O^{(h)}$ and $W_V^{(h)}$ be the output projection matrix and the value projection matrix of the attention head $h \in \mathcal{H}$, respectively. The combined attention OV matrix for a group $\mathcal{H} \subseteq [16]$ is then defined as

$$W_{OV}^{(\mathcal{H})} = \sum_{h \in \mathcal{H}} W_V^{(h)} W_O^{(h)}.$$

If the OV matrix is responsible for copying the identity of the variable, we expect to see a large amplification for the "`variable`" factored embedding, and a small amplification for the other types of embeddings $\{$`syntax`, `operation`, `value`$\}$. With a slight abuse of notation, for each `factored embedding type` $\in \{$`syntax`, `variable`, `operation`, `value`$\}$, we can define

the norm amplification as

$$\text{Norm Amplification}(W_{OV}^{(\mathcal{H})}, \texttt{factored embedding type}) = \mathbb{E}_{x \in \texttt{factored embedding type}}\Big[\frac{\|W_{OV}^{(\mathcal{H})}x\|_2}{\|x\|_2}\Big].$$

Here, $\mathbb{E}_{x \in \texttt{factored embedding type}}$ is the average over the set of all factored embeddings of the same type. For example, if we consider the "`variable`" factored embedding type, we have

$$\text{Norm Amplification}(W_{OV}^{(\mathcal{H})}, \texttt{variable}) = \mathbb{E}_{x \in \texttt{variable}}\Big[\frac{\|W_{OV}^{(\mathcal{H})}x\|_2}{\|x\|_2}\Big],$$

where x iterates over all the 128 factored embeddings of the type `variable`. The results in Figure 6 (right) are computed by averaging the norm amplification over all the factored embeddings within each "`factored embedding type`". In Figure 17, we further histogram each factored embedding's norm amplification for different "`factored embedding type`" while different groups are highlighted in different colors, which provides a more detailed view of the norm amplification across different groups.

Comparing the Norm Amplification Across Different Groups. It can be observed from Figure 17 that the amplification factor for the "`variable`" factored embedding is significantly larger than that of the other types of embeddings, confirming our hypothesis that the OV matrix is responsible for copying the `variable` factored embeddings of the variable to the RHS position. We also observe that the amplification factor for the "`syntax`" factored embedding is also relatively large, which is consistent with the fact that the model is copying the variable identity.

From Figure 17, we confirm that for the first layer, attention has larger norm amplification for the "`variable`" factored embedding (≈ 15) than the other types of embeddings (≈ 5). In particular, the larger norm from the "other" group as shown in Figure 6 is due to the self-copying operation of the attention head 15 at the RHS position.

Additional Evidence on change of number of variables. We provide one interesting observation on how the model handles different numbers of variables in the input equations in Figure 18. Head 4 and head 8 are the two attention heads that attend to the first variable position in the first layer attention when the number of variables is 3. When the number of variables is changed to 2, we observe that head 4 now attends to the [`sep`] token, while head 8 attends to the equal sign token of the previous equation. This indicates that the equal sign token and the [`sep`] token act as *attention sink* for head 4 and head 8, respectively.

E.1.3 First Layer MLP Residual Stream Does Not Change the Residual Stream Significantly

We measure the changes brought by the first MLP layer to the residual stream by computing the Relative L2 Error as:

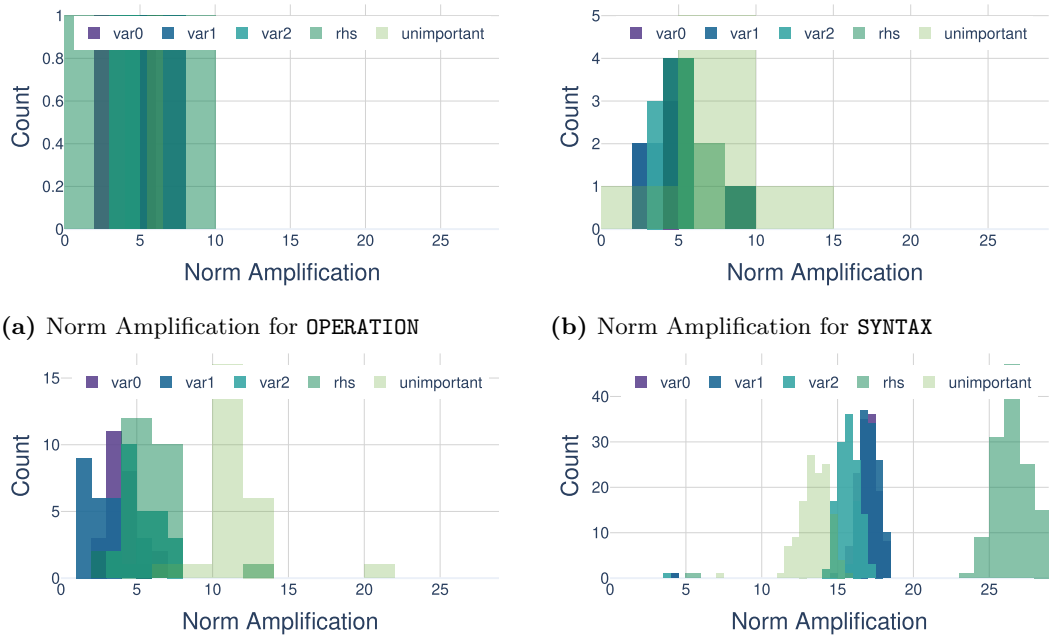
$$\text{L2 Relative Error} = \frac{\|\text{Residual Before MLP} - \text{Residual After MLP}\|_2}{\|\text{Residual Before MLP}\|_2}.$$

This metric quantifies how much the MLP alters the original residual signal. Figure 19 illustrates the heatmap of the relative L2 error computed at the $\langle$`rhs`$\rangle$ position across a set of 256 samples. We observe that the relative L2 error is relatively small, which indicates that the MLP does not change the residual stream significantly.

E.2 Second Layer Attention: Value Copying

A Hypothesis on the Second Layer Attention Heads. As we have shown previously, the first layer attention heads copy the `variable` factored embeddings of the variable to the RHS position, which tells the model the identity of all the variables on the LHS of an equation. To compute the final answer for the RHS position, the model still needs to copy the values of the variables $\langle$`var0`$\rangle$, $\langle$`var1`$\rangle$, and $\langle$`var2`$\rangle$ to the RHS position. Thus, we hypothesize that the second layer attention heads will copy the values of the variables $\langle$`var0`$\rangle$, $\langle$`var1`$\rangle$, and $\langle$`var2`$\rangle$ to the RHS position.

The Second Layer Attention Heads also Have a Group Structure To test this hypothesis, we prepare data that contains probe equations of the same form in (9) and conduct a controlled experiment designed to analyze how attention heads respond to changes



(a) Norm Amplification for `OPERATION`

(b) Norm Amplification for `SYNTAX`

(c) Norm Amplification for `VALUE`

(d) Norm Amplification for `VARIABLE`

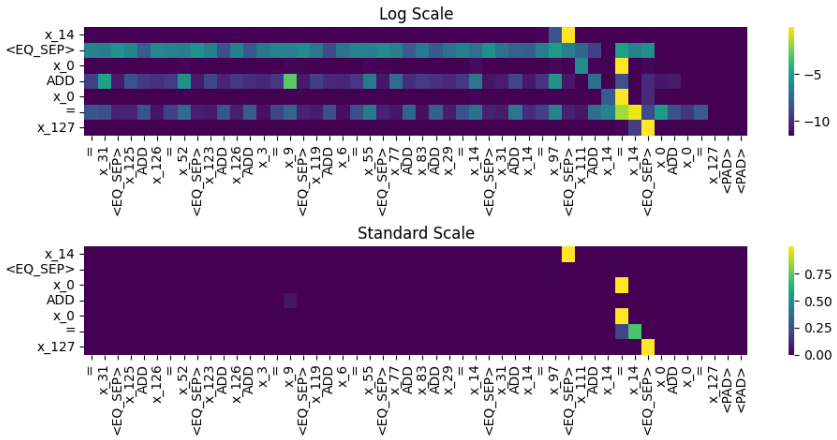
Figure 17. Histogram of norm amplification (defined in (11)) for the embeddings in the four factored embedding types {`syntax`, `variable`, `operation`, `value`} when passed through the first attention layer's combined OV matrix. Each subfigure contains five histograms in different colors, while each histogram corresponds to a different group of attention heads' combined OV matrix. Here, the 16 attention heads are grouped by the different variable they attend to, which are ⟨var0⟩, ⟨var1⟩, ⟨var2⟩, and ⟨rhs⟩, and an additional group for the heads that do not demonstrate a clear pattern.

in individual variable values. Different from the previous experiment where we change the variable identity, this time we fix the variable identity and only change the value of each variable ⟨`var0`⟩, ⟨`var1`⟩, and ⟨`var2`⟩ one at a time while keeping the other two variable values fixed. This is achieved by altering the previous equations that compute the value of the variable to be changed. Specifically, for each of the three variables ⟨`var0`⟩, ⟨`var1`⟩, and ⟨`var2`⟩, we conduct a separate experiment where we collect N samples by varying only that variable's value while keeping the other two variables fixed. Then, for each variable ⟨`var i`⟩, we collect the second layer attention head outputs across the N samples at the RHS position of the probe equation, and compute the following metrics:

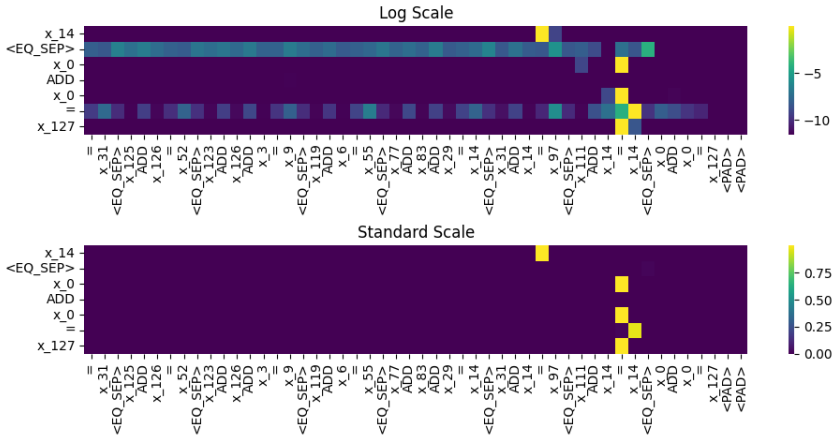
- The variance of the outputs (numerator in (10))
- The average squared norm (denominator in (10))
- The relative variance (ratio of the above quantities)

These metrics help us identify which heads are sensitive to changes in each variable's value. The results are shown in Figure 20. We deduce from the results (especially the relative variance) that the Heads (0, 8, 15) form the first group, which copy the value for ⟨`var0`⟩; Heads (5, 10) form the second group, which copy the value for ⟨`var1`⟩; and Heads (2, 3, 4, 7, 9) form the third group, which copy the value for ⟨`var2`⟩.

Second Layer Attention Heads Copy the Values of the Variables to the RHS Position. Similar to the experiment in the first layer, we compute the norm amplification coefficient for the second layer attention heads' OV matrix, combined by groups, as shown in Figure 21. We observe that the norm amplification coefficient for the `value` factored embedding is significantly larger than that of the other types of embeddings, confirming our



(a) Head 4 for equation with 2 variables



(b) Head 8 for equation with 2 variables

Figure 18. Visualization of the attention maps for the head group that attends to the first variable position in the first layer attention, which includes head 4 and head 8. Each row corresponds to a different query position, and each column corresponds to a different key position. We only show the rows within the last probe equation, and the columns within the last 50 positions in the sequence. Here, we notice that at the RHS query position (token $\langle x_{127} \rangle$ in the last row), head 4 attends to the [sep] token and attention head 8 attends to the equal sign token

hypothesis that the OV matrix is responsible for copying the `value` factored embeddings of the variable to the RHS position.

E.3 Second Layer MLP: Module Addition in the Frequency Domain

Extracting the Copied `value` Factored Embeddings. After confirming that the second layer attention heads copy the values of $\langle \texttt{var0} \rangle$, $\langle \texttt{var1} \rangle$, and $\langle \texttt{var2} \rangle$ to the RHS position, we now look closer at how the `value` factored embeddings of all three variables coexists in the residual stream after the second layer attention. To do so, we need to determine what should be the "`value`" factored embedding after passing through the second layer attention. Note that in the second layer attention, the copied `value` factored embedding for each $\langle \texttt{var i} \rangle$ are passed through the OV combined weight matrix for the corresponding group of attention heads, where the group structure is already determined in the previous experiment. For this reason, we can define the new "`value`" factored embedding at the output of the second layer attention as:

$$\texttt{new value}(i) = W_{OV}^{(\mathcal{H}_i)} \cdot \texttt{value}(i),$$

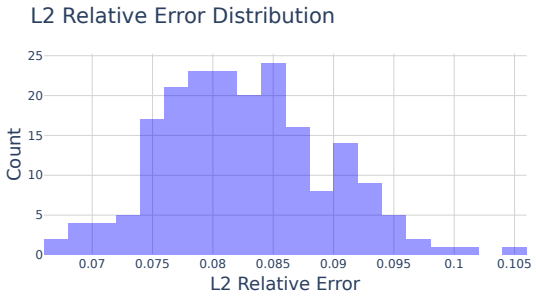
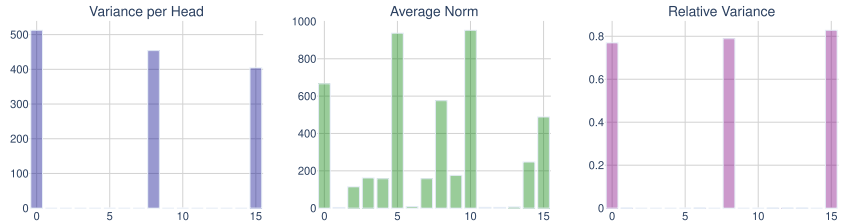
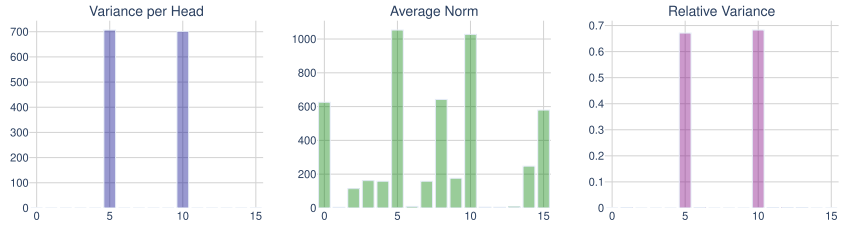


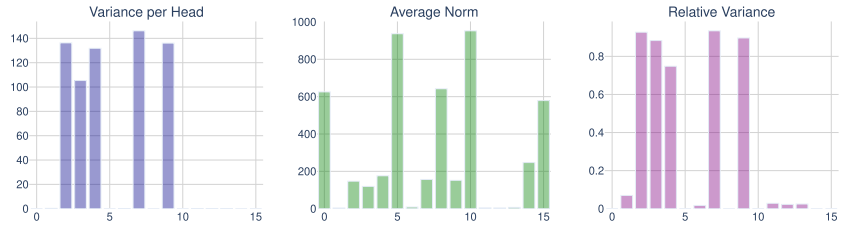
Figure 19. Histogram of the L2 relative error between the residual stream before and after the first layer MLP.



(a) ⟨`var0`⟩ statistics



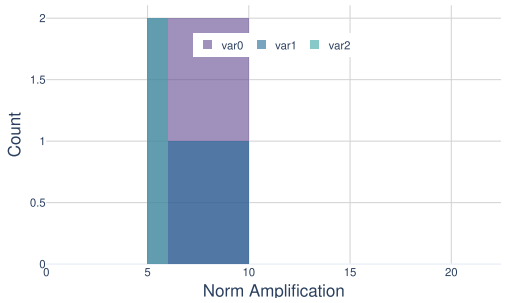
(b) ⟨`var1`⟩ statistics



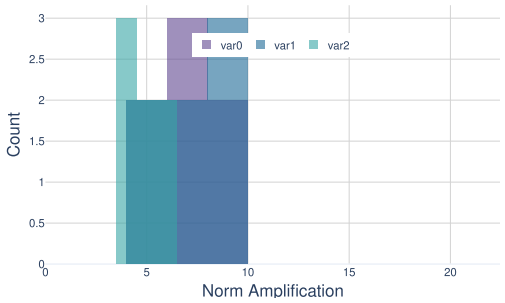
(c) ⟨`var2`⟩ statistics

Figure 20. Attention head statistics for the second layer attention. Each subfigure shows three histograms corresponding to the variance (numerator to (10)), average norm (denominator to (10)), and relative variance for each attention head's outputs.

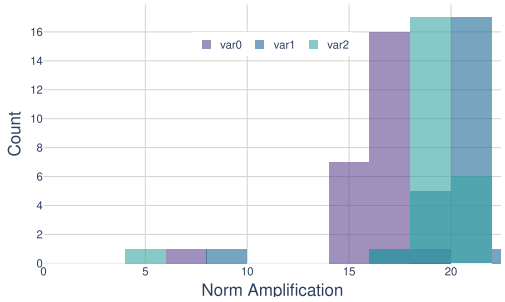
where `new value`(i) represents the new "`value`" factored embedding for ⟨`var i`⟩, and $\mathcal{H}_i$ represents the group of attention heads that copy the value of ⟨`var i`⟩ for $i = 0, 1, 2$. We also consider the same definition for the embedding of "`N/A`" and "`empty`" in the value factor. Therefore, we have in total 75 new "`value`" factored embeddings, where the first 25 are for



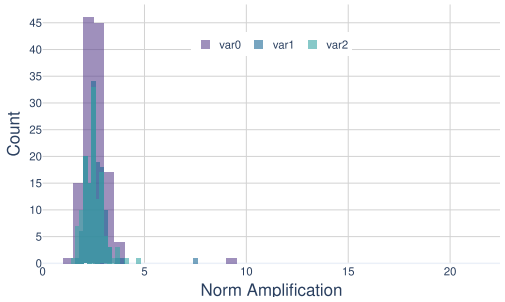
(a) Norm Amplification for `OPERATION`



(b) Norm Amplification for `SYNTAX`



(c) Norm Amplification for `VALUE`



(d) Norm Amplification for `VARIABLE`

Figure 21. Histograms of norm amplification for the four factored embedding types in the second layer attention's OV matrix. The 16 attention heads are grouped by the variable they attend to, which are ⟨var0⟩, ⟨var1⟩, and ⟨var2⟩. In each subfigure, we make three histograms each corresponding to the combined OV matrix for each group of attention heads. The three histograms in each subfigure are shown in different colors, and each histogram is for all the embeddings of the corresponding factored embedding type. It can be observed that the amplification factor for the "`value`" factored embedding is significantly larger than that of the other types of embeddings, confirming our hypothesis that the OV matrix is responsible for copying the "`value`" factored embeddings of the variable to the RHS position.

⟨`var0`⟩, the next 25 are for ⟨`var1`⟩, and the last 25 are for ⟨`var2`⟩. We plot the cosine similarity among the new "`value`" factored embeddings as shown in Figure 22.

From Figure 22, we can observe two interesting phenomena:

- The `value` factored embeddings for the three variables are almost orthogonal for two different variables, but not for the embeddings within the same variable.
- The `value` factored embeddings for the same variable show a periodic pattern.

The peoriodic pattern implies that the `value` factored embeddings for the same variable are likely formed by some combination of sin and cos functions. Therefore, it will be easier to understand the module addition operation in the frequency domain.

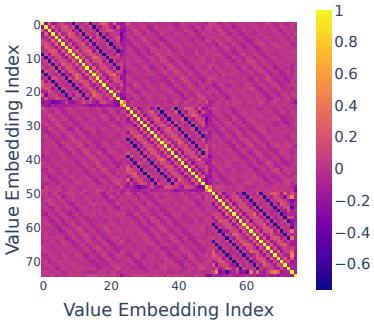


Figure 22. Cosine similarity of the `new value` factored embeddings for all three variables in the residual stream after the second layer attention.

Module Addition in the Frequency Domain. To systematically analyze how the model performs the module addition operation, we prepare a equation of the form as in (9), and we change the previous equations to alter the value of each variable ⟨var0⟩, ⟨var1⟩, and ⟨var2⟩. Specifically, we let ⟨var0⟩, ⟨var1⟩, and ⟨var2⟩ to iterate over the set $\{0, 1, 2, \ldots, 22\}$ since the model is trained on modular-23 addition. To study how the MLP performs the module addition operation, we pick the following four positions in the model: (i) pre-activation of the second layer's MLP, (ii) post-activation of the second layer's MLP, (iii) the output of the second layer's MLP, and (iv) the model's decoder output. For each of these positions, we take the vector obtained at the RHS position of the prepared equation, where we denote such vector as $v(a, b, c)$ with dimension d when the input variables are ⟨var0⟩ $= a$, ⟨var1⟩ $= b$, and ⟨var2⟩ $= c$. We then compute the 3-dimensional 23-point Discrete Fourier Transform (DFT) applied independently to each coordinate of v over (a, b, c), which is defined as:

$$\mathrm{DFT}_3(v)_{j,k,l} = \frac{1}{\sqrt{23^3}} \sum_{a=0}^{22} \sum_{b=0}^{22} \sum_{c=0}^{22} v(a, b, c)\, e^{-2\pi i \frac{aj+bk+cl}{23}}, \quad j, k, l = 0, 1, \ldots, 22,$$

The obtained DFT tensor is a 4D tensor with dimension $23^3 \times d$. We then compute the norm of the DFT tensor along the last dimension, which represents the magnitude of the corresponding frequency component. Since the obtained DFT tensor is conjugate symmetric, we have

$$\mathrm{DFT}_3(v)_{j,k,l} = \overline{\mathrm{DFT}_3(v)_{22-j,22-k,22-l}},$$

Therefore, we only need to focus on the first half of the tensor, which has dimension 12^3.

Studying the DFT Tensor by Frequency Group. We further partition the tensor into 7 groups by the algebraic patterns of the frequency component (j, k, l):

- Group 1: $(0, 0, 0)$
- Group 2: $(0, 0, a)$, $(0, a, 0)$, $(a, 0, 0)$ for $a \neq 0$
- Group 3: $(0, a, b)$, $(a, 0, b)$, $(a, b, 0)$ for nonzero $a \neq b$
- Group 4: $(0, a, a)$, $(a, 0, a)$, $(a, a, 0)$ for $a \neq 0$
- Group 5: (a, b, c) for nonzero $a \neq b$, $b \neq c$, $c \neq a$
- Group 6: (a, a, b), (a, b, a), (b, a, a) for nonzero $a \neq b$
- Group 7: (a, a, a) for $a \neq 0$

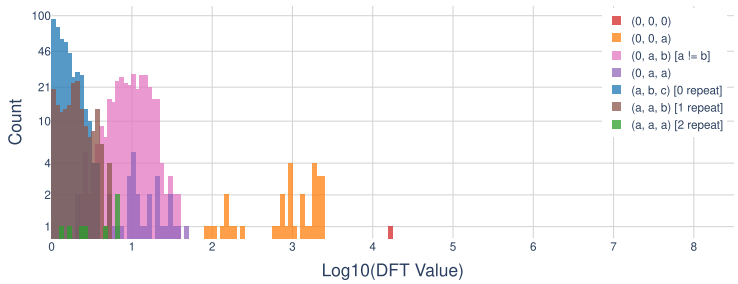
We then histogram the norm of the DFT tensor in the last dimension for each group, as shown in Figure 23. At the pre-activation stage of the second layer MLP, the DFT tensor shows its highest norm for the group $(0, 0, 0)$, which suggests a dominant bias term that is independent of the input variables. Progressing from the pre-activation (Figure 23a) to the MLP output (Figure 23c), this bias term gradually diminishes, while the norm corresponding to the group (a, a, a) steadily increases. This trend indicates that the MLP output contains a strong frequency component of the form

$$\cos\left(\frac{2\pi a x}{23}\right) \cdot \cos\left(\frac{2\pi a y}{23}\right) \cdot \cos\left(\frac{2\pi a z}{23}\right), \tag{12}$$

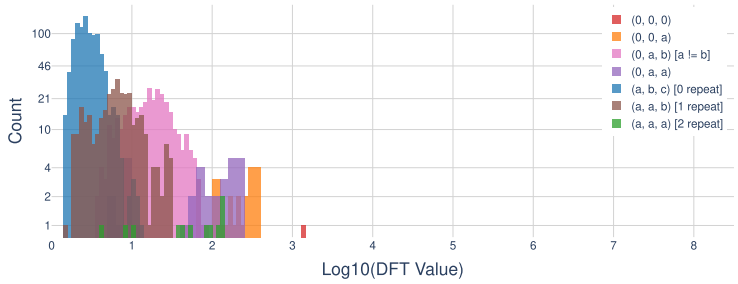
or a similar combination involving both sine and cosine functions with the same frequency a. In (12), x, y, and z denote the value of the three variables in the equation, and a is the frequency. The term in (12) corresponds to a degree-3 term on frequency a, indicating that the model is capable of computing terms in the form of $\cos(2\pi a(x + y + z)/23 + \varphi)$ for some frequencies a and phase φ, and eventually decodes to the correct answer $x + y + z \mod 23$.

E.4 Error Analysis

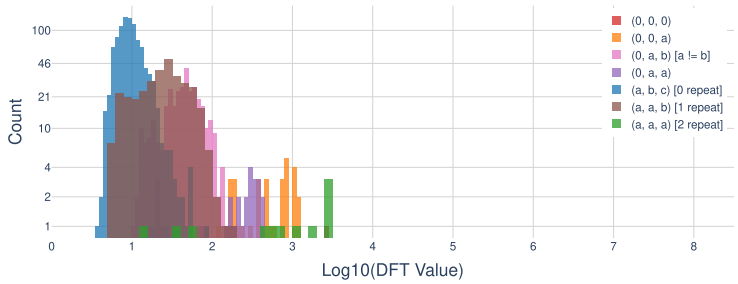
To better understand the probed model's performance, we analyze its prediction errors. As we have three functional components in the model—the first layer attention, the second layer attention, and the last feedforward layer—we consider three sources of errors: *(i)* the first layer's attention mapping copies from the wrong variable position, *(ii)* the second layer's attention fails to copy the correct variable value, and *(iii)* the feedforward layer miscalculates



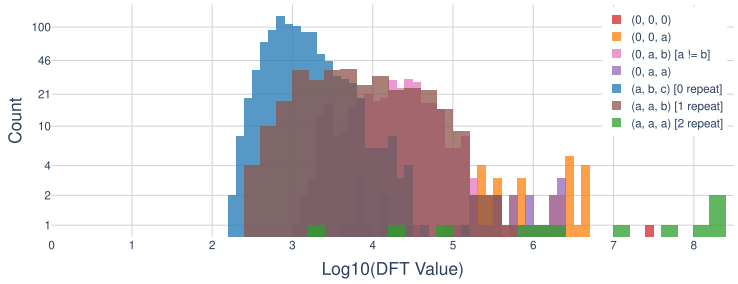
(a) DFT of L1 MLP pre-activation



(b) DFT of L1 MLP post-activation



(c) DFT of L1 MLP output



(d) DFT of decoder output

Figure 23. Combined DFT histograms for the second layer MLP pre-activation, MLP post-activation, MLP output, and decoder output.

the sum of the LHS variables. An account of the errors by source is shown in Table 2, where the major source of error is the feedforward layer calculation. Note that when considering the three sources of errors, if the error *(i)* occurs, we don't count towards error *(ii)* and *(iii)*. Similarly, when error *(ii)* occurs, we don't count towards error *(iii)*. In the following, we details how we identify the three sources of errors.

E.4.1 Identifying Different Sources of Errors

When our loop transformer model is computing the RHS value for all the equations in the sequence, we have two key concepts:

- **Depth of equation**: The depth of an equation is the number of iterations required to compute the correct RHS value. More formally, the depth of an equation is the depth of the RHS variable in the computation graph. Take Figure 1 as an example, the depth of the equation "$20 = x_7$" is 1, as the model only needs a single loop to compute the correct RHS value, and the depth of the equation "$x_7 + x_{42} = x_{23}$" is 2, as the model needs two loops to compute the correct RHS value.
- **Number of iterations**: The number of iterations describes how many times the loop transformer model has iterated over the input sequence.

By definition, the minimum number of iterations needed for computing the correct RHS value of an equation of depth d is at least d. In fact, we observe that most of the equations can be computed with exactly the number of iterations equal to the depth. For this reason, we only consider the equations and the number of iterations such that

$$\texttt{depth of equation} \geq \texttt{number of iterations}, \quad \text{or for short,} \quad \texttt{depth} \geq \texttt{iter}. \tag{13}$$

Moreover, we do not add any probe equations in this error analysis. This means that we apply the knowledge learned from the previous experiments with probe equations to identify errors happening in the whole sequence.

In the following, we details how we identify the three sources of errors.

First Layer Attention Error. We identify first layer attention errors by analyzing how well each attention head group focuses on its assigned variable position. For each equation's RHS position, we examine the attention map (an $H \times L \times L$ tensor, where H is the number of attention heads and L is the sequence length) to extract the relevant attention probabilities.

Table 2. Attribution of errors by source in the testing dataset with $N = 128$ and 23k sentences.

Error Source	Count
First Layer Attention Error	9
Second Layer Copy Error	1
Feedforward Calculation Error	30
Total	**40**

Consider a concrete example: For the head group $\mathcal{H}_0$ that is responsible for attending to $\langle\texttt{var0}\rangle$, we look at the attention probabilities where the query is at the $\langle\texttt{rhs}\rangle$ position and the key is at the $\langle\texttt{var0}\rangle$ position, for all heads in $\mathcal{H}_0$. We then average these probabilities within the head group.

For each equation, we can use the above strategy to obtain a single **group-wise attention probability** for each head group at the $\langle\texttt{rhs}\rangle$ position. If this group-wise attention probability is less than our threshold of 0.9, we classify it as a first layer attention error, indicating that the head group failed to maintain sufficient focus on its designated variable position. In fact, the error analysis is not very sensitive to the choice of the threshold. As we will see later in Figure 24 (Top Row), the computed group-wise attention probability is either very close to 1 or very close to 0 (for $\langle\texttt{var0}\rangle$ and $\langle\texttt{var2}\rangle$, where $\langle\texttt{var1}\rangle$ has a slightly larger deviation from 1 on the high end). It is very easy to identify when an error occurs in the first layer attention.

Second Layer Copy Error. For the second layer attention, we analyze the attention head's output rather than the attention map. This approach is necessary because the "`value`" factored embedding from the first layer may be distributed across multiple positions, including special tokens (like delimiters or operators), rather than being confined to the original variable position. Fortunately, we already have the extracted "`new value`(i)" factored embeddings for each $\langle\texttt{var i}\rangle$ in the previous experiment. We thus treat these "`new value`(i)" factored embeddings as the ground truth value embeddings for $\langle\texttt{var i}\rangle$ in the second layer attention output.

For each equation containing $\langle\texttt{var i}\rangle$, we compute the cosine similarity between the ground truth value embedding for $\langle\texttt{var i}\rangle$ and the designated head group's output at the $\langle\texttt{rhs}\rangle$ position in the second layer attention. We call this cosine similarity the "group-wise cosine similarity". If the cosine similarity is less than our pre-determined threshold of 0.9, we

consider it a second layer copy error for that head group, indicating the model fails to copy the correct variable value to the RHS position.

Similar to the first layer attention analysis, the choice of threshold is not critical. As shown in Figure 24 (Middle Row), the cosine similarity between the second layer attention outputs and the target value embeddings exhibits a clear pattern: either very close to 1 for correct copies, or significantly lower for incorrect copies. This stark separation makes it straightforward to identify second layer copy errors.

Feedforward Calculation Error. The feedforward calculation error is defined in the following way: If an equation passes the first two error checks, meaning that the first layer attention successfully attends to the correct variable position, and the second layer attention successfully copies the correct variable value to the RHS position, but the model still makes a mistake when applying the factored decoder after the second layer MLP, we consider it a feedforward calculation error.

An account of the errors by source is shown in Table 2, where the major source of error is the feedforward layer calculation. Overall, the model demonstrates a remarkable accuracy, where the total number of errors is only 40 out of 23k examples. A more detailed analysis of the errors is shown in Figure 24.

E.4.2 Additional Error Analysis

Here, we provide additional evidence for the above discussion. In Figure 24, instead of just counting the number of times a specific error occurs, we histogram all the statistics used by the above error analysis procedure. Figure 24 (Top Row) is a histoplot of the the group-wise attention probability in the first layer, organized by three head groups $\mathcal{H}_0$, $\mathcal{H}_1$, and $\mathcal{H}_2$, where each $\mathcal{H}_i$ is responsible for copying the value of $\langle \texttt{var i} \rangle$. See the "First Layer Attention Error" paragraph above for more details. We see that the attention scores generally concentrate their probability mass around 1 on the correct variable; however, the heads responsible for copying $\langle \texttt{var2} \rangle$ are somewhat less concentrated, resulting in more errors. Moreover, for some examples where the final prediction is incorrect, we observe a clear error pattern in the histogram: the attention head group completely fails to attend to the correct variable position, with the group-wise attention probability dropping to nearly 0. This stark contrast between successful and failed attention patterns makes it easy to identify first layer attention errors.

In addition, Figure 24 (Middle Row) histogram the cosine similarity between the second layer attention outputs and the target value embedding, again for all three head groups. For most examples, the cosine similarity is close to 1, showing that the second layer retrieves the value embeddings. However, for some examples where the final prediction is incorrect, we also observe a clear error pattern in the histogram: the cosine similarity drops to nearly 0. This stark contrast between successful and failed second layer copy patterns makes it easy to identify second layer copy errors as well.

Does the Model Perform Self-Correction? The first two rows in Figure 24 are reported only for equations with $\texttt{depth} \geq \texttt{iter}$. This is because the number of iterations required for computing the correct RHS value of equations is at most its depth. However, if we let the number of iterations go beyond the depth of the equations, as shown in Figure 24 (Bottom Row), the first layer attention heads are not able to concentrate their probability mass on the correct variable. This finding indicates that there is no further computation performed by the model at an equation position after the number of iterations reaches the depth of the equations, hence the model does not perform self-correction. One possible explanation for this to happen is the use of weight-decay in the training process. As the value for the $\langle \texttt{rhs} \rangle$ variable is already computed after the number of iterations reaches the depth of the equations, the model can directly pass on the computed value to the next iteration via the residual stream without any further computation.

How to let the model perform self-correction? We observe that the model does not perform self-correction because we only train the model on "perfect" data, where the model has no need to perform any further computation beyond the depth of the equations. In fact, we can let the model perform self-correction by training the model on "imperfect" data, where the model has to perform some further computation beyond the depth of the equations. This motivates our proposal of *Discrete Latent Space Supervision* ↻ method, which trains the model with corrupted data to teach the model to recover from errors. Consequently,

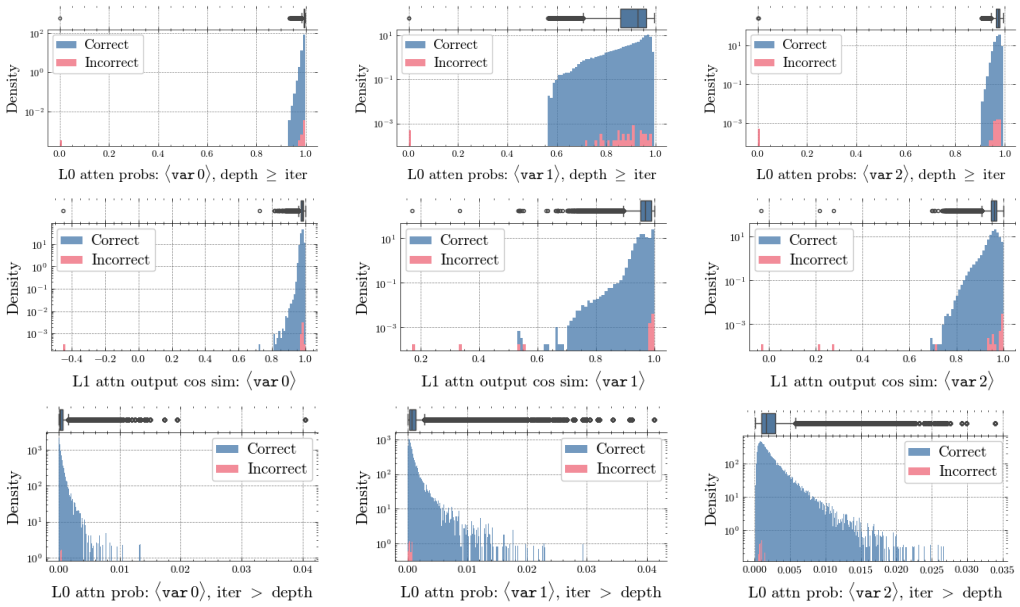


Figure 24. Error analysis. **Top Row.** Histograms for the group-wise attention probability in the first layer for all three head groups attending to ⟨var0⟩, ⟨var1⟩, and ⟨var2⟩, respectively. Here, the target equations considered all satisfy `depth` $\geq$ `iter` as defined in (13). We use different colors to separate the equations based on whether the decoded RHS value is correct or not after the second layer MLP. **Middle Row.** Histograms of the group-wise cosine similarity for the second layer attention head groups' outputs with the target values' embedding. Only equations with `depth` $\geq$ `iter` are included. **Bottom Row.** Histograms of the group-wise attention probability in the first layer for all three head groups. Here, the target equations considered all satisfy `depth` $<$ `iter`, meaning that the number of iterations is beyond the depth of the equations.

increasing the number of iterations beyond the depth of the input can be useful because it allows the model to correct any errors in previous iterations.