

Skip a Layer or Loop It? Learning Program-of-Layers in LLMs

Ziyue Li¹ Yang Li Tianyi Zhou²

Project: <https://github.com/tianyi-lab/PoLar>

Abstract

Large language models (LLMs) perform inference by following a fixed depth and order, non-recurrent execution of all layers. We reveal the wide existence of training-free, flexible, dynamic “program-of-layers (POLAR)”, where pretrained layers can be packed as modules and then skipped or looped to form a customized program for each input. For most inputs, substantially shorter program executions can achieve the same or better accuracy, while incorrect predictions of the original LLM can be corrected by alternative programs with fewer layers. These observations indicate that inference admits multiple valid latent computations beyond the standard forward pass. To efficiently achieve POLAR in practice, we propose a lightweight POLAR prediction network, which learns to generate execution programs that dynamically skip or repeat pretrained layers for each input. Experiments on mathematical reasoning benchmarks demonstrate that POLAR consistently improves accuracy over standard inference and prior dynamic-depth methods, often while executing fewer layers, and that these gains persist under out-of-distribution evaluation. Our results suggest that fixed-depth execution captures only a narrow subset of an LLM’s latent reasoning capacity.

1. Introduction

Generalist foundation models, e.g., LLMs and VLMs, uniformly deploy a static, pre-defined architecture to all inputs, despite their diversity and high variance in complexity and difficulty (Liu et al., 2020; Xin et al., 2020; Zhou et al., 2020; Liu et al., 2021a). In contrast, conventional problem

¹University of Maryland, College Park, MD, USA
²MBZUAI, Abu Dhabi, UAE. Correspondence to: Tianyi Zhou <tianyi.zhou@mbzuai.ac.ae>.

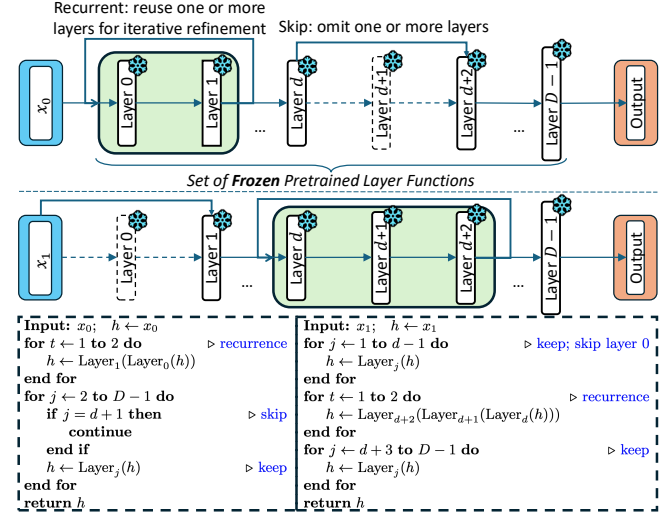


Figure 1. **Program-of-layers (POLAR) for two different inputs.** The D layers in a pretrained LLM define D functions $f_0, \dots, f_{D-1}$. Instead of calling them in a static fixed order from f_0 to f_{D-1} , the dynamic inference of POLAR executes an *input-specific program* $\pi = (i_1, \dots, i_K)$ that calls the functions with *layer skipping* and *recurrence*. POLAR enables a training-free architecture of dynamic depth for different inputs, yielding diverse latent computations that cannot be fully covered by existing methods.

solving by programs can be more flexible and adaptive in algorithmic structures and complexity. For example, an experienced programmer can save more steps and compute on easier tasks, and meanwhile knows how to scale up the space/time complexity to address more challenging problems. However, these programs are specifically designed and optimized for every problem class, so they are not as general as LLMs. This raises the questions: *Is it always optimal and efficient to apply the same architecture or “program”, i.e., forward pass through all the layers in a fixed order, to different tasks? Can a generalist model further optimize its “program” applied to each input?*

In this paper, we formulate layers in a pretrained LLM as a library of atomic functions that a program can call in arbitrary order for arbitrary times. This formulation allows us to represent a dynamic model architecture during inference as a program-of-layers (POLAR) for each input, as

illustrated in Figure 1. As the first empirical study of its kind, we investigate POLAR beyond the standard forward pass by Monte-Carlo Tree Search (MCTS) and find that better (more accurate and/or shorter) programs almost always exist for every input task evaluated. Unlike previous works on layer-skipping/recurrence, early exit, and looped transformer (Liu et al., 2020; Xin et al., 2020; Zhou et al., 2020; Fan et al., 2019; 2024; Yang et al., 2023), which only adopt one operation (either skip or repeat) to produce architectures of dynamic depths, our empirical study on the MCTS-searched programs reveals that searching in a joint space of layer-skip/repeat often discovers much better programs than those found in separate spaces. While most effective programs can be shorter than the default, increasing program complexity via skip/repeat operations can substantially improve the output quality, especially on more difficult tasks. In addition, most successful programs are predominantly composed of contiguous layer segments. These observations not only verify the broad existence of better POLAR without requiring any training, but also motivate a practical POLAR prediction method that avoids the expensive cost of MCTS in POLAR’s large search space. In particular, we aim to replace search-based program discovery with a direct, inference-time mechanism for generating execution programs. Instead of enumerating or exploring execution paths for each input (Li et al., 2025), our goal is to predict an input-specific program-of-layers that determines how pretrained layers are executed during inference. This shifts program selection from an online search problem to a single-shot prediction problem, enabling practical deployment of program-of-layers inference in LLMs.

To this end, we propose a POLAR algorithm that predicts execution programs over frozen pretrained layers at inference time. The predicted execution program specifies how pretrained layers are selectively skipped or recurrently applied and is executed once to produce the final output. This design offers several advantages. First, it makes program-of-layers inference computationally feasible by eliminating the need for expensive per-input search. Second, by jointly supporting layer skipping and recurrence within a unified execution framework, POLAR strictly generalizes prior dynamic-depth methods that are limited to a single form of execution control. Third, it enables flexible test-time computation scaling in fully frozen models, allowing inference to adapt to input difficulty while preserving model generality.

We evaluate POLAR on a range of mathematical reasoning benchmarks using multiple pretrained LLMs. Our results show that POLAR consistently improves accuracy over standard inference and prior dynamic-depth methods, often while executing fewer layers on average. Moreover, increasing the number of candidate execution programs yields strong test-time computation scaling, and execution programs learned on in-distribution data generalize effectively

to out-of-distribution benchmarks across diverse domains.

2. Dynamic Inference as a Program-of-Layers (POLAR) in Large Language Models

Inference in pretrained LLMs is implemented as a fixed-depth, fixed-order forward pass: every input is processed by executing the same sequence of transformer layers. Yet inputs to LLMs vary dramatically in difficulty. Some are answered correctly with minimal reasoning, while others require complex, multi-step computation. This discrepancy raises a basic question: **Is the standard forward pass sufficient for correct inference across diverse inputs?**

One possibility is that this fixed computation is indeed sufficient for all cases. Another is that correct prediction requires input-dependent variation in computation. In this work, we investigate the latter possibility. Such variation can occur either in token space, through longer and more explicit chains of thought, or within the model’s hidden states, a form of computation we refer to as *latent reasoning*.

Conjecture: Inference as Program-of-Layers

Define the layers in a pretrained LLM as functions, for each input, there can exist multiple distinct executions of programs-of-layers (beyond the standard forward pass) that produce correct predictions.

Inference as the execution of a program. In this view, inference is a step-by-step procedure that selects and composes pretrained modules. The execution may vary across inputs in both length and order, while each module remains a fixed, pretrained function.

Consider a pretrained LLM with D transformer layers, where each layer defines a fixed computation function

$$f_i : \mathbb{R}^{T \times d} \rightarrow \mathbb{R}^{T \times d}, \quad i \in \{0, \dots, D-1\}.$$

A program is defined as a finite sequence of layer indices

$$\pi = (i_1, i_2, \dots, i_K), \quad i_k \in \{0, \dots, D-1\},$$

which induces the composed computation

$$F_\pi = f_{i_K} \circ \dots \circ f_{i_1}.$$

Executing a program applies this composition to the input and produces a prediction. A program is considered *valid* if it yields a correct prediction for a given input.

Searching for valid execution programs. We explore the space of execution programs using MCTS. Execution programs are variable-length sequences over pretrained transformer layers, allowing both skipping and repetition. This space is large, discrete, and highly non-convex, making exhaustive search infeasible. MCTS provides a principled way to prioritize promising partial programs, enabling us to verify the existence of valid programs and analyze their structural properties. We use MCTS strictly as a diagnostic

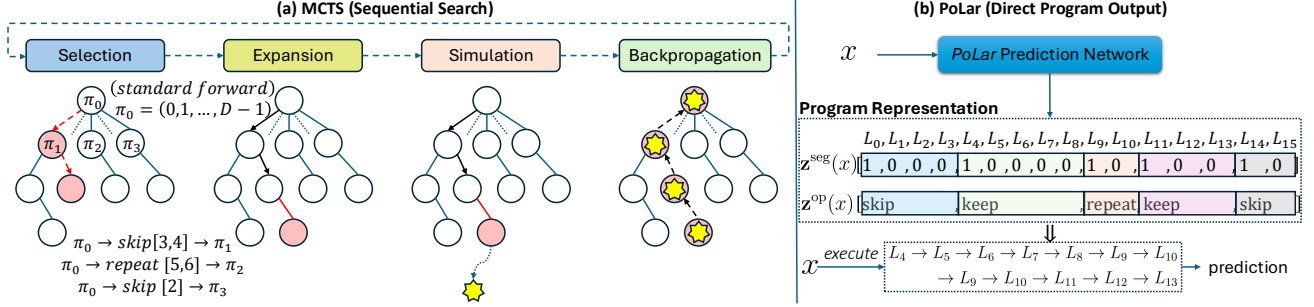


Figure 2. **Sequential MCTS (left) vs. End-to-end POLAR network (right) for prediction of programs.** (a) MCTS in the space of execution programs via sequential iterations of selection, expansion, simulation, and backpropagation. Each node represents a partial or complete execution program, and skip/repeat operations expand the search tree iteratively. This explicit and thorough search is expensive and impractical. (b) Our POLAR trains an end-to-end, lightweight prediction network that directly produces a program representation composed of (i) a binary mask $z^{\text{seg}}(x)$ segmenting layers into modules, and (ii) a vector of operation labels $z^{\text{op}}(x)$ that applies one operation out of *skip*, *keep*, or *repeat* to each module. Our method is scalable in practice and does not require sequential search.

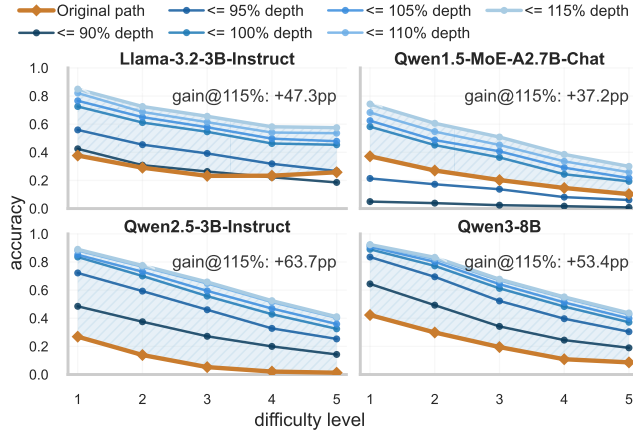


Figure 3. **Accuracy of MCTS discovered programs under varying execution-depth budgets** across five difficulty levels in DART-Math. We compare the original forward pass (orange) with 90–115% depth-budgeted programs (blue). Shaded regions denote the maximum gain achieved under the highest budget (115%).

tool rather than a practical inference-time method; implementation details are given in Appendix B. All experiments are conducted on **DART-Math** (Tong et al., 2024), a structured mathematical reasoning benchmark with five difficulty levels (DM-1 to DM-5). We evaluate four pretrained transformer models: *LLaMA-3.2-3B-Instruct*, *Qwen1.5-MoE-A2.7B-Chat*, *Qwen2.5-3B-Instruct*, and *Qwen3-8B*.

We summarize our empirical findings below.

Finding 1

Layer recurrence-only performs better than layer skipping-only, but combining the two complementary operators produces the best program-of-layers.

As shown in Table 1, execution programs that allow layer recurrence (**Loop**) consistently outperform those that allow

only layer skipping (**Skip**) across all evaluated models and difficulty levels. Moreover, combining skipping with recurrence (**Skip&Loop**) yields substantially larger gains than either operation alone, achieving the highest accuracy in every setting reported in Table 1. These results indicate that recurrence provides a stronger mechanism for improving inference than skipping alone, while the two operations play complementary roles when combined.

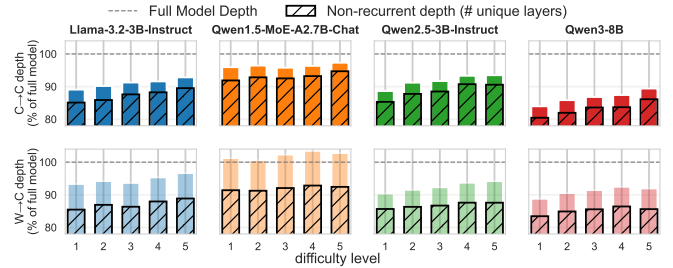


Figure 4. **Latent execution programs often admit shorter valid solutions.** We compare the standard forward-pass depth with that of MCTS-discovered valid programs, for initially correct (C→C) and initially incorrect (W→C) inputs. Bars report total execution depth as a fraction of full model depth, with hatched overlays indicating effective depth (the number of unique layers).

Finding 2

(Occam’s razor) Most valid execution programs are often shorter than the standard forward pass.

As shown in Figure 3, which reports the best accuracy obtained by MCTS under explicit budgets on total layer executions, many inputs remain solvable even when the overall computation is constrained to be significantly shorter than the standard forward pass. Consistent with this trend, Figure 4 shows that across models we frequently discover valid execution programs that require fewer layer applications than standard inference. In particular, among inputs already

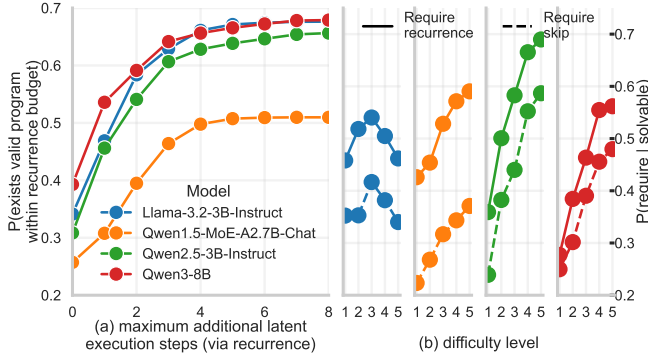


Figure 5. (a) Test-time scaling via recurrence over layer segments. Allowing more latent execution steps through segment recurrence leads to a monotonic increase in the probability of discovering valid execution programs across models. **(b) Recurrence and skipping are increasingly demanded for harder inputs.** The fraction of inputs relying on layer recurrence or skipping to be solved increases with increasing difficulty for most models, except LLaMA-3.2-3B-Instruct, whose deviation is explained by effective difficulty in Table 1.

solved correctly by standard inference ($C \rightarrow C$), 71.9% admit shorter valid programs. Even for inputs initially solved incorrectly ($W \rightarrow C$), 34.0% admit shorter programs that correct the model’s prediction. These results indicate that standard inference often over-computes, and that correct inference can frequently be achieved with substantially fewer latent computation steps.

Finding 3

Increasing latent execution complexity systematically expands the space of valid programs and improves inference on harder inputs.

While simple inputs often admit short execution programs (Finding 2), harder inputs demand greater test-time execution complexity. Across models and datasets, increasing execution depth and structural flexibility systematically expands valid program space and improves inference accuracy.

(a) Test-time scaling expands the space of valid execution programs for latent reasoning. As shown in Figure 5(a), allocating more test-time computation—via recurrence—monotonically increases the existence of valid execution programs across models. This establishes test-time scaling at latent reasoning: greater computation yields a larger feasible program space and higher correctness.

(b) Harder inputs require more complex execution programs. Figure 5(b) shows that the fraction of solvable inputs that require recurrence and/or skipping generally increases with dataset difficulty for most models. As task difficulty grows, valid execution programs become more constrained and increasingly rely on non-trivial execution structures rather than standard inference. This indicates that

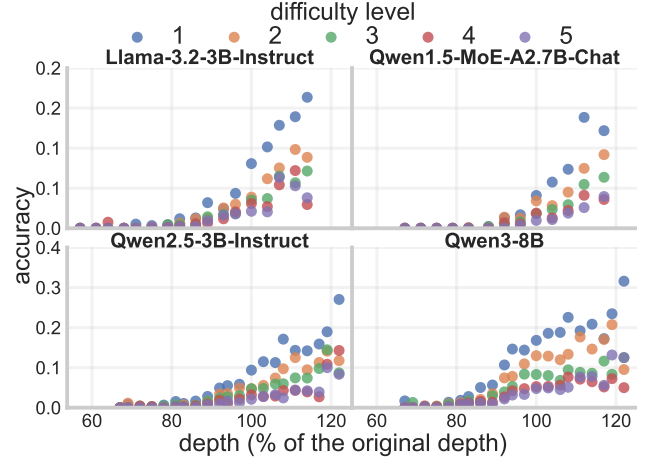


Figure 6. Accuracy vs. total layer executions. For each model, we report how the average accuracy of valid execution programs changes with the total number of layer executions (% of base model depth). Across models and difficulty levels, accuracy increases with executed layers, revealing a consistent effect of depth-scaling.

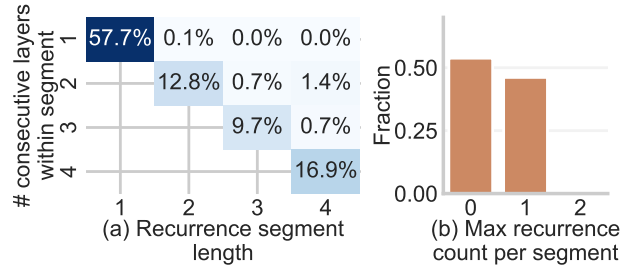


Figure 7. Structural bias of valid execution programs. Valid programs rely primarily on contiguous layer segments as modules (a) and require at most one recurrence of each module (b).

higher latent execution complexity is not merely helpful, but often necessary for solving harder inputs. The different trend observed for LLaMA-3.2-3B-Instruct is explained by a mismatch between dataset-defined difficulty and the model’s effective difficulty (Table 1).

(c) Inference accuracy improves systematically with execution depth. As shown in Figure 6, for all models and difficulty levels, the average accuracy of valid execution programs increases with total execution depth, measured relative to the original forward-pass depth. This reveals a consistent computation–accuracy trade-off: while many inputs admit short execution programs (Finding 2), harder inputs benefit from—and often require—deeper or recurrent execution to achieve correct inference.

Latent execution programs reveal a continuum of test-time inference behaviors that standard inference cannot access. Allocating more execution complexity enables harder inputs to be solved and yields higher accuracy.

Table 1. Accuracy (%) of programs searched for DART-Math (Diff 1–5) in different spaces: Base (standard forward), Skip (layer skipping), Loop (layer recurrence), and Skip&Loop (skipping + recurrence). **Gain** reports the absolute gain of **Skip&Loop** over Base. **Bold** indicates the best result, and underline indicates the second-best result in each row.

Metric	Base	Skip	Loop	Skip&Loop	Gain
LLaMA-3.2-3B-Instruct					
DM-1	37.7	45.3	<u>55.0</u>	85.0	+47.3
DM-2	29.1	35.9	<u>46.6</u>	72.6	+43.5
DM-3	23.2	29.4	<u>38.0</u>	65.6	+42.4
DM-4	23.4	29.0	<u>36.0</u>	58.2	+34.8
DM-5	25.8	30.2	<u>37.5</u>	57.5	+31.7
Qwen1.5-MoE-A2.7B-Chat					
DM-1	37.2	42.7	<u>57.3</u>	74.7	+37.5
DM-2	27.1	33.0	<u>44.9</u>	61.5	+34.4
DM-3	20.3	24.4	<u>35.1</u>	52.0	+31.7
DM-4	14.6	17.2	<u>26.0</u>	39.5	+24.9
DM-5	10.3	12.8	<u>19.5</u>	31.3	+21.0
Qwen2.5-3B-Instruct					
DM-1	26.9	47.6	<u>60.9</u>	87.8	+60.9
DM-2	13.9	34.2	<u>46.0</u>	76.6	+62.7
DM-3	5.3	24.6	<u>36.9</u>	66.0	+60.7
DM-4	2.1	16.1	<u>24.1</u>	52.8	+50.7
DM-5	1.3	11.6	<u>16.9</u>	41.8	+40.5
Qwen3-8B					
DM-1	42.3	67.1	<u>70.3</u>	92.4	+50.1
DM-2	29.9	52.6	<u>58.3</u>	83.5	+53.6
DM-3	19.5	36.9	<u>41.9</u>	68.3	+48.8
DM-4	10.9	24.9	<u>30.2</u>	55.6	+44.7
DM-5	8.6	18.7	<u>22.5</u>	44.2	+35.6

Finding 4

Valid execution programs are predominantly composed of contiguous layer segments and typically require at most a single recurrence per segment.

As shown in Figure 7, valid execution programs discovered by pretrained models exhibit a strong structural bias toward simplicity. A segment denotes a set of layers executed as a unit and need not be contiguous, while recurrence always corresponds to re-execution of the same segment. We therefore analyze segment structure by measuring the number of consecutive layers within each segment. Most valid programs are dominated by highly local segments and involve at most a single recurrence; long-range jumps and deep iterative reuse are rare. Figure 7(a) shows that 57.7% of segments consist of a single layer, and over two-thirds contain at most two consecutive layers, whereas segments with predominantly non-consecutive layers account for less than 2.9% of cases. Consistently, Figure 7(b) indicates that most segments are repeated at most once. Together, these results reveal an inherent limitation of pretrained models as execution-program generators: their training objectives favor short-range, local reuse over rich program composition

and complex control flow.

These findings show that standard inference selects only one execution from a vast space of valid latent programs. While MCTS reveals this space, its reliance on sequential search over an exponentially large program space makes it impractical for inference. This motivates a different approach: rather than searching over programs at test time, we ask whether a lightweight model can directly predict execution programs. Figure 2 contrasts the MCTS-based sequential search with our proposed direct program prediction approach. In the remainder of this work, we pursue this learning-based alternative, retaining the benefits of latent program selection uncovered by MCTS while eliminating sequential search.

3. Learning Program-of-Layers (POLAR) in Large Language Models

Building on our empirical analysis (Section 2), we propose POLAR, a method for *programming* pretrained language models at inference time by predicting input-specific execution programs (Figure 2). POLAR dynamically segments and composes pretrained layers into reusable modules, enabling flexible computation without parameter updates.

3.1. Program Representation

We instantiate the function library using *packed modules*, which segment contiguous pretrained transformer layers into reusable computation units. For a pretrained model of depth D , an execution program specifies (i) a segmentation of layers into modules and (ii) an operation applied to each segment. Each execution program is represented by two discrete structures: a binary boundary mask encoding the segmentation, and an operation label vector specifying the segment-level operations.

Segmentation. We partition the D layers of a pretrained model into contiguous segments

$$[0 = s_1, s_2), [s_2, s_3), \dots, [s_M, s_{M+1} = D),$$

with each segment length bounded by $s_{j+1} - s_j \leq K_{\max}$. Segmentation is represented by a binary boundary mask

$$\mathbf{z}^{\text{seg}}(x) \in \{0, 1\}^D,$$

where $\mathbf{z}_i^{\text{seg}} = 1$ indicates that layer index i starts a new segment, and $\mathbf{z}_i^{\text{seg}} = 0$ otherwise.

We set $K_{\max} = 4$ based on empirical evidence. **Finding 4** in Section 2 shows that valid execution programs are dominated by short, contiguous layer segments. Bounding the segment length therefore captures the dominant local execution structures while substantially reducing the complexity of the program space. Although this representation restricts the set of admissible programs, it preserves the most prevalent compositional patterns in practice and enables stable learning with strong empirical performance.

Operations. For each segment $[s_j, s_{j+1})$, the execution program assigns one of three operations $\{\text{skip}, \text{keep}, \text{repeat}\}$, which determines how the segment is executed:

$\text{skip} : \emptyset$,

$\text{keep} : [s_j, \dots, s_{j+1} - 1]$,

$\text{repeat} : [s_j, \dots, s_{j+1} - 1, s_j, \dots, s_{j+1} - 1]$.

The **skip** operator omits a segment to reduce computation, while **repeat** applies a single additional pass. Operations are represented by a categorical label vector

$$\mathbf{z}^{\text{op}}(x) \in \{\text{skip}, \text{keep}, \text{repeat}\}^D,$$

where $\mathbf{z}_i^{\text{op}}$ is defined only when $\mathbf{z}_i^{\text{seg}} = 1$ (i.e., at segment start positions); labels at all other positions are ignored.

This operator set is intentionally minimal and empirically grounded. **Finding 4** in Section 2 shows valid execution programs rarely require more than a single re-execution within a segment, and that **skip** and **repeat** account for the most effective execution patterns, offering strong performance–efficiency trade-offs. The **keep** operator preserves the original computation when no modification is needed. Although our implementation allows at most one additional execution through **repeat**, the representation is not fundamentally limited to a single recurrence. The operation vocabulary can be extended to $\{\text{repeat-2}, \dots, \text{repeat-}k\}$ to support multiple recurrences per segment. We use a single-repeat operator because the MCTS traces in Section 2 show that effective programs rarely benefit from deeper repeated execution of the same segment. This choice keeps the prediction space tractable while covering the dominant valid programs.

3.2. Program-of-Layers (POLAR) Prediction Network

We train a lightweight predictor to output logits for the program representation defined in Section 3.1.

Architecture. Given an input x , we first encode it using a frozen embedding model (Qwen3-Embedding-0.6B), as token-level representations $\mathbf{H} = E(x) \in \mathbb{R}^{T \times d_q}$, where T is the token length and d_q is the hidden size of the embedding model. We project token representations to a working dimension d : $\tilde{\mathbf{H}} = \mathbf{H}\mathbf{W}_h \in \mathbb{R}^{T \times d}$.

Layer queries. We associate each pretrained transformer layer index $i \in \{0, \dots, D-1\}$ with a learnable embedding $\mathbf{e}_i \in \mathbb{R}^d$, and stack them as $\mathbf{E} \in \mathbb{R}^{D \times d}$. These embeddings act as layer-specific queries.

Cross-attention. We apply multi-head cross-attention with layer embeddings as queries and token embeddings as keys/values: $\mathbf{X} = \text{MHA}(\mathbf{Q}, \mathbf{K}, \mathbf{V})$, $\mathbf{Q} = \mathbf{E}$, $\mathbf{K} = \tilde{\mathbf{H}}$, $\mathbf{V} = \tilde{\mathbf{H}}$, where padding tokens are masked using the input attention mask. The output $\mathbf{X} \in \mathbb{R}^{D \times d}$ provides an input-conditioned representation for each layer index.

Cross-layer encoder. To model dependencies across model depth, we apply a lightweight transformer encoder over the

layer dimension:

$$\mathbf{X}' = \text{ENC}_{\text{layer}}(\mathbf{X}) \in \mathbb{R}^{D \times d}.$$

This enables self-attention across layers, allowing decisions at each layer to depend on global depth context.

Prediction heads. Two linear heads produce logits for segmentation boundaries and operations:

$$\ell^{\text{seg}} = \mathbf{X}'\mathbf{W}_{\text{seg}} + \mathbf{b}_{\text{seg}} \in \mathbb{R}^D, \ell^{\text{op}} = \mathbf{X}'\mathbf{W}_{\text{op}} + \mathbf{b}_{\text{op}} \in \mathbb{R}^{D \times 3}.$$

Supervision from Valid Execution Programs. We supervise training using valid execution programs collected offline via MCTS (Section 2). Each program is deterministically parsed into program representation, producing ground-truth segmentation and operation labels $\mathbf{z}^{\text{seg}}(x)$ and $\mathbf{z}^{\text{op}}(x)$ in the format defined in Section 3.1. When multiple valid programs are available for an input and at least one is shorter than the full model depth, we down-weight the loss of the full-depth execution. This choice follows **Finding 2**, which shows that shorter valid programs are preferred while still preserving supervision from the original computation.

Training Objective. We train the predictor to match the ground-truth execution program, specified by segmentation and operation labels $(\mathbf{z}^{\text{seg}*}(x), \mathbf{z}^{\text{op}*}(x))$. Let $p_i^{\text{seg}} = \sigma(\ell_i^{\text{seg}})$ and $\mathbf{p}_i^{\text{op}} = \text{SOFTMAX}(\ell_i^{\text{op}})$. Segmentation is supervised with binary cross-entropy over boundary indicators:

$$\mathcal{L}_{\text{seg}} = - \sum_{i=0}^{D-1} [\mathbf{z}_i^{\text{seg}*} \log p_i^{\text{seg}} + (1 - \mathbf{z}_i^{\text{seg}*}) \log(1 - p_i^{\text{seg}})].$$

Operation prediction uses a masked cross-entropy applied only at segment start positions. With mask $m_i = \mathbf{z}_i^{\text{seg}*}$, we compute

$$\mathcal{L}_{\text{op}} = - \sum_{i=0}^{D-1} m_i \cdot \log \mathbf{p}_i^{\text{op}}[\mathbf{z}_i^{\text{op}*}].$$

The final objective is $\mathcal{L} = \mathcal{L}_{\text{seg}} + \mathcal{L}_{\text{op}}$.

Inference-Time Program Decoding. At inference time, execution programs are decoded in two stages. First, segment boundaries are determined deterministically by thresholding the predicted segmentation logits ℓ^{seg} . If any resulting segment exceeds the maximum length constraint K_{max} , additional boundaries are inserted to enforce it, yielding segment start positions $\{s_j\}$. Conditioned on this segmentation, we compute operation log-probabilities at each segment start from the predicted logits:

$$\log p(o_j | x, s_j) = \log \text{SOFTMAX}(\ell_{s_j}^{\text{op}})[o_j].$$

Rather than selecting operations independently via local argmax, we apply a small beam search over segment-level operation choices to account for non-local interactions between segments and to ensure globally consistent execution programs. This search operates over a highly constrained space and produces a ranked set of candidate execution programs $\pi(x)$. Finally, each candidate program is mapped deterministically to a concrete executed program using the segment-to-path rules in Section 3.1.

Table 2. Pass@k accuracy under different inference strategies applied to LLaMA-3.2-3B-Instruct.

Method	p@	DM-1	DM-2	DM-3	DM-4	DM-5
LLaMA-3.2-3B-Instruct						
Base ($\tau=0$)	1	51.1	30.6	29.9	27.1	26.1
Base (sampling)	1	48.9	29.4	30.4	28.6	26.6
	2	53.9	33.2	34.2	32.0	31.2
	3	59.6	35.9	36.5	34.6	33.2
	4	61.7	38.9	38.7	36.5	33.5
ShortGPT	5	65.2	41.2	39.5	38.5	34.5
	1	16.3	7.1	5.3	6.5	5.3
	2	21.3	13.4	11.1	10.9	9.9
	3	24.1	14.2	11.6	12.0	11.7
MindSkip	4	25.5	15.1	12.9	12.2	12.2
	5	27.7	18.7	16.2	13.3	14.0
	1	14.2	6.5	4.8	6.5	4.8
	2	24.1	12.8	9.9	12.5	7.1
FlexiDepth	3	26.2	12.8	10.9	12.5	8.4
	4	29.1	13.6	12.7	12.8	8.9
	5	30.5	16.9	16.5	14.6	10.4
	1	2.1	0.0	0.3	0.5	0.5
DR.LLM	2	16.3	7.1	5.6	6.5	5.6
	3	19.9	12.2	8.9	8.1	8.9
	4	20.6	12.2	8.9	8.1	8.9
	5	24.8	14.5	10.9	8.6	10.4
POLAR	1	51.1	23.1	26.8	18.8	18.0
	2	57.4	27.6	30.9	19.0	19.3
	3	64.5	31.5	37.2	22.1	24.6
	4	65.2	36.5	42.5	28.6	29.4
POLAR	5	68.8	39.8	45.3	30.2	31.7
	1	54.6	32.3	32.7	29.7	26.9
	2	65.2	38.0	37.0	37.0	32.2
	3	67.4	41.2	41.8	39.6	36.3
POLAR	4	69.5	44.5	45.3	42.7	39.1
	5	74.5	46.9	47.6	45.1	41.9
Δ vs. Base (sampling)	5	+9.3	+5.7	+8.1	+6.6	+7.4

4. Experiments

We evaluate POLAR across both in-distribution and out-of-distribution benchmarks to assess whether learning *latent execution programs* provides a practical and transferable alternative to search-based test-time computation.

4.1. Experimental Setup

Models. We evaluate POLAR on a diverse set of pretrained, instruction-tuned LLMs spanning different architectures and scales: *LLaMA-3.2-3B-Instruct*, *Qwen1.5-MoE-A2.7B-Chat*, *Qwen2.5-3B-Instruct*, and *Qwen3-8B*. All models are used in a fully frozen setting with no parameter updates.

Datasets. We use **DART-Math** (Tong et al., 2024), a structured mathematical reasoning dataset with five difficulty levels (DM-1 to DM-5), as in-distribution benchmark. For out-of-distribution (OOD) evaluation, we use **ASDiv** (Miao et al., 2020) and **MAWPS** (Kadlčık et al., 2023), which focus on arithmetic word problems, as well as subject subsets from **MMLU-Pro** (Wang et al., 2024) spanning mathemat-

ics, natural sciences, social sciences, and humanities. These benchmarks differ substantially from DART-Math in both format and domain coverage.

In-distribution evaluation. For DART-Math, we first deduplicate examples within each difficulty level by question text and then adopt a difficulty-wise train/test split: each difficulty level is split independently, and models are trained and evaluated within the same difficulty distribution.

Out-of-distribution (OOD) evaluation. For OOD evaluation, POLAR is trained on the union of DART-Math training data across all difficulty levels and evaluated zero-shot. This setting directly tests whether POLAR learns transferable computation control strategies rather than heuristics specific to a dataset or difficulty level.

Metric. We report **pass@k accuracy**, defined as the probability that at least one of the top- k candidates produces a correct answer. For POLAR, the k candidates correspond to the top- k predicted execution programs selected via beam search. For sampling-based baselines, k corresponds to the number of stochastic decoding samples. Unless otherwise stated, OOD results are reported using pass@1.

Baselines. We compare POLAR against standard inference and representative dynamic-computation methods. **Base** ($\tau = 0$) uses greedy decoding with temperature $\tau = 0$. **Base (sampling)** samples k outputs using stochastic decoding with $\tau \in \{0.3, 0.7, 1.0\}$ and reports the best result across temperatures, increasing output diversity without altering internal execution. **DR.LLM** (Heakl et al., 2025) learns layer-routing policies from execution paths and applies them at inference time. **ShortGPT** (Men et al., 2025) statically prunes layers based on estimated importance, yielding a reduced-depth model. **MindSkip** (He et al., 2024) and **FlexiDepth** (Luo et al., 2025) learn router-based dynamic-depth policies, primarily optimized for inference efficiency. Several approaches, such as Mixture-of-Depths (Raposo et al., 2024), LaCo (Yang et al., 2024), and Mixture-of-Recursions (Bae et al., 2025), require substantial additional training or architectural modification. In contrast, POLAR performs lightweight test-time program selection without modifying pretrained model parameters.

More dataset and training details are in Appendix D.

4.2. Main Results

We evaluate in-distribution performance on DART-Math. Table 2 reports pass@ k results using *LLaMA-3.2-3B-Instruct*, with complete results provided in Appendix C.1.

Accuracy gains arise from improved latent execution within the frozen model. At pass@1, POLAR outperforms Base (sampling) across all difficulty levels. For example, on DM-1, accuracy improves from 48.9% to 54.6%, yielding

Table 3. Out-of-distribution (OOD) performance at pass@1. We report accuracy using Qwen1.5-MoE-A2.7B-Chat.

Method	ASDiv	MAWPS	MMLU-Pro												
			Math	Phys	Chem	Law	Eng	Other	Econ	Health	Psych	Bus	Bio	Phil	Hist
Base ($\tau=0$)	59.1	41.7	13.9	15.6	13.8	16.6	15.1	22.8	31.0	26.8	30.7	18.4	34.7	22.8	22.6
ShortGPT	2.3	0.6	3.8	0.4	4.2	3.3	3.8	2.6	2.4	2.2	2.8	4.9	2.9	1.8	4.7
MindSkip	0.0	0.0	0.9	1.2	1.0	1.9	1.3	1.3	0.6	0.7	1.3	0.8	1.7	0.6	1.8
FlexiDepth	0.0	0.0	2.1	3.5	2.5	4.4	3.3	2.5	1.7	3.5	2.0	2.9	3.8	3.0	3.7
DR.LLM	59.1	41.3	14.6	17.4	13.2	19.8	16.0	20.8	31.8	27.0	32.2	17.5	33.5	22.8	21.3
PoLAR	63.8	46.7	18.5	20.3	18.3	20.4	19.9	26.6	34.6	29.5	35.3	20.9	36.9	25.9	23.5

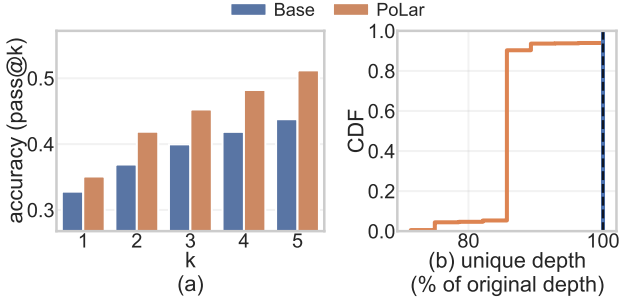


Figure 8. Pass@ k accuracy and unique depth on LLaMA-3.2-3B-Instruct, averaged across DART-Math DM-1 to DM-5. (a) reports macro-averaged pass@ k accuracy for Base (sampling) and PoLAR. (b) reports the CDF of the shortest successful execution program among the top-5 candidates, measured by unique executed layers as a percentage of the original model depth.

an absolute pass@1 gain of +5.7 percentage points. Since pass@1 evaluates a single decoded output, this gain reflects more effective latent execution selection rather than output-space diversity.

Exploring the execution-program space enables effective test-time scaling. Increasing the number of candidate execution programs (k) monotonically improves PoLAR, evidencing effective test-time computation scaling through execution-program exploration. Figure 8(a) illustrates this behavior on LLaMA-3.2-3B-Instruct, macro-averaged across DART-Math DM-1 to DM-5: Base (sampling) improves from 32.8% at pass@1 to 43.8% at pass@5, while PoLAR improves from 35.1% to 51.2%. Thus, at pass@5, PoLAR achieves an average absolute gain of +7.4 percentage points over Base (sampling). Crucially, Figure 8(b) shows that among successful top-5 candidates, PoLAR often finds execution programs that use fewer unique layers than a standard forward pass, indicating that the gains arise from better latent execution programs rather than simply executing the full depth. In contrast, Base (sampling) explores output-space diversity under a fixed computation graph and therefore always uses the original full-depth execution. These results show that structured execution-program exploration can outperform output sampling under the same frozen model.

Program-level execution exploration is more effective than local routing decisions. Existing dynamic-depth methods primarily make local, layer-wise routing decisions, which restrict inference to a limited execution space and often degrade accuracy in our setting. DR.LLM supports both layer skipping and repetition but operates at the individual-layer level, limiting global coordination across depth. In contrast, PoLAR formulates inference as *program-level* exploration over execution programs defined on packed contiguous segments, enabling coordinated skip and repeat patterns across depth. This design directly reflects the execution structures uncovered by MCTS, while replacing expensive search with a lightweight, learned predictor.

PoLAR incurs negligible inference overhead and reduces end-to-end latency. Beyond counting executed layers, we measure wall-clock latency on *Qwen1.5-MoE-A2.7B-Chat* with 24 layers. As shown in Table 4, the encoder, predictor head, and beam search introduce a total additional overhead of only 3.05 ms, corresponding to 0.8% of a standard forward pass and approximately 0.23 LLM layers. This overhead is small compared with the latency reduction achieved by executing fewer layers. Consequently, PoLAR reduces end-to-end latency while improving accuracy: it achieves $0.83\times$ the base runtime on easier inputs and $0.95\times$ on harder inputs. The learned predictor is also lightweight in parameter count: across all evaluated backbones, it contains approximately 2.1M parameters, corresponding to only 0.01%–0.06% of the base LLM size. Full parameter counts are provided in Appendix D.3.

4.3. Out-of-Distribution Performance

We evaluate the OOD generalization of execution programs learned from in-distribution data. As shown in Table 3, PoLAR consistently outperforms standard inference on all OOD benchmarks using *Qwen1.5-MoE-A2.7B-Chat*, with full results reported in Appendix C.2.

Execution programs learned on mathematical datasets transfer across domains. On arithmetic word problem benchmarks such as ASDiv and MAWPS, PoLAR achieves clear improvements over the standard forward pass. More notably, on MMLU-Pro, PoLAR improves accuracy across

Table 4. Component-wise inference overhead and end-to-end latency of POLAR on *Qwen1.5-MoE-A2.7B-Chat* with 24 layers. The additional components introduce negligible overhead compared with a standard forward pass, while POLAR reduces end-to-end latency and improves accuracy.

Component-wise overhead			
Component	Latency (ms)	Equiv. layers	% full forward
One LLM layer	13.23	1.00	3.5%
Predictor head	0.99	0.07	0.3%
Beam search	0.11	0.01	0.03%
Encoder	1.95	0.15	0.5%
Total additional overhead	3.05	0.23	0.8%
End-to-end latency			
Method	Avg. layers	Latency (ms)	Rel. / Acc. gain
Base	24.00	373.45	1.00× / –
POLAR (DM-1)	23.30	311.41	0.83× / +5.7
POLAR (DM-5)	23.76	353.31	0.95× / +9.4

diverse subject areas. We conjecture that this cross-domain transfer comes from two complementary sources. First, the external input representation maps examples from different domains into a shared semantic space, allowing the small POLAR prediction head trained on mathematics to generalize beyond its training distribution. Second, the predicted programs are constrained to simple structural patterns, namely contiguous segments with limited recurrence, which encourages reusable computation strategies rather than benchmark-specific execution heuristics.

5. Related Works

Transformers process inputs through sequential layer stacks, making layer-level computation reduction a critical research direction. Early-exit and layer skipping methods (Liu et al., 2020; Xin et al., 2020; Zhou et al., 2020; Liu et al., 2021a) dynamically terminate computation at intermediate layers using auxiliary classifiers and confidence metrics, allowing easy inputs to exit early. LayerSkip (Elhoushi et al., 2024) shares classifiers across layers to reduce overhead. LayerDrop (Fan et al., 2019) trains models so arbitrary layer subsets can be skipped during inference. ShortGPT (Men et al., 2025) assesses layer importance based on input-output similarity and drops low-importance layers. LaCo (Yang et al., 2024) merges layers using weight arithmetic. Recent work introduces learned routing for adaptive skipping: FlexiDepth (Luo et al., 2025) and MindSkip (He et al., 2024) attach lightweight routers to pretrained models for input-adaptive layer skipping.

In addition to skipping layers, another line of research explores layer reuse and recurrence. Universal Transformers (Dehghani et al., 2018) apply self-attention blocks recurrently with halting mechanisms to adapt depth per token. Recent looped transformers (Fan et al., 2024; Yang et al., 2023) repeatedly apply single blocks to achieve better length generalization on algorithmic tasks by adjusting loop counts

during inference. The Inner Thinking Transformer (Chen et al., 2025) interleaves adaptive loops with residual “thinking” connections and per-token routing, devoting extra computation to difficult tokens. While these approaches demonstrate the value of recurrence, they require architectural redesign and training from scratch.

Li et al. (2025) studies test-time depth adaptation by using search to dynamically skip or repeat pretrained transformer layers without finetuning. Their work demonstrates that alternative execution paths can improve inference, but the method remains search-based and requires expensive per-input program discovery. In contrast, we use MCTS only as an offline diagnostic tool to characterize the structure of the program space, and then replace search with a learned predictor that generates execution programs in a single shot.

Following this direction, DR.LLM (Heakl et al., 2025) learns routing policies from MCTS-generated supervision and supports both skipping and repeating layers. However, DR.LLM performs sequential layer-wise routing, where each decision is made locally during the forward pass and depends on intermediate hidden states. In contrast, POLAR predicts the entire execution program upfront, before executing the frozen LLM. This avoids interleaving routing with layer execution and enables more efficient inference. Moreover, DR.LLM is limited to single-layer recurrence, whereas POLAR operates on contiguous layer segments; for example, POLAR can represent multi-layer recurrent modules such as $4 \rightarrow 5 \rightarrow 4 \rightarrow 5$, which are outside the single-layer routing space. Thus, POLAR provides a more coordinated and expressive program space while preserving fully frozen base model parameters.

6. Conclusion

We show that inference in LLMs need not be limited to a fixed-depth forward pass. By viewing pretrained transformer layers as reusable functions, we uncover multiple valid execution programs for a single input, many of which are shorter than standard execution and can correct model errors. Motivated by this insight, we introduce POLAR, a lightweight framework that predicts input-dependent execution programs by selectively skipping or repeating contiguous layer segments at inference time, without modifying model parameters. Across models and both in-distribution and out-of-distribution benchmarks, POLAR consistently outperforms standard inference and prior dynamic-depth methods. These findings suggest that fixed-depth execution captures only a narrow subset of an LLM’s latent reasoning capacity. Enabling flexible, programmatic execution over pretrained layers reallocates computation at inference time, offering a simple and effective route to more expressive and efficient inference in foundation models.

Impact Statement

This work adapts the internal computation of pretrained LLMs at inference time by dynamically skipping or repeating layer segments. Its main potential benefit is improved efficiency: POLAR can reduce unnecessary computation on easier inputs while allocating more latent computation to harder ones, lowering inference cost, latency, and energy use without retraining the base model. This may make capable LLMs more accessible to researchers and practitioners with limited compute, and may support more sustainable deployment of foundation models. As with other methods that improve LLM capability or efficiency, broader deployment may amplify both beneficial and harmful uses. Potential benefits include education, scientific reasoning, and software assistance, while potential misuse includes scalable generation of misleading or harmful content. These risks largely arise from the underlying pretrained models and their applications rather than from dynamic execution itself. Since POLAR changes the execution path in an input-dependent manner, future work may further study how such paths can be audited or interpreted. Our experiments focus on mathematical reasoning and related benchmarks. Before applying dynamic execution in high-stakes domains, future work should evaluate robustness, calibration, interpretability, and safety alongside accuracy and efficiency. We hope this work encourages more responsible test-time computation methods that improve model performance while making compute allocation more transparent and efficient.

References

- Andreas, J., Rohrbach, M., Darrell, T., and Klein, D. Neural module networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 39–48, 2016.
- Bae, S., Kim, Y., Bayat, R., Kim, S., Ha, J., Schuster, T., Fisch, A., Harutyunyan, H., Ji, Z., Courville, A., et al. Mixture-of-recursions: Learning dynamic recursive depths for adaptive token-level computation. *arXiv preprint arXiv:2507.10524*, 2025.
- Chen, Y., Shang, J., Zhang, Z., Xie, Y., Sheng, J., Liu, T., Wang, S., Sun, Y., Wu, H., and Wang, H. Inner thinking transformer: Leveraging dynamic depth scaling to foster adaptive internal thinking. *arXiv preprint arXiv:2502.13842*, 2025.
- Dehghani, M., Gouws, S., Vinyals, O., Uszkoreit, J., and Kaiser, Ł. Universal transformers. *arXiv preprint arXiv:1807.03819*, 2018.
- Elhoushi, M., Shrivastava, A., Liskovich, D., Hosmer, B., Wasti, B., Lai, L., Mahmoud, A., Acun, B., Agarwal, S., Roman, A., et al. Layerskip: Enabling early exit inference and self-speculative decoding. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 12622–12642, 2024.
- Eyzaguirre, C., Del Rio, F., Araujo, V., and Soto, A. Dact-bert: Differentiable adaptive computation time for an efficient bert inference. In *Proceedings of NLP Power! The First Workshop on Efficient Benchmarking in NLP*, pp. 93–99, 2022.
- Fan, A., Grave, E., and Joulin, A. Reducing transformer depth on demand with structured dropout. *arXiv preprint arXiv:1909.11556*, 2019.
- Fan, Y., Du, Y., Ramchandran, K., and Lee, K. Looped transformers for length generalization. *arXiv preprint arXiv:2409.15647*, 2024.
- Gordon, M., Duh, K., and Andrews, N. Compressing bert: Studying the effects of weight pruning on transfer learning. In *Proceedings of the 5th Workshop on Representation Learning for NLP*, pp. 143–155, 2020.
- He, S., Ge, T., Sun, G., Tian, B., Wang, X., and Yu, D. Router-tuning: A simple and effective approach for enabling dynamic-depth in transformers. *arXiv preprint arXiv:2410.13184*, 2024.
- Heakl, A., Gubri, M., Khan, S., Yun, S., and Oh, S. J. Dr. llm: Dynamic layer routing in llms. *arXiv preprint arXiv:2510.12773*, 2025.
- Jain, G., Hegde, N., Kusupati, A., Nagrani, A., Buch, S., Jain, P., Arnab, A., and Paul, S. Mixture of nested experts: Adaptive processing of visual tokens. *Advances in Neural Information Processing Systems*, 37:58480–58497, 2024.
- Kadlčík, M., Štefánik, M., Sotolár, O., and Martinek, V. Calc-x and calcformers: Empowering arithmetical chain-of-thought through interaction with symbolic systems. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pp. 12101–12108, 2023.
- Li, Z., Li, Y., and Zhou, T. Skip a layer or loop it? test-time depth adaptation of pretrained llms. *arXiv preprint arXiv:2507.07996*, 2025.
- Liu, W., Zhou, P., Wang, Z., Zhao, Z., Deng, H., and Ju, Q. Fastbert: a self-distilling bert with adaptive inference time. In *Proceedings of the 58th annual meeting of the association for computational linguistics*, pp. 6035–6044, 2020.
- Liu, Y., Meng, F., Zhou, J., Chen, Y., and Xu, J. Faster depth-adaptive transformers. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 35, pp. 13424–13432, 2021a.
- Liu, Z., Li, F., Li, G., and Cheng, J. Ebert: Efficient bert inference with dynamic structured pruning. In *Findings of the Association for Computational Linguistics: ACL-IJCNLP 2021*, pp. 4814–4823, 2021b.
- Luo, X., Wang, W., and Yan, X. Adaptive layer-skipping in pre-trained llms. *arXiv preprint arXiv:2503.23798*, 2025.
- Men, X., Xu, M., Zhang, Q., Yuan, Q., Wang, B., Lin, H., Lu, Y., Han, X., and Chen, W. Shortgpt: Layers in large language models are more redundant than you expect. In *Findings of the Association for Computational Linguistics: ACL 2025*, pp. 20192–20204, 2025.
- Miao, S.-Y., Liang, C.-C., and Su, K.-Y. A diverse corpus for evaluating and developing english math word problem solvers. In *Proceedings of the 58th annual meeting of the Association for Computational Linguistics*, pp. 975–984, 2020.
- Raposo, D., Ritter, S., Richards, B., Lillicrap, T., Humphreys, P. C., and Santoro, A. Mixture-of-depths: Dynamically allocating compute in transformer-based language models. *arXiv preprint arXiv:2404.02258*, 2024.
- Tang, S., Wang, Y., Kong, Z., Zhang, T., Li, Y., Ding, C., Wang, Y., Liang, Y., and Xu, D. You need multiple exiting: Dynamic early exiting for accelerating unified vision language model. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pp. 10781–10791, 2023.

- Tong, Y., Zhang, X., Wang, R., Wu, R., and He, J. Dartmath: Difficulty-aware rejection tuning for mathematical problem-solving. *Advances in Neural Information Processing Systems*, 37:7821–7846, 2024.
- Wang, Y., Ma, X., Zhang, G., Ni, Y., Chandra, A., Guo, S., Ren, W., Arulraj, A., He, X., Jiang, Z., et al. Mmlu-pro: A more robust and challenging multi-task language understanding benchmark. *Advances in Neural Information Processing Systems*, 37:95266–95290, 2024.
- Wu, Q., Ke, Z., Zhou, Y., Sun, X., and Ji, R. Routing experts: Learning to route dynamic experts in multi-modal large language models. *arXiv preprint arXiv:2407.14093*, 2024.
- Xin, J., Tang, R., Lee, J., Yu, Y., and Lin, J. Deebert: Dynamic early exiting for accelerating bert inference. *arXiv preprint arXiv:2004.12993*, 2020.
- Xu, G., Hao, J., Shen, L., Hu, H., Luo, Y., Lin, H., and Shen, J. Lgvit: Dynamic early exiting for accelerating vision transformer. In *Proceedings of the 31st ACM International Conference on Multimedia*, pp. 9103–9114, 2023.
- Yang, L., Lee, K., Nowak, R., and Papailiopoulos, D. Looped transformers are better at learning learning algorithms. *arXiv preprint arXiv:2311.12424*, 2023.
- Yang, Y., Cao, Z., and Zhao, H. Laco: Large language model pruning via layer collapse. *arXiv preprint arXiv:2402.11187*, 2024.
- Zhou, W., Xu, C., Ge, T., McAuley, J., Xu, K., and Wei, F. Bert loses patience: Fast and robust inference with early exit. *Advances in Neural Information Processing Systems*, 33:18330–18341, 2020.

A. Related Work

Layer Pruning and Early-Exit Neural Networks Many works aim to accelerate large Transformers by statically pruning weights or dynamically halting computation. Static pruning typically removes redundant neurons, heads, or layers after training. For example, [Liu et al. \(2021b\)](#) demonstrate that a significant fraction of BERT’s attention heads can be dropped with negligible performance loss, and [Gordon et al. \(2020\)](#) investigate fine-grained weight pruning in BERT. [Fan et al., 2019](#) introduce LayerDrop, a structured dropout technique that effectively trains models so arbitrary subsets of layers can be skipped during inference without requiring fine-tuning. These methods produce smaller models that trade computation for a small accuracy loss.

By contrast, early-exit or input-adaptive methods add auxiliary classifiers at intermediate layers so that “easy” inputs exit early. Notable examples include FastBERT ([Liu et al., 2020](#)) and DeeBERT ([Xin et al., 2020](#)), which insert classifiers after each block and use confidence or entropy metrics to decide when to stop. PABEE ([Zhou et al., 2020](#)) employs a patience criterion to halt when predictions stabilize. DACT-BERT ([Eyzaguirre et al., 2022](#)) adopts a differentiable Adaptive Computation Time mechanism to learn how many Transformer layers to run for each example. [Liu et al. \(2021a\)](#) estimate input “hardness” via mutual information or reconstruction error to pre-determine the number of Transformer layers to use.

These early-exit networks achieve significant speedups on NLP tasks by adaptively reducing depth per input. More recently, early-exit ideas have been extended to vision and multimodal Transformers. [Xu et al. \(2023\)](#) propose LGViT, which adds heterogeneous exit heads (local and global) to ViT so that vision transformers can terminate early with minimal feature loss. [Tang et al. \(2023\)](#) introduce MuE (“Multiple Exiting”), a strategy for unified vision-language models that dynamically skips layers in both encoder and decoder based on input similarity. These works demonstrate that later layers can be skipped to allow image and vision-language models to adapt computation per sample with minimal accuracy drop. Our work generalizes this approach by allowing skipping of arbitrary layers and enabling reuse of certain layers.

Looped Transformer and Recurrent Depth Another line of research makes Transformer depth adaptive by looping or repeating layers. The Universal Transformer ([Dehghani et al., 2018](#)) was an early example: it applies the same self-attention block recurrently and uses a halting mechanism to determine when each position is “done” (adapting depth per token). Building on these ideas, recent work explicitly introduces loops in model architectures. [Fan et al. \(2024\)](#) demonstrate that a Looped Transformer – a single Transformer block applied repeatedly – can achieve much better length generalization on algorithmic tasks by adjusting the number of loops during inference. Similarly, [Yang et al. \(2023\)](#) note that looped architectures excel at learning algorithms by explicitly incorporating iterative characteristics into the transformer architecture. More sophisticated variants like the Inner Thinking Transformer [Chen et al. \(2025\)](#) interleave adaptive loops with residual “thinking” connections and per-token routing, enabling the model to devote extra computation only to particularly difficult tokens. In summary, these approaches explore recurrent or elastic depth via explicit loops to tailor the number of applied layers to each input’s complexity. Unlike our approach, they require special architecture design and training from scratch, whereas our work focuses on pure test-time adaptation.

Dynamic Routing and Modular Inference A third theme treats networks as collections of modules or experts with dynamically chosen pathways per sample. Mixture-of-Experts (MoE) Transformer layers are a well-known example: they maintain multiple sub-networks (“experts”) and route each token to a subset. [Wu et al. \(2024\)](#) introduce Routing Experts (RoE) for multimodal LLMs, retrofitting trained models into a mixture-of-experts style by learning input-dependent shortcut routes through layers, guided by sparsity regularizers. [Jain et al. \(2024\)](#) present Mixture of Nested Experts (MoNE): experts organized in a hierarchy of increasing capacity, where tokens are sent to smaller experts when sufficient. MoNE learns to prioritize easy tokens through low-cost experts and reserve full models for hard cases, halving inference compute on ImageNet/Video tasks.

These methods exemplify sample-wise routing: at inference time, the model conditionally activates different sub-modules or experts for each input. Similarly, neural module networks ([Andreas et al., 2016](#)) assemble task-specific computation graphs from a library of modules. In modern LLMs/VLMs, these routing approaches – whether through gating experts, skipping layers, or assembling modules – form a spectrum of modular inference techniques that adapt the computation graph on a per-sample basis to balance cost and accuracy. Interestingly, our work suggests that transformer layers can function effectively as modules even without being specifically trained for that purpose.

B. Searching the Execution Program Space

This appendix provides full details of the execution-program search procedure used to test the conjecture in Section 2. The search is used purely as a diagnostic tool to study the existence and structure of valid execution programs, rather than as a practical inference-time method.

B.1. Execution Program Space

We follow the formalization in the main text and represent inference as the execution of a variable-length program that composes pretrained transformer layers. Consider a pretrained LLM with D transformer layers, where each layer defines a fixed computation function

$$f_i : \mathbb{R}^{T \times d} \rightarrow \mathbb{R}^{T \times d}, \quad i \in \{0, \dots, D-1\}.$$

An execution program is defined as a finite sequence of layer indices

$$\pi = (i_1, i_2, \dots, i_K), \quad i_k \in \{0, \dots, D-1\},$$

which induces the composed computation

$$F_\pi = f_{i_K} \circ \dots \circ f_{i_1}.$$

Executing a program applies this composition to the input representation and produces a prediction. A program is considered *valid* for a given input if it yields a correct prediction.

Programs may be shorter than the standard forward pass through layer skipping, or longer through layer repetition. Increasing program length corresponds to increasing the number of latent reasoning steps.

B.2. Search Space Constraints

The unconstrained space of programs grows exponentially with program length. To make search tractable while preserving expressiveness, we restrict the action space to structured operations on contiguous subsequences of layer indices. Specifically, we allow two classes of actions:

- **Skip:** remove a contiguous block of k indices from the program;
- **Repeat:** duplicate a contiguous block of k indices for r repetitions.

In all experiments, block size k and repetition count r are bounded by small constants ($k, r \leq 4$). These constraints significantly reduce the branching factor while retaining the ability to realize layer skipping, recurrence, and emergent reordering patterns.

B.3. Monte Carlo Tree Search Formulation

We formulate program discovery as a sequential decision process and employ Monte Carlo Tree Search (MCTS) to explore the constrained program space.

State and Actions. Each MCTS node corresponds to a partial or complete execution program π . Actions modify the current program by applying a valid skip or repeat operation, yielding a new program.

Reward. For a completed program π and input x with ground-truth answer y , we define a binary reward

$$r(\pi, x) = \mathbf{1}\{F_\pi(x) = y\},$$

where $F_\pi(x)$ denotes executing the composed computation induced by π .

Tree Policy. Tree traversal is guided by a UCB-style objective that balances exploitation, exploration, and program length regularization:

$$\text{UCB}(\pi) = \frac{R(\pi)}{v(\pi)} + c \sqrt{\frac{\ln V}{v(\pi)}} - \lambda \frac{|\pi|}{D},$$

where $R(\pi)$ is the cumulative reward, $v(\pi)$ is the visit count, V is the total number of simulations, and λ penalizes long programs to encourage efficiency.

Algorithm 1 Monte Carlo Tree Search for Execution Programs

Require: Input x , number of simulations N_{sim}

- 1: Initialize root program $\pi_0 = (0, 1, \dots, D - 1)$
- 2: **for** $n = 1$ to N_{sim} **do**
- 3: **Selection:** traverse tree using UCB to reach a leaf program
- 4: **Expansion:** generate child programs via valid skip/repeat actions
- 5: **Simulation:** execute $F_\pi(x)$ and compute reward $r(\pi, x)$
- 6: **Backpropagation:** update visit counts and cumulative rewards
- 7: **end for**
- 8: **return** explored programs and associated statistics

B.4. Search Algorithm

Algorithm 1 summarizes the MCTS procedure. We initialize the root node with the standard forward execution $\pi_0 = (0, 1, \dots, D - 1)$ and iteratively perform selection, expansion, simulation, and backpropagation. After a fixed number of simulations, we collect all explored programs with nonzero visit counts and analyze their validity and structural properties.

C. Experimental Results

This appendix provides additional experimental results that complement the main paper. We report full quantitative comparisons for both in-distribution and out-of-distribution evaluations across multiple pretrained LLMs. Unless otherwise stated, all models are evaluated in a fully frozen setting, and POLAR only predicts execution programs at inference time.

C.1. In-Distribution Performance

We first report detailed in-distribution results on DART-Math, a structured mathematical reasoning benchmark with five difficulty levels (DM-1 to DM-5). Tables 5, 6, and 7 present pass@k accuracy under different inference strategies for Qwen1.5-MoE-A2.7B-Chat, Qwen2.5-3B-Instruct, and Qwen3-8B, respectively.

Across all models and difficulty levels, POLAR achieves strong in-distribution performance and generally outperforms standard inference and prior dynamic-depth baselines. In particular, increasing $p@k$ leads to monotonic accuracy improvements for POLAR, demonstrating effective test-time computation scaling through execution-program exploration. At $p@5$, POLAR achieves substantial absolute gains over Base (sampling), with improvements of +15.6 / +17.5 / +15.0 / +10.1 / +12.2 on Qwen1.5-MoE-A2.7B-Chat, +40.4 / +19.2 / +17.5 / +14.8 / +13.5 on Qwen2.5-3B-Instruct, and +14.1 / +23.4 / +15.5 / +10.6 / +9.9 on Qwen3-8B for DM-1 to DM-5, respectively.

Notably, methods that rely solely on layer skipping (e.g., ShortGPT, MindSkip, FlexiDepth) often suffer accuracy degradation, especially on harder difficulty levels. In contrast, POLAR jointly supports layer skipping and recurrence, allowing it to retain or improve accuracy while exploring diverse execution programs. Compared to DR.LLM, which performs layer-level routing, POLAR achieves stronger performance in most settings, particularly at larger $p@k$, indicating the benefit of structured, program-level execution prediction.

C.2. Out-of-Distribution Generalization

We further evaluate the out-of-distribution (OOD) generalization of POLAR on benchmarks that differ substantially from DART-Math in both format and domain. Tables 8, 9, and 10 report pass@1 accuracy on ASDiv, MAWPS, and subject-wise subsets of MMLU-Pro using LLaMA-3.2-3B-Instruct, Qwen2.5-3B-Instruct, and Qwen3-8B, respectively.

Across all evaluated models, POLAR consistently improves over standard inference on arithmetic word problem benchmarks (ASDiv and MAWPS), indicating strong transfer from structured mathematical reasoning to natural language problem settings. On MMLU-Pro, which spans diverse domains including mathematics, natural sciences, social sciences, and humanities, POLAR achieves broad and consistent gains across most subject areas.

These results suggest that the execution programs learned by POLAR capture general, transferable computation control strategies rather than dataset-specific heuristics. Despite being trained on mathematical reasoning data, POLAR generalizes effectively to heterogeneous domains, highlighting the robustness of program-of-layers inference and its applicability beyond the original training distribution.

Table 5. Pass@k accuracy under different inference strategies applied to Qwen1.5-MoE-A2.7B-Chat.

Method	p@	DM-1	DM-2	DM-3	DM-4	DM-5
Qwen1.5-MoE-A2.7B-Chat						
Base ($\tau=0$)	1	48.2	25.5	24.1	16.7	10.9
Base (sampling)	1	41.8	24.9	21.3	18.2	12.4
	2	46.8	28.2	23.3	20.3	14.2
	3	49.6	30.0	25.8	21.6	14.7
	4	50.4	31.8	26.8	22.7	15.5
	5	50.4	33.5	27.3	22.7	16.5
ShortGPT	1	22.7	12.8	8.6	8.3	3.0
	2	45.4	23.7	15.4	13.5	7.1
	3	54.6	30.9	18.2	16.9	10.4
	4	56.0	33.2	21.3	17.2	10.4
	5	59.6	35.0	23.0	18.2	11.2
MindSkip	1	22.0	12.2	8.9	8.1	2.8
	2	36.9	22.0	13.4	12.8	5.1
	3	48.9	27.6	17.2	15.6	6.9
	4	51.1	30.9	19.7	17.4	8.6
	5	54.6	31.5	22.0	20.3	10.4
FlexiDepth	1	15.6	7.7	5.6	3.1	2.8
	2	27.0	13.4	7.8	6.2	4.6
	3	44.0	19.9	16.2	12.2	6.9
	4	44.7	21.7	18.7	13.5	6.9
	5	47.5	23.7	19.0	14.6	6.9
DR.LLM	1	35.5	23.4	20.3	16.9	11.4
	2	54.6	26.7	25.1	23.2	18.8
	3	56.0	33.2	27.8	24.5	19.3
	4	56.7	37.1	31.9	24.5	20.1
	5	63.1	40.9	34.7	28.6	23.9
POLAR	1	53.9	36.8	28.4	21.1	20.3
	2	58.2	39.8	32.2	25.3	22.8
	3	61.7	43.0	34.9	27.9	25.1
	4	64.5	47.2	40.0	29.4	27.9
	5	66.0	51.0	42.3	32.8	28.7
Δ vs. Base (sampling)	5	+15.6	+17.5	+15.0	+10.1	+12.2

Table 6. Pass@k accuracy under different inference strategies applied to Qwen2.5-3B-Instruct.

Method	p@	DM-1	DM-2	DM-3	DM-4	DM-5
Qwen2.5-3B-Instruct						
Base ($\tau=0$)	1	17.7	7.1	3.8	3.4	1.5
Base (sampling)	1	17.7	9.2	4.3	2.3	1.5
	2	22.0	10.7	5.6	3.6	2.3
	3	24.8	12.5	6.3	3.9	2.3
	4	27.0	13.1	6.3	4.4	2.5
	5	28.4	13.4	6.8	5.5	2.5
ShortGPT	1	12.1	3.0	1.5	1.3	0.3
	2	35.5	10.7	5.6	2.3	2.3
	3	35.5	12.5	7.3	5.2	3.0
	4	41.1	13.9	8.6	8.6	5.8
	5	53.9	21.7	13.9	12.8	9.1
MindSkip	1	12.8	3.9	1.5	1.3	1.3
	2	24.8	10.4	5.8	5.2	3.6
	3	35.5	14.2	9.1	7.6	4.3
	4	39.0	18.4	12.4	10.4	6.3
	5	46.1	26.1	16.5	13.5	9.1
FlexiDepth	1	0.0	0.0	0.0	0.0	0.0
	2	0.7	0.3	0.3	0.0	0.5
	3	7.8	2.4	0.3	1.8	0.5
	4	21.3	6.8	3.5	5.2	2.5
	5	25.5	8.3	4.3	5.5	2.5
DR.LLM	1	1.4	6.5	3.5	2.6	0.8
	2	4.3	17.5	12.7	6.2	1.8
	3	7.1	22.3	15.2	7.3	2.5
	4	8.5	23.7	15.7	9.6	2.8
	5	9.9	27.6	18.7	11.5	3.8
POlar	1	46.1	19.9	14.9	12.0	7.1
	2	57.4	25.5	19.0	15.4	11.4
	3	63.1	28.2	21.3	17.7	13.2
	4	66.0	31.5	22.5	18.2	15.2
	5	68.8	32.6	24.3	20.3	16.0
Δ vs. Base (sampling)	5	+40.4	+19.2	+17.5	+14.8	+13.5

Table 7. Pass@k accuracy under different inference strategies applied to Qwen3-8B.

Method	p@	DM-1	DM-2	DM-3	DM-4	DM-5
Qwen3-8B						
Base ($\tau=0$)	1	57.4	30.6	17.5	16.7	11.7
Base (sampling)	1	56.7	30.0	19.0	16.9	11.9
	2	62.4	34.1	21.5	19.3	12.4
	3	64.5	35.6	22.3	19.5	13.5
	4	66.0	35.9	22.5	19.8	14.5
	5	66.0	37.1	23.0	20.1	15.2
ShortGPT	1	19.1	7.4	4.6	5.2	3.3
	2	39.0	11.6	5.6	7.3	5.1
	3	51.1	15.1	8.4	8.9	6.3
	4	57.4	22.0	12.2	12.5	7.4
	5	60.3	23.7	14.4	13.5	10.2
MindSkip	1	47.5	19.0	10.6	12.5	7.1
	2	54.6	22.8	14.4	15.6	8.6
	3	60.3	28.8	17.7	17.4	10.2
	4	67.4	31.5	19.2	19.0	11.2
	5	68.8	34.7	20.0	19.8	13.2
FlexiDepth	1	1.4	0.3	0.3	0.3	0.3
	2	27.7	9.2	5.1	3.1	2.0
	3	34.8	13.1	8.4	6.5	2.8
	4	44.0	18.1	11.4	10.7	4.1
	5	55.3	23.4	13.2	14.1	6.3
DR.LLM	1	5.0	26.1	15.7	6.8	1.3
	2	12.8	27.3	19.5	8.6	2.0
	3	17.7	40.9	25.8	10.9	3.0
	4	22.0	49.6	30.9	14.1	4.1
	5	26.2	53.4	32.4	15.4	5.1
PoLAR	1	61.0	42.7	21.3	21.4	16.0
	2	69.5	47.8	27.6	26.6	19.5
	3	74.5	54.9	33.2	28.6	21.6
	4	78.0	57.6	35.7	29.4	23.4
	5	80.1	60.5	38.5	30.7	25.1
Δ vs. Base (sampling)	5	+14.1	+23.4	+15.5	+10.6	+9.9

Table 8. Out-of-distribution (OOD) performance at pass@1. We report accuracy on OOD benchmarks using LLaMA-3.2-3B-Instruct.

Method	ASDiv	MAWPS	MMLU-Pro												
			Math	Phys	Chem	Law	Eng	Other	Econ	Health	Psych	Bus	Bio	Phil	Hist
Base ($\tau=0$)	78.4	71.5	19.5	19.0	18.6	17.1	19.8	22.4	36.1	28.5	33.0	22.2	44.9	23.8	28.3
ShortGPT	2.3	4.6	29.9	26.9	26.9	23.3	20.1	33.0	36.7	35.3	38.2	28.6	36.0	41.7	40.9
MindSkip	9.0	7.5	40.4	47.7	43.3	31.8	43.7	38.5	47.6	47.2	51.5	31.7	52.3	49.7	55.6
FlexiDepth	4.7	1.3	7.8	5.9	4.8	6.4	6.3	4.7	5.9	5.9	3.6	6.0	4.9	7.4	4.7
DR.LLM	75.4	61.3	13.6	14.4	16.2	10.0	16.4	9.0	13.6	10.6	15.2	13.0	19.8	8.2	23.1
PoLAR	81.4	73.7	40.1	39.0	41.0	34.3	43.2	42.9	50.1	52.4	52.4	32.7	56.6	44.5	48.8

Table 9. Out-of-distribution (OOD) performance at pass@1. We report accuracy on OOD benchmarks using Qwen2.5-3B-Instruct.

Method	ASDiv	MAWPS	MMLU-Pro												
			Math	Phys	Chem	Law	Eng	Other	Econ	Health	Psych	Bus	Bio	Phil	Hist
Base ($\tau=0$)	49.5	36.2	26.3	28.7	27.7	24.4	37.0	33.9	47.5	40.3	52.6	31.3	62.2	33.7	34.6
ShortGPT	1.3	1.2	8.9	9.3	10.0	12.1	12.1	12.8	12.6	13.4	8.9	11.5	10.3	9.8	12.1
MindSkip	8.0	3.5	7.6	7.5	8.3	6.0	9.5	7.4	8.6	9.8	7.5	8.5	7.5	9.4	6.8
FlexiDepth	2.7	0.6	9.8	9.3	10.5	10.4	11.4	11.5	10.8	11.5	9.9	8.2	10.6	11.4	11.0
DR.LLM	0.0	0.0	4.0	4.8	2.8	3.2	3.0	3.2	2.8	2.6	1.4	4.6	5.8	3.0	4.2
POLaR	78.1	57.7	32.1	35.6	33.0	27.6	41.5	38.9	53.0	46.5	57.4	33.4	66.0	38.9	35.4

Table 10. Out-of-distribution (OOD) performance at pass@1. We report accuracy on OOD benchmarks using Qwen3-8B.

Method	ASDiv	MAWPS	MMLU-Pro												
			Math	Phys	Chem	Law	Eng	Other	Econ	Health	Psych	Bus	Bio	Phil	Hist
Base ($\tau=0$)	67.1	52.1	21.4	24.8	19.2	32.6	23.4	43.0	60.4	56.2	62.8	26.6	72.2	42.3	50.1
ShortGPT	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
MindSkip	20.9	13.1	26.1	33.9	31.5	58.0	32.1	67.0	64.7	68.0	71.9	36.6	68.8	68.5	65.9
FlexiDepth	1.0	0.4	0.9	1.0	0.6	0.6	0.5	0.8	0.9	1.0	0.1	0.8	1.0	0.6	1.3
DR.LLM	69.4	0.0	0.6	0.2	1.8	1.8	0.8	0.6	1.2	1.4	2.0	1.2	0.8	0.6	0.8
POLaR	72.8	66.9	26.8	34.6	23.4	48.7	35.8	57.1	65.2	67.4	72.4	34.1	75.4	56.7	62.0

D. Empirical Details

D.1. Dataset Details

All in-distribution experiments are conducted on **DART-Math**, a structured mathematical reasoning benchmark consisting of five difficulty levels, denoted as **DM-1** to **DM-5**.

We construct the in-distribution data from the DART-Math MATH pool. Since the released pool may contain multiple response records associated with the same underlying question, we deduplicate examples within each difficulty level by question text before splitting. We then adopt a difficulty-wise data split, where each difficulty level is split independently into training, validation, and test sets.

After deduplication, the five difficulty levels contain 565, 1,349, 1,579, 1,537, and 1,577 examples for DM-1 to DM-5, respectively. For each difficulty level, we use approximately 62.5% of the examples for training, 12.5% for validation, and 25% for testing.

Overall, this results in 4,130 training examples, 826 validation examples, and 1,651 test examples. All in-distribution results are reported on the held-out test sets corresponding to the same difficulty level used for training.

Revision note. This arXiv version updates the DART-Math in-distribution evaluation protocol by removing duplicate questions prior to data splitting. We have rerun the affected in-distribution analyses, and the qualitative conclusions remain unchanged. The out-of-distribution (OOD) results are unaffected.

D.2. Training Configuration

We train the POLAR prediction network using supervised learning on the training splits described above. All hyperparameters are selected via validation tuning.

Optimization. We use the AdamW optimizer and tune the learning rate from $\{1e-4, 3e-4, 5e-4, 8e-4, 1e-3, 3e-3\}$, the batch size from $\{32, 128, 256\}$, and the number of training epochs from $\{3, 10\}$ based on validation performance. We adopt a cosine learning rate schedule with linear warmup, and tune the number of warmup steps on the validation split together with the other hyperparameters.

D.3. Predictor Size

The learned POLAR predictor is lightweight compared with the frozen base LLM. Across the evaluated models, the predictor contains approximately 2.1M parameters, corresponding to only 0.01%–0.06% of the base model size. This small footprint makes training and inference inexpensive relative to standard LLM fine-tuning or full-model execution.

Table 11. Parameter size of the learned POLAR predictor compared with each frozen base LLM.

Model	Base LLM params	Predictor params	Predictor / Base
Qwen1.5-MoE-A2.7B-Chat	14.32B	2.11M	0.0148%
Qwen2.5-3B-Instruct	3.40B	2.12M	0.0623%
Qwen3-8B	8.19B	2.12M	0.0258%
LLaMA-3.2-3B-Instruct	3.61B	2.11M	0.0586%

D.4. Direct Prompting

We adopt a direct prompting strategy throughout all experiments, without eliciting chain-of-thought or intermediate reasoning. The model is explicitly instructed to output only the final answer in a strictly formatted form.

Given a math problem instance `question`, the input prompt is constructed as follows:

```
Solve the following math problem and output ONLY the final answer directly,
formatted strictly as \boxed{ANSWER}.
### Problem Start
{question}
```

Problem End

Answer:

This prompt design enforces concise answer generation and isolates the effect of latent execution programs from token-level reasoning strategies.