POFORMER: RECURRENT POINTER ADDRESSING FOR LENGTH GENERALIZATION

Anonymous authors
Paper under double-blind review

ABSTRACT

Length generalization requires a model trained on short sequences to apply the same algorithm to longer inputs, enabling compact, reusable computational subcircuits rather than length-specific behaviors. Looped Transformers provide recurrent computation but do not explicitly retain previously attended locations, making precise traversal difficult when repeated symbols render content matching ambiguous or when the next address depends on the current state or retrieved content. We introduce POFORMER, a recurrent Transformer whose recurrent state is factorized into a controller state that carries intermediate computation and a pointer state that tracks the active memory location. Selected attention heads act as pointer heads that maintain and update active addresses over the input, enabling local address shifts and global content-conditioned jumps while preserving content-based retrieval. We formalize this mechanism with P-RASP, a recurrent pointer extension of RASP, and constructively show that POFORMER can simulate P-RASP programs over finite execution horizons. Across controlled algorithmic tasks, POFORMER substantially improves OOD length generalization over standard and looped Transformer baselines. Its interpretable pointer trajectories reveal stable traversal strategies and expose address-level failures such as pointer drift and positional aliasing. Together, these results establish explicit recurrent address state as an effective inductive bias for length generalization in tasks requiring precise, stateful addressing.

1 INTRODUCTION

Length generalization (LG) is the ability of a model trained on short sequences to apply the same computation to substantially longer inputs. It tests whether a model has learned an executable procedure rather than a length-specific statistical pattern. Controlled algorithmic tasks provide a useful testbed for length generalization (Kozachinskiy et al., 2026; Kim & Suzuki, 2025; Jelassi et al., 2024; Zhou et al., 2024b; Fan et al., 2025). Their solutions typically require repeatedly applying a length-independent transition rule while precisely traversing the input and maintaining intermediate state. Because the target computation may depend on every input position, even a single addressing or state-update error can invalidate the entire output.

Existing approaches improve LG by increasing iterative computation. Chain-of-thought and scratchpad methods (Huang et al., 2025; Bavandpour et al., 2025; Kim & Suzuki, 2025; Li et al., 2024b) externalize intermediate states, while looped and latent recurrent Transformers (Geiping et al., 2025; Jeddi et al., 2026; Hao et al., 2025; Merrill & Sabharwal, 2025; Yu et al., 2026) repeatedly apply shared blocks. However, prior work suggests that many length-generalization failures arise not from learning an incorrect local state-update rule, but from failures of attention-based addressing (Ebrahimi et al., 2026; Duan & Shi, 2024). When evaluated on sequences beyond the training-length range, models often fail to preserve extrapolatable attention patterns, causing subsequent computation to operate on incorrect token positions. Therefore, additional depth does not guarantee that each computation step reads the correct token position. Standard and looped Transformers propagate computational state through hidden representations but do not explicitly retain the previously attended address. RASP-based analyses (Zhou et al., 2024a; Fan et al., 2025; Yang & Chiang, 2024; Huang et al., 2025) further suggest that length-generalizable Transformer computations favor local or translation-invariant positional operations. Such operations do not naturally provide a recurrent address that can be updated according to the controller state, retrieved content, or previously visited location. This limitation is particularly evident with repeated symbols, where content matching

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

cannot identify the relevant occurrence (Jelassi et al., 2024; Zhou et al., 2024a; Huang et al., 2025; Fan et al., 2025). Thus, a model may learn the correct local transition yet fail on longer inputs because its implicit address trajectory drifts, skips positions, or attends to an aliased location.

To address this limitation, we introduce POFORMER, which leverages the computational-state-carrying capability of looped Transformers and introduces an explicit addressing channel as a new inductive bias. PoFormer separates recurrent computation from memory addressing: a recurrent controller maintains the intermediate computational state, while the addressing channel tracks where the model is operating in the input memory. This design enables adaptive and directly observable traversal policies by separating *what* information is processed from *where* the computation is performed.

Building on this idea, we make the following contributions:

- We propose POFORMER, a recurrent Transformer architecture that augments iterative computation with explicit active pointers over static memory, addressing the bottleneck of recurrent address control in length generalization.
- We introduce P-RASP, a formal framework that characterizes pointer shifts, content-conditioned jumps, and finite-state controller updates, and show how these operations can be realized by POFORMER.
- We empirically show that POFORMER improves out-of-distribution length generalization across multiple controlled algorithmic tasks, while its pointer trajectories expose address-level failure modes.

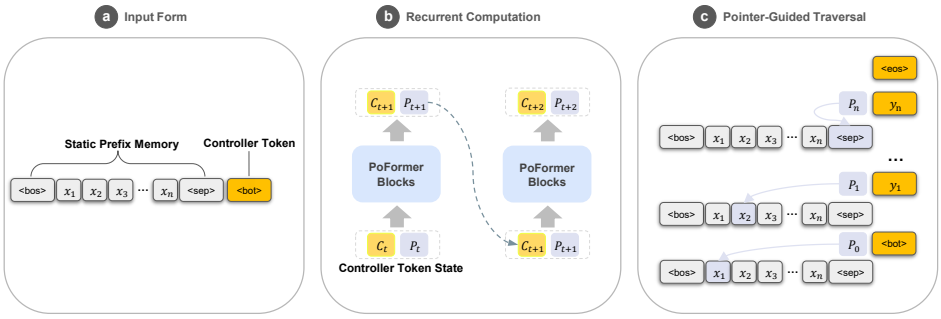

Figure 1: Overview of PoFormer. (a) The input prefix is stored as static memory, while the controller token conducts recurrent computation. (b) At each loop step, the PoFormer block updates the recurrent controller state c_t and pointer state p_t, producing (c_{t+1}, p_{t+1}) for the next step. (c) The pointer state guides traversal over the static memory: successive pointers attend to different memory positions and emit one output token per recurrent step, terminating when the model predicts <eos>.

2 BACKGROUND

Positional Representations in Transformers Transformers use a range of positional schemes to encode sequence order: NoPE (Haviv et al., 2022) relies on positional information emerging implicitly from causal structure, APE (Devlin et al., 2019) assigns position-specific representations, and relative methods (Raffel et al., 2020; Press et al., 2022; Li et al., 2024a) encode pairwise distances or biases. RoPE (Su et al., 2024) instead rotates queries and keys according to their positions, providing a flexible relative-position geometry within dot-product attention. While these schemes differ in their extrapolation properties, they represent static positional relationships rather than a recurrent state that records and updates the location accessed during computation.

Formal Analysis with RASP RASP abstracts Transformer computation through elementwise operations, attention-like selectors, and aggregation (Weiss et al., 2021). Its variants study different

aspects of length-generalizable computation: RASP-L restricts the available primitives to operations that remain stable across sequence lengths (Zhou et al., 2024a); subsequent frameworks formalize it in terms of counting-based sequence computation (Yang & Chiang, 2024; Huang et al., 2025), and extend RASP-L with recurrent execution (Fan et al., 2025). Collectively, these analyses show how Transformers can reuse local or translation-invariant attention rules across lengths. However, when a task requires precise position-based addressing, such rules do not by themselves provide a persistent state that records and updates the previously accessed location, making length generalization difficult on traversal-based problems.

Looped Transformers and Latent Computation Looped Transformers repeatedly apply a shared computational block, allowing effective depth to grow with the number of recurrent iterations without increasing the number of parameters (Dehghani et al., 2019; Geiping et al., 2025; Jeddi et al., 2026; Saunshi et al., 2025). Their recurrent hidden states carry intermediate results from one iteration to the next, enabling the model to iteratively refine a solution and implement state-dependent computation (Hao et al., 2025; Amos et al., 2026; Wang et al., 2026; Xu & Sato, 2026). From this perspective, a looped Transformer can implement finite-state computations by encoding the current state in its recurrent hidden representations and using the shared Transformer block as a learned transition function.

Length Generalization Existing work improves length generalization by modifying the addressing mechanism, training procedure, or computational structure. Positional-anchor methods introduce landmarks or distinctive tokens that convert positional addressing into content-based retrieval, which standard attention performs more naturally (Zhou et al., 2024a; Ebrahimi et al., 2024; Zhou et al., 2024b; Cho et al., 2025; Golowich et al., 2025). However, they intrusively modify the input representation, and the number of uniquely identifiable anchors is ultimately limited by the vocabulary. Another line of work injects task-specific biases or masks at the positions that the model is expected to attend to (Csordás et al., 2022; Duan & Shi, 2024; Zhou et al., 2024b). Such methods can provide strong extrapolation when the required attention pattern is known, but their manually specified inductive bias is less flexible when traversal depends on the current state or input content. Auxiliary-task approaches first train on a simpler or less expensive task whose attention pattern resembles that of the target task, often extending the auxiliary training to longer lengths before transferring to the target task (Awasthi et al., 2025; Cai et al., 2025). Their effectiveness depends on identifying and designing an appropriate auxiliary task, which may not be possible for every target problem. RASP-inspired looped approaches repeatedly apply length-generalizable local rules, effectively extending RASP-L computation toward an n-RASP-L iterative program (Fan et al., 2025; Merrill & Sabharwal, 2025). However, these approaches may require preallocated padding or output positions to store intermediate computation, making their execution closer to an encoder-style architecture than standard next-token prediction paradigm. They may also require an additional termination mechanism.

3 METHODOLOGY

3.1 OVERVIEW

A standard looped Transformer repeatedly applies a shared Transformer module F_θ,

$$X^{t+1} = F_\theta(X^t), \tag{1}$$

where t denotes the recurrent step. Parameter sharing allows the effective computational depth to grow with the number of iterations, while recurrent hidden states propagate intermediate computational results. However, the memory location accessed at each step remains implicit in the hidden representations and attention patterns.

PoFormer augments this recurrent computation with an explicit address state. It separates the input into a static prefix memory and a recurrent controller, and equips selected attention heads with active pointers. The controller propagates the intermediate computational state, while each pointer records the memory location currently addressed by its head. Consequently, the recurrent state of PoFormer consists of both a content state and an address state.

Given an input

$$x = (x_1, \ldots, x_L, \texttt{<bot>}), \tag{2}$$

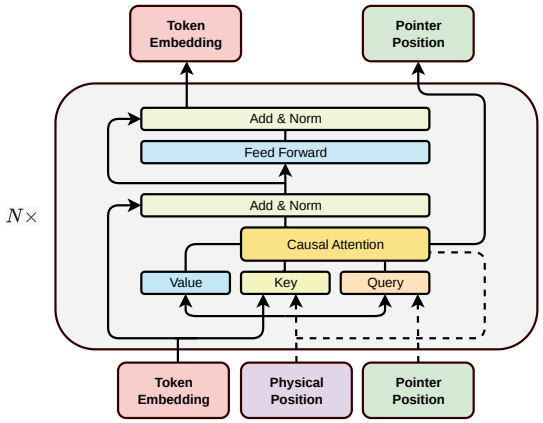

Figure 2: Pointer attention in a PoFormer layer. Queries are rotated by the active pointer position, while keys are rotated by physical token positions and values remain content-only. The resulting attention distribution is used both to read content and to update the pointer position.

PoFormer first encodes the prefix tokens into a static memory

$$M = (M_1, \ldots, M_L), \qquad M_j \in \mathbb{R}^d. \tag{3}$$

The prefix representations are computed once and remain fixed during recurrent execution. The final token <bot> initializes a controller state $c^{0,0}$, which is updated across layers and loop steps while attending to the static memory (Figure 1).

3.2 Pointer Attention

PoFormer builds its addressing mechanism on RoPE (Su et al., 2024). Standard RoPE satisfies

$$(R_{r_i} q_i)^\top (R_{r_j} k_j) = q_i^\top R_{r_j - r_i} k_j \tag{4}$$

where R_r is the rotation associated with position r. PoFormer replaces the query's physical position r_i with a recurrent active pointer $p_{i,h}^{t,\ell} \in \mathbb{R}$ maintained by pointer head h. The pointer is initially set to the physical position of the corresponding token, i.e., $p_{i,h}^{0,0} = r_i$. For query position i and key–value position j, the projections are

$$q_{i,h}^{t,\ell} = W_{\ell,h}^Q X_i^{t,\ell}, \qquad k_{j,h}^{t,\ell} = W_{\ell,h}^K X_j^{t,\ell}, \qquad v_{j,h}^{t,\ell} = W_{\ell,h}^V X_j^{t,\ell} \tag{5}$$

The query is rotated by its active pointer, while the key is rotated by its physical position:

$$\widetilde{q}_{i,h}^{t,\ell} = R_{p_{i,h}^{t,\ell}} q_{i,h}^{t,\ell}, \qquad \widetilde{k}_{j,h}^{t,\ell} = R_{r_j} k_{j,h}^{t,\ell} \tag{6}$$

Values remain content-only. The resulting attention score is

$$s_{i,j,h}^{t,\ell} = \frac{\left(\widetilde{q}_{i,h}^{t,\ell}\right)^\top \widetilde{k}_{j,h}^{t,\ell}}{\sqrt{d_h}} = \frac{\left(q_{i,h}^{t,\ell}\right)^\top R_{r_j - p_{i,h}^{t,\ell}} k_{j,h}^{t,\ell}}{\sqrt{d_h}} \tag{7}$$

Thus, positional interaction is measured relative to the active pointer rather than the query token's fixed physical position. Let $\mathcal{A}(i)$ be the positions visible to query i under the causal mask. Given temperature $\tau > 0$, the attention distribution is

$$a_{i,j,h}^{t,\ell} = \frac{\exp(s_{i,j,h}^{t,\ell}/\tau)}{\sum_{m \in \mathcal{A}(i)} \exp(s_{i,m,h}^{t,\ell}/\tau)}, \qquad j \in \mathcal{A}(i). \tag{8}$$

During training, the head uses this distribution both to retrieve content and to update its active pointer through the expected physical position:

$$z_{i,h}^{t,\ell} = \sum_{j \in \mathcal{A}(i)} a_{i,j,h}^{t,\ell} v_{j,h}^{t,\ell}, \qquad p_{i,h}^{t,\ell+1} = \sum_{j \in \mathcal{A}(i)} a_{i,j,h}^{t,\ell} r_j. \tag{9}$$

This expected-position update provides a differentiable relaxation of discrete address selection. During inference, we use a lower temperature $\tau < 1$ and execute the corresponding discrete pointer transition:

$$p_{i,h}^{t,\ell+1} = r_{j^\star}, \qquad j^\star = \arg\max_{j \in \mathcal{A}(i)} a_{i,j,h}^{t,\ell}. \tag{10}$$

For the recurrent controller, the pointer produced by the final layer is carried into the next loop step:

$$p_{L+1,h}^{t+1,0} = p_{L+1,h}^{t,N}. \tag{11}$$

In contrast, pointers for prefix tokens ($i \le L$) update only during the initial forward pass ($t = 0$). For $t > 0$, their representations are fixed as static memory, i.e., $z_{i,h}^{t+1,\ell} = z_{i,h}^{t,\ell}$. Consequently, the attention distribution determines both *what* content is retrieved and *where* the head will attend in the next layer or recurrent step.

3.3 Pointer, Slave, and Normal Heads

PoFormer supports three types of attention heads:

- **Pointer heads** maintain independent active pointers and update them using Equation 9. They learn the address transitions that define the model's traversal policy.
- **Slave heads** follow the pointer of a designated pointer head. If $\pi(h)$ is the master pointer assigned to slave head h, then $p_h^{t,\ell} = p_{\pi(h)}^{t,\ell}$. Slave heads retain independent query, key, and value projections, allowing feature-specific readouts from the same addressed location.
- **Normal heads** use standard RoPE with physical query and key positions. They preserve ordinary Transformer attention for operations that do not require recurrent addressing, such as instruction recognition, boundary detection, and global content retrieval.

This division separates traversal control from feature extraction. Pointer heads determine where the model reads, slave heads extract complementary information along the same trajectory, and normal heads retain conventional content-based attention.

3.4 Output Decoding and Training

For a fixed integer $k \ge 1$, PoFormer decodes one output token after every k loop steps; we use $k = 1$ unless otherwise specified. For output step m, define $z^m := c^{mk,N}$, where $c^{mk,N}$ is the controller state after loop step mk and the final layer N of the recurrent module. The corresponding output distribution is

$$P_\theta(y_m \mid M, z^m) = \mathrm{softmax}(W_U z^m). \tag{12}$$

The decoded token is used only as an output readout: it is not appended to the input sequence and is not fed back into subsequent recurrent computation. Thus, all output tokens are conditioned on the same static input x, while their predictions differ through the evolving latent controller states $z^1, z^2, \ldots$. The model terminates when it predicts `<eos>` at a decoding step or reaches a predefined maximum number of recurrent iterations. Given a ground-truth output sequence

$$y^* = (y_1^*, \ldots, y_T^*), \tag{13}$$

the model is unrolled for kT loop steps and trained by minimizing the sum of cross-entropy losses at the decoding checkpoints:

$$\mathcal{L}_{\mathrm{CE}} = -\sum_{m=1}^{T} \log P_\theta(y_m^* \mid M, z^m) = -\sum_{t \in \{k, 2k, \ldots, kT\}} \log P_\theta\left(y_{t/k}^* \mid M, z^{t/k}\right) \tag{14}$$

Because each z^m depends on all preceding recurrent updates, the loss at a later output step backpropagates through the earlier controller and pointer trajectories. Unless otherwise stated, only the output sequence is supervised; the intermediate controller states and pointer trajectories remain latent.

4 P-RASP AND THE EXPRESSIVITY OF POFORMER

4.1 P-RASP

RASP-L and C-RASP characterize length-stable sequence computations expressible by standard Transformers under their respective restrictions (Zhou et al., 2024a; Yang & Chiang, 2024; Fan et al., 2025). Building on these formalisms, we introduce *Pointer-RASP* (P-RASP), a RASP-style language for describing recurrent computation with explicit address states over a static memory. P-RASP retains the elementwise, selection, and aggregation operations of existing RASP variants while maintaining a finite controller state and a collection of pointers that persist across recurrent iterations. By extending length-stable RASP-style computation with recurrent address state, P-RASP provides a symbolic framework for expressing algorithms whose operations can be reused across input lengths, supporting length generalization.

The central pointer primitives are $\mathrm{p_shift}$ and $\mathrm{p_jump}$. The $\mathrm{p_shift}$ primitive updates a pointer by a constant relative offset independent of its current position, enabling translation-invariant traversal operations such as moving forward, moving backward, or remaining at the current memory location. The $\mathrm{p_jump}$ primitive instead updates a pointer to the position of a memory item selected by a content-dependent predicate, allowing non-local transitions whose target depends on the current state and stored content. In addition, P-RASP provides primitives for reading the value at a pointer, updating the finite controller state from retrieved values, and emitting output symbols. Formal definitions of these primitives and their operational semantics are given in Appendix B.

Definition 4.1 (P-RASP program)**.** *A* P-RASP *program is a recurrent sequence computation of the form*

$$P = P_{\mathrm{post}} \circ \left(P_{\mathrm{loop}}\right)^{T(L)} \circ P_{\mathrm{pre}}, \tag{15}$$

where P_{pre} initializes a static memory $M = (M_1, \ldots, M_L)$, a finite controller state s, and a collection of pointers $\mathbf{p} = (p_1, \ldots, p_H)$. The recurrent program P_{loop} applies a fixed composition of P-RASP *primitives to update s and $\mathbf{p}$ while leaving M unchanged, and P_{post} decodes the program output. The same recurrent program is applied for $T(L)$ iterations, where $T(L)$ may depend on the input length.*

4.2 EXPRESSIVITY OF POFORMER

We next relate PoFormer to the primitives of P-RASP. The key constructions show that pointer attention can implement relative address transitions and content-conditioned jumps. Standard attention readout and a sufficiently expressive feed-forward module additionally provide memory reads, finite-state controller updates, and output emission. We assume a finite address range in which the required RoPE displacements are distinguishable, unique attention targets with positive margins, and sufficiently sharp attention.

Proposition 4.2 (Relative pointer shift)**.** *For any fixed offset $\Delta \in \mathbb{Z}$, a PoFormer pointer head can approximate the* P-RASP *operation*

$$p^{t+1} = \mathrm{p_shift}(p^t, \Delta) = p^t + \Delta \tag{16}$$

arbitrarily accurately, provided that the resulting address is valid and the required relative displacements are distinguishable within the considered RoPE range.

Proposition 4.3 (Content-conditioned pointer jump)**.** *Suppose a controller-dependent content predicate identifies a unique memory cell with a positive score margin. If the positional distortion induced by RoPE is smaller than this margin over the considered displacement range, then a PoFormer pointer head can approximate the* P-RASP *operation*

$$p^{t+1} = \mathrm{p_jump}(M, \phi_{s^t}) \tag{17}$$

arbitrarily accurately by updating its active pointer to the physical address of the selected memory cell.

Theorem 4.4 (Finite-horizon simulation of P-RASP)**.** *Consider a finite-state* P-RASP *program whose pointer operations remain within a finite RoPE-distinguishable range and whose reads and jumps have unique targets with positive margins. For every finite execution horizon T and every $\epsilon > 0$, there exists a sufficiently wide and deep PoFormer that simulates the encoded controller states, pointer trajectories, and output emissions of the program up to time T with error at most ϵ.*

Corollary 4.5 (Finite-state traversal over static memory)**.** *PoFormer can simulate any finite-state algorithm that operates over static memory through pointer-based reads, a finite set of relative shifts, uniquely resolved content-conditioned jumps, and state-dependent updates and emissions. An algorithm requiring $T(L)$ memory accesses can be realized using $O(T(L))$ recurrent computation. This class includes sequential, reverse, and content-linked traversal as well as finite-state accumulation; representative examples are Duplicate-Token Copy, Prefix Parity, and Shuffled Linked-List Traversal, each admitting an $O(L)$-step* P-RASP *program.*

The constructive proofs of Propositions 4.2–4.3 and Theorem 4.4, together with the auxiliary pointer-read and controller-update results, are given in Appendix C. Example P-RASP programs establishing Corollary 4.5 are provided in Appendix D.

5 Experiments

We evaluate whether the addressing mechanisms characterized in Section 4.2 can be learned from supervision and remain stable beyond the training horizon. Our tasks require repeated access to a static input memory and isolate complementary aspects of address control: relative traversal, finite-state accumulation, repeated-symbol disambiguation, and content-conditioned jumps. We first study six basic tasks using output supervision alone, and then consider two advanced tasks whose more underconstrained traversal policies additionally benefit from auxiliary pointer supervision. Full data-generation procedures, model configurations, training hyperparameters, and additional results are provided in Appendix E.

5.1 Tasks and Experimental Setup

Tasks We evaluate six basic algorithmic tasks that isolate different forms of recurrent address control. **Duplicate-Token Copy** and **Reverse** require constant pointer shifts in opposite directions; **Parity** and **Flip-Flop** combine sequential traversal with finite-state updates; **Unique Set** requires detecting repeated symbols while preserving their input order; and **Shuffled Linked List** requires content-conditioned jumps between non-adjacent memory records. We further consider two advanced tasks that require more complex pointer logic. **Bidirectional Copy** requires two opposing traversal policies to be compressed into the same pointer head; **Binary Addition** coordinates two pointer trajectories with a recurrent carry state. Inputs may contain repeated symbols, preventing the required positional traversal from being replaced by token-identity matching. Formal input–output definitions and data-generation configurations are given in Appendix E.1.

Baselines We compare PoFormer against decoder-only NTP Transformers using NoPE, APE, RoPE (Su et al., 2024), and ALiBi (Press et al., 2022), as well as a weight-shared RoPE Looped Transformer without explicit pointer states. Models are matched in size whenever their architectures permit. Complete architectural definitions and hyperparameters are provided in Appendix E.2.

Training and Evaluation All models use the same task examples and output-token supervision. We adopt a two-stage curriculum (Garg et al., 2022; Pouransari et al., 2024): models first learn the task on short instances, after which the maximum sampled length is progressively increased to the training limit. Length schedules, vocabulary sizes, repetition rates, and optimization settings are reported in Appendix E.2.1. We use sequence-level exact-match accuracy to evaluate the length generalization results.

5.2 Results on Basic Tasks

As shown in Figure 3, PoFormer achieves strong length generalization under a compact model configuration. For a controlled architectural comparison, all baselines use the same backbone scale as PoFormer (Table 3), although several fail to fully learn the ID tasks at this scale. To control for this underfitting, we additionally evaluate enlarged baselines with strong ID performance; they nevertheless exhibit similar degradation beyond the training-length range (Appendix E.3). These results indicate that the OOD gap cannot be explained solely by limited model capacity or incomplete ID learning, supporting the role of explicit recurrent addressing in PoFormer's length generalization.

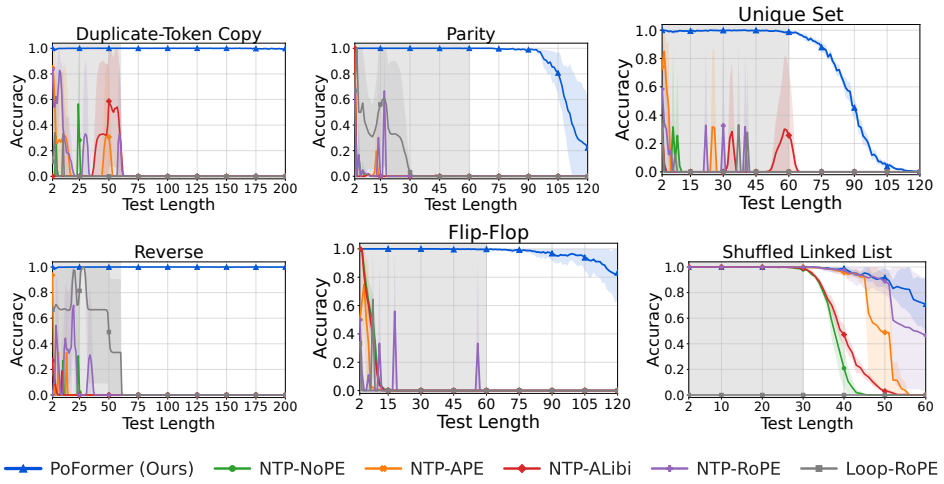

Figure 3: Length generalization on six basic tasks. Curves show mean EM accuracy ± SD over three seeds; the background shading marks the training-length range.

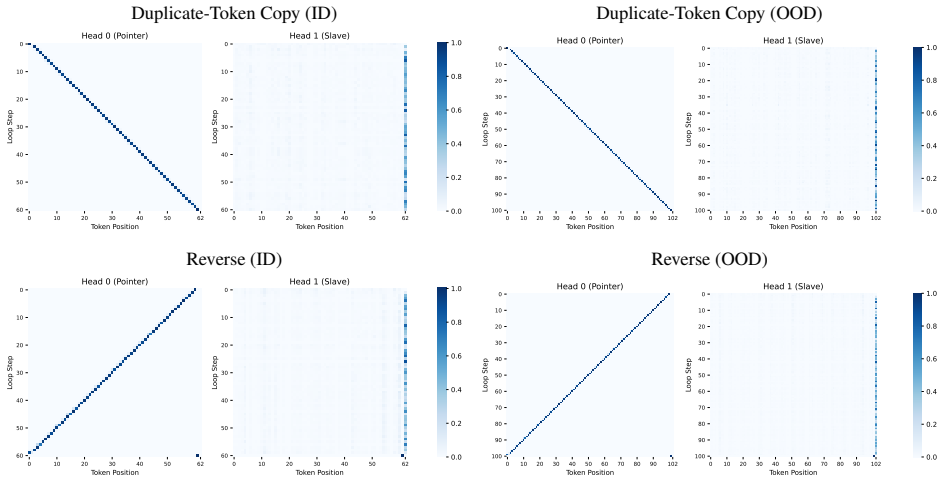

Figure 4: Pointer attention for forward and backward traversal. Rows denote tasks and columns compare an in-distribution length $L = 60$ with an OOD length $L = 100$. PoFormer spontaneously learns these interpretable traversal patterns without specific trajectory supervision.

Figure 4 visualizes representative pointer attention. Duplicate-Token Copy produces a forward diagonal and Reverse produces a backward diagonal, corresponding to the learned transitions $p_{t+1} = p_t + 1$ and $p_{t+1} = p_t - 1$. The same geometry persists at OOD lengths, supporting the interpretation that PoFormer has learned an extrapolatable relative transition rather than a position-specific lookup table. Complete layer-wise attention maps and further failure case analysis are provided in Appendix E.5.

5.3 Advanced Tasks

We further evaluate whether PoFormer can compose more complex pointer behaviors. In **Bidirectional Copy**, a single pointer head must encode both forward and backward traversal within shared query–key projections and select between them from an input command. **Binary Addition** instead requires two synchronized pointers to scan the operands right-to-left while the recurrent controller maintains the carry state. Complete task formats and architectural configurations are given in Appendix E.4. In preliminary experiments, we found that the task objective alone was insufficient for the model to learn the correct addressing policies on these advanced tasks. We therefore construct

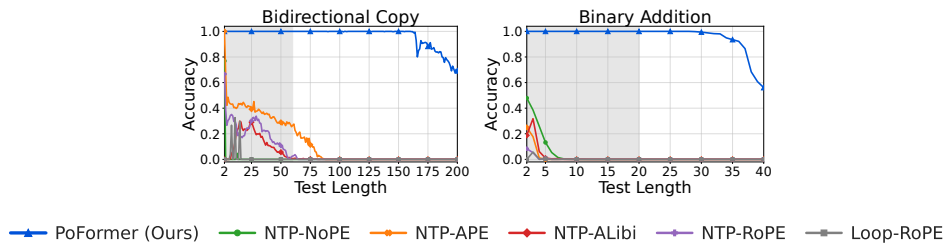

Figure 5: Length generalization on Bidirectional Copy (left) and Binary Addition (right) with auxiliary pointer supervision.

a P-RASP program for each task and translate its symbolic execution trace into target pointer trajectories used as auxiliary supervision. Let $j_t^{(\ell,h),*}$ be the target address for head h in layer ℓ at step t, and let $a_{t,j}^{(\ell,h)}$ be its attention distribution. We optimize

$$\mathcal{L} = \mathcal{L}_{\text{task}} - \frac{\lambda_{\text{ptr}}}{|\mathcal{I}_{\text{ptr}}|} \sum_{(t,\ell,h)\in\mathcal{I}_{\text{ptr}}} \log a_{t,j_t^{(\ell,h),*}}^{(\ell,h)}. \tag{18}$$

This auxiliary loss supervises only the accessed addresses; output values, state transitions, and termination remain learned from the task objective. As shown in Figure 5, PoFormer achieves perfect ID accuracy on both tasks. Bidirectional Copy remains nearly perfect up to approximately $2.75\times$ the maximum training length, while Binary Addition generalizes strongly up to approximately $1.75\times$.

5.4 Language Modeling Compatibility

Finally, we test whether introducing pointer attention degrades general language-modeling capability. We compare matched 0.6B-parameter Standard and PoFormer language models, each pretrained for 12B tokens on FineWeb-Edu (Penedo et al., 2024) with a context length of 2,048. We evaluate pretraining perplexity and three long-context QA tasks from RULER (Hsieh et al., 2024) and SCROLLS (Shaham et al., 2022), using the same instruction- and task-fine-tuning pipeline for both.

Table 1: Long-context QA results at 0.6B parameters. Context denotes the maximum evaluation sequence length; all results report token F1(↑).

Task	Context	Standard	PoFormer
RULER QA-2	4K	0.249	**0.268**
SCROLLS Qasper	8K	**0.110**	0.107
SCROLLS QuALITY	16K	0.393	**0.397**

PoFormer nearly matches the Standard model in validation perplexity (11.72 versus 11.69) and achieves comparable performance across the evaluated 4K–16K context lengths, with small mixed differences. Overall, introducing pointer attention causes no material degradation in either next-token modeling or the evaluated long-context QA tasks. We view this experiment as a compatibility check rather than evidence of improved natural-language length generalization: it shows that explicit pointer states can be incorporated into a standard language model without substantially compromising its conventional modeling capabilities. Full model, fine-tuning, and evaluation configurations are provided in Appendix E.6.

6 Conclusion

Length-generalizing computation requires knowing both *what* has been computed and *where* execution currently is. PoFormer makes the latter explicit by recurrently updating active pointer states over static memory. P-RASP formalizes this view and characterizes scans, jumps, and finite-state updates as recurrent pointer programs. The experiments show that this inductive bias yields substantially stronger extrapolation and turns opaque output failures into inspectable address-level errors.

AI USE STATEMENT

We used generative AI tools to assist with manuscript writing and editing, mathematical exposition, code implementation and debugging, and scientific plotting. All AI-assisted text, proofs, code, figures, and citations were reviewed and verified by the authors; reported results were produced by our experiments rather than generated by AI. We take full responsibility for the final content of this work.

REPRODUCIBILITY STATEMENT

To facilitate reproducibility, the main paper and appendices provide detailed specifications of the model architectures, training and evaluation protocols, and hyperparameter settings used in our experiments. The appendices additionally document the task-generation procedures and experimental configurations. The anonymized supplementary material includes the core source code for implementing the models and reproducing the main training and evaluation experiments.

REFERENCES

Ido Amos, Avi Caciularu, Mor Geva, Amir Globerson, Jonathan Herzig, Lior Shani, and Idan Szpektor. Latent reasoning with supervised thinking states, 2026. URL https://arxiv.org/abs/2602.08332.

Pranjal Awasthi, Anupam Gupta, and Ravi Kumar. Length generalization via auxiliary tasks. In *The Thirty-ninth Annual Conference on Neural Information Processing Systems*, 2025. URL https://openreview.net/forum?id=QL3J1fyAFv.

Alireza Amiri Bavandpour, Xinting Huang, Mark Rofin, and Michael Hahn. Lower bounds for chain-of-thought reasoning in hard-attention transformers. In *Forty-second International Conference on Machine Learning*, 2025. URL https://openreview.net/forum?id=Oh9sG5ae2b.

Ziyang Cai, Nayoung Lee, Avi Schwarzschild, Samet Oymak, and Dimitris Papailiopoulos. Extrapolation by association: Length generalization transfer in transformers. In *The Thirty-ninth Annual Conference on Neural Information Processing Systems*, 2025. URL https://openreview.net/forum?id=aLUAzLDIOc.

Hanseul Cho, Jaeyoung Cha, Srinadh Bhojanapalli, and Chulhee Yun. Arithmetic transformers can length-generalize in both operand length and count. In *The Thirteenth International Conference on Learning Representations*, 2025. URL https://openreview.net/forum?id=eIgGesYKLG.

Mike Conover, Matt Hayes, Ankit Mathur, Jianwei Xie, Jun Wan, Sam Shah, Ali Ghodsi, Patrick Wendell, Matei Zaharia, and Reynold Xin. Free dolly: Introducing the world's first truly open instruction-tuned llm, 2023. URL https://www.databricks.com/blog/2023/04/12/dolly-first-open-commercially-viable-instruction-tuned-llm.

Róbert Csordás, Kazuki Irie, and Jürgen Schmidhuber. The neural data router: Adaptive control flow in transformers improves systematic generalization. In *International Conference on Learning Representations*, 2022. URL https://openreview.net/forum?id=KBQP4A_J1K.

Mostafa Dehghani, Stephan Gouws, Oriol Vinyals, Jakob Uszkoreit, and Lukasz Kaiser. Universal transformers. In *International Conference on Learning Representations*, 2019. URL https://openreview.net/forum?id=HyzdRiR9Y7.

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: Pre-training of deep bidirectional transformers for language understanding. In Jill Burstein, Christy Doran, and Thamar Solorio (eds.), *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pp. 4171–4186, Minneapolis, Minnesota, June 2019. Association for Computational Linguistics. doi: 10.18653/v1/N19-1423. URL https://aclanthology.org/N19-1423/.

Yufeng Du, Phillip Harris, Minyang Tian, Eliu A Huerta, Srikanth Ronanki, Subendhu Rongali, Aram Galstyan, and Hao Peng. Rope distinguishes neither positions nor tokens in long contexts, provably, 2026. URL https://arxiv.org/abs/2605.15514.

Shaoxiong Duan and Yining Shi. From interpolation to extrapolation: Complete length generalization for arithmetic transformers, 2024. URL https://openreview.net/forum?id=0sbIEkIutN.

MohammadReza Ebrahimi, Sunny Panchal, and Roland Memisevic. Your context is not an array: Unveiling random access limitations in transformers. In *First Conference on Language Modeling*, 2024. URL https://openreview.net/forum?id=MLD1cwfjUb.

Reza Ebrahimi, Michaël Defferrard, Sunny Panchal, and Roland Memisevic. On the "induction bias" in sequence models. In *ICLR 2026 Workshop on Logical Reasoning of Large Language Models*, 2026. URL https://openreview.net/forum?id=Y4S6W5EMb3.

Ying Fan, Yilun Du, Kannan Ramchandran, and Kangwook Lee. Looped transformers for length generalization. In *The Thirteenth International Conference on Learning Representations*, 2025. URL https://openreview.net/forum?id=2edigk8yoU.

Shivam Garg, Dimitris Tsipras, Percy Liang, and Gregory Valiant. What can transformers learn in-context? a case study of simple function classes. In Alice H. Oh, Alekh Agarwal, Danielle Belgrave, and Kyunghyun Cho (eds.), *Advances in Neural Information Processing Systems*, 2022. URL https://openreview.net/forum?id=flNZJ2eOet.

Jonas Geiping, Sean Michael 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. In *The Thirty-ninth Annual Conference on Neural Information Processing Systems*, 2025. URL https://openreview.net/forum?id=S3GhJooWIC.

Noah Golowich, Samy Jelassi, David Brandfonbrener, Sham M. Kakade, and Eran Malach. The role of sparsity for length generalization in LLMs. In *Forty-second International Conference on Machine Learning*, 2025. URL https://openreview.net/forum?id=S9LkB0UBKb.

Alex Graves, Greg Wayne, and Ivo Danihelka. Neural turing machines. *CoRR*, abs/1410.5401, 2014. URL http://arxiv.org/abs/1410.5401.

Shibo Hao, Sainbayar Sukhbaatar, DiJia Su, Xian Li, Zhiting Hu, Jason E Weston, and Yuandong Tian. Training large language models to reason in a continuous latent space. In *Second Conference on Language Modeling*, 2025. URL https://openreview.net/forum?id=Itxz7S4Ip3.

Adi Haviv, Ori Ram, Ofir Press, Peter Izsak, and Omer Levy. Transformer language models without positional encodings still learn positional information. In Yoav Goldberg, Zornitsa Kozareva, and Yue Zhang (eds.), *Findings of the Association for Computational Linguistics: EMNLP 2022*, pp. 1382–1390, Abu Dhabi, United Arab Emirates, December 2022. Association for Computational Linguistics. doi: 10.18653/v1/2022.findings-emnlp.99. URL https://aclanthology.org/2022.findings-emnlp.99/.

Cheng-Ping Hsieh, Simeng Sun, Samuel Kriman, Shantanu Acharya, Dima Rekesh, Fei Jia, and Boris Ginsburg. RULER: What's the real context size of your long-context language models? In *First Conference on Language Modeling*, 2024. URL https://openreview.net/forum?id=kIoBbc76Sy.

Xinting Huang, Andy Yang, Satwik Bhattamishra, Yash Sarrof, Andreas Krebs, Hattie Zhou, Preetum Nakkiran, and Michael Hahn. A formal framework for understanding length generalization in transformers. In *The Thirteenth International Conference on Learning Representations*, 2025. URL https://openreview.net/forum?id=U49N5V51rU.

Ahmadreza Jeddi, Marco Ciccone, and Babak Taati. Loopformer: Elastic-depth looped transformers for latent reasoning via shortcut modulation. In *The Fourteenth International Conference on Learning Representations*, 2026. URL https://openreview.net/forum?id=RzYXb5YWBs.

Samy Jelassi, David Brandfonbrener, Sham M. Kakade, and eran malach. Repeat after me: Transformers are better than state space models at copying. In *Forty-first International Conference on Machine Learning*, 2024. URL https://openreview.net/forum?id=duRRoGeoQT.

Amirhossein Kazemnejad, Inkit Padhi, Karthikeyan Natesan, Payel Das, and Siva Reddy. The impact of positional encoding on length generalization in transformers. In *Thirty-seventh Conference on Neural Information Processing Systems*, 2023. URL https://openreview.net/forum?id=Drrl2gcjzl.

Juno Kim and Taiji Suzuki. Transformers provably solve parity efficiently with chain of thought. In *The Thirteenth International Conference on Learning Representations*, 2025. URL https://openreview.net/forum?id=n2NidsYDop.

Alexander Kozachinskiy, Tomasz Steifer, and Przemysław Wałęga. Parity, sensitivity, and transformers, 2026. URL https://arxiv.org/abs/2602.05896.

Karol Kurach, Marcin Andrychowicz, and Ilya Sutskever. Neural random access machines. *ICLR*, 2016. URL http://arxiv.org/abs/1511.06392.

Hung Le, Dung Nguyen, Kien Do, Svetha Venkatesh, and Truyen Tran. Plug, play, and generalize: Length extrapolation with pointer-augmented neural memory. *Transactions on Machine Learning Research*, 2024. ISSN 2835-8856. URL https://openreview.net/forum?id=dyQ9vFbF6D.

Shanda Li, Chong You, Guru Guruganesh, Joshua Ainslie, Santiago Ontanon, Manzil Zaheer, Sumit Sanghai, Yiming Yang, Sanjiv Kumar, and Srinadh Bhojanapalli. Functional interpolation for relative positions improves long context transformers. In *The Twelfth International Conference on Learning Representations*, 2024a. URL https://openreview.net/forum?id=rR03qFesqk.

Zhiyuan Li, Hong Liu, Denny Zhou, and Tengyu Ma. Chain of thought empowers transformers to solve inherently serial problems. In *The Twelfth International Conference on Learning Representations*, 2024b. URL https://openreview.net/forum?id=3EWTEy9MTM.

William Merrill and Ashish Sabharwal. Exact expressive power of transformers with padding. In *The Thirty-ninth Annual Conference on Neural Information Processing Systems*, 2025. URL https://openreview.net/forum?id=O1abxStFcy.

Guilherme Penedo, Hynek Kydlíček, Loubna Ben allal, Anton Lozhkov, Margaret Mitchell, Colin Raffel, Leandro Von Werra, and Thomas Wolf. The fineweb datasets: Decanting the web for the finest text data at scale. In *The Thirty-eight Conference on Neural Information Processing Systems Datasets and Benchmarks Track*, 2024. URL https://openreview.net/forum?id=n6SCkn2QaG.

Hadi Pouransari, Chun-Liang Li, Jen-Hao Rick Chang, Pavan Kumar Anasosalu Vasu, Cem Koc, Vaishaal Shankar, and Oncel Tuzel. Dataset decomposition: Faster LLM training with variable sequence length curriculum. In *The Thirty-eighth Annual Conference on Neural Information Processing Systems*, 2024. URL https://openreview.net/forum?id=r8M9SfYMDi.

Ofir Press, Noah Smith, and Mike Lewis. Train short, test long: Attention with linear biases enables input length extrapolation. In *International Conference on Learning Representations*, 2022. URL https://openreview.net/forum?id=R8sQPpGCv0.

Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. Exploring the limits of transfer learning with a unified text-to-text transformer. *J. Mach. Learn. Res.*, 21:140:1–140:67, 2020. URL https://jmlr.org/papers/volume21/20-074/20-074.pdf.

Nikunj Saunshi, Nishanth Dikkala, Zhiyuan Li, Sanjiv Kumar, and Sashank J. Reddi. Reasoning with latent thoughts: On the power of looped transformers. In *The Thirteenth International Conference on Learning Representations*, 2025. URL https://openreview.net/forum?id=din0lGfZFd.

Uri Shaham, Elad Segal, Maor Ivgi, Avia Efrat, Ori Yoran, Adi Haviv, Ankit Gupta, Wenhan Xiong, Mor Geva, Jonathan Berant, and Omer Levy. SCROLLS: Standardized CompaRison over long language sequences. In *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing*, pp. 12007–12021, Abu Dhabi, United Arab Emirates, dec 2022. Association for Computational Linguistics. URL https://aclanthology.org/2022.emnlp-main.823.

Jianlin Su, Murtadha Ahmed, Yu Lu, Shengfeng Pan, Wen Bo, and Yunfeng Liu. Roformer: Enhanced transformer with rotary position embedding. *Neurocomput.*, 568(C), February 2024. ISSN 0925-2312. doi: 10.1016/j.neucom.2023.127063. URL https://doi.org/10.1016/j.neucom.2023.127063.

Oriol Vinyals, Meire Fortunato, and Navdeep Jaitly. Pointer networks. In C. Cortes, N. Lawrence, D. Lee, M. Sugiyama, and R. Garnett (eds.), *Advances in Neural Information Processing Systems*, volume 28. Curran Associates, Inc., 2015. URL https://proceedings.neurips.cc/paper_files/paper/2015/file/29921001f2f04bd3baee84a12e98098f-Paper.pdf.

Jiecong Wang, Hao Peng, and Chunyang Liu. Latent chain-of-thought as planning: Decoupling reasoning from verbalization, 2026. URL https://arxiv.org/abs/2601.21358.

Gail Weiss, Yoav Goldberg, and Eran Yahav. Thinking like transformers. In *ICML*, volume 139 of *Proceedings of Machine Learning Research*, pp. 11080–11090. PMLR, 2021. URL https://proceedings.mlr.press/v139/weiss21a/weiss21a.pdf.

Kevin Xu and Issei Sato. A formal comparison between chain-of-thought and latent thought, 2026. URL https://openreview.net/forum?id=KE5C7qK82R.

Andy Yang and David Chiang. Counting like transformers: Compiling temporal counting logic into softmax transformers. In *First Conference on Language Modeling*, 2024. URL https://openreview.net/forum?id=FmhPg4UJ9K.

Chengting Yu, Xiaobo Shu, Yadao Wang, Yizhen Zhang, Haoyi Wu, You Wu, Rujiao Long, Ziheng Chen, Yuchi Xu, Wenbo Su, and Bo Zheng. Spiralformer: Looped transformers can learn hierarchical dependencies via multi-resolution recursion, 2026. URL https://arxiv.org/abs/2602.11698.

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*, 2024a. URL https://openreview.net/forum?id=AssIuHnmHX.

Yongchao Zhou, Uri Alon, Xinyun Chen, Xuezhi Wang, Rishabh Agarwal, and Denny Zhou. Transformers can achieve length generalization but not robustly. In *ICLR 2024 Workshop on Mathematical and Empirical Understanding of Foundation Models*, 2024b. URL https://openreview.net/forum?id=DWkWIh3vFJ.

A Discussion and Limitations

Our results support the hypothesis that traversal-based length generalization requires not only recurrent computation, but also a persistent mechanism for controlling where that computation takes place. Standard and looped Transformers can propagate intermediate computational states, yet must encode the currently relevant memory location implicitly in their hidden representations. PoFormer separates these roles: the controller carries computational state, while explicit pointers maintain and update address state. The learned attention trajectories and the P-RASP analysis provide complementary evidence for this interpretation, showing that traversal tasks can be expressed as repeated memory reads followed by learned address transitions.

Relation to positional anchors Anchor and landmark methods convert positional addressing into content matching by inserting distinguishable markers into the input (Zhou et al., 2024a; Ebrahimi et al., 2024; Zhou et al., 2024b; Cho et al., 2025; Golowich et al., 2025). These methods can provide highly reliable addresses when suitable markers are available, but they intrusively modify the input representation and the number of unique anchors is constrained by the vocabulary and task format. PoFormer instead places the addressing bias inside the architecture and can repeatedly update an address without requiring a unique token at every location.

Relation to biased attention structure Task-specific attention masks and positional biases directly favor a known predecessor, successor, or relative offset (Csordás et al., 2022; Duan & Shi, 2024). Such mechanisms can be stable, sample-efficient, and highly interpretable when the desired traversal rule is fixed in advance. PoFormer does not prescribe a particular transition: the same pointer mechanism can learn forward and backward shifts, controller-dependent choices, and content-conditioned non-local jumps. This flexibility comes at the cost of having to discover the appropriate traversal policy through optimization.

Relation to auxiliary-task transfer Auxiliary-task approaches first learn an attention transition on a simpler or less expensive task and then transfer it to the target problem (Awasthi et al., 2025; Cai et al., 2025). They can be effective when a well-matched auxiliary task is available, but depend on identifying such a task and designing an appropriate transfer procedure. PoFormer can learn the basic traversal tasks directly from their output sequences; nevertheless, our advanced tasks show that difficult address policies may still benefit from additional explicit trajectory supervision.

Relation to length-generalizing looped Transformers The length-generalizing looped Transformer of Fan et al. (2025) pre-allocates padding tokens, applies recurrent computation to all token states, and uses a prespecified iteration schedule. PoFormer instead retains an NTP-compatible decoding interface: it emits one token at each decoding checkpoint and learns program termination by predicting <eos>, without requiring padding-based output slots or a separate halting component. The two approaches nevertheless provide different inductive biases. Updating all token positions in parallel is particularly effective for tasks such as Binary Addition, whereas PoFormer's sequential pointer-based formulation requires auxiliary trajectory supervision to reliably learn this task in our experiments.

Relation to Pointer- and Address-Based Neural Networks PoFormer is related to prior pointer- and address-based neural architectures, including Neural Turing Machines (Graves et al., 2014), Pointer Networks (Vinyals et al., 2015), NRAM (Kurach et al., 2016), and more recent pointer-augmented memory models (Le et al., 2024). However, our goal is not to replace the Transformer with a separate or external memory architecture, but to augment the existing attention mechanism with an explicit recurrent address state. In PoFormer, the static input prefix remains the memory accessed by ordinary Transformer attention, while pointer states modify the positional geometry of selected attention heads and are updated through the same attention distributions used for content retrieval. The resulting inductive bias therefore differs from architectures in which addressing is implemented through a dedicated external memory interface or task-specific pointer module: PoFormer preserves the standard attention pathway for content-based retrieval while adding explicit control over where that retrieval is performed.

This distinction is also reflected empirically. On Shuffled Linked-List Traversal, successful execution requires content-conditioned retrieval of non-adjacent memory records, showing that pointer attention does not reduce the model to purely positional traversal. At larger scale, the language-modeling experiment shows no material degradation in validation perplexity or the evaluated long-context QA tasks relative to a matched Standard Transformer. We therefore view PoFormer as an attention-compatible augmentation that adds an explicit addressing inductive bias while largely preserving the content-based modeling capabilities of standard attention, rather than as a fundamentally separate neural-memory architecture.

Limitations Prior work suggests that due to the inherent limitations of RoPE, RoPE-based attention cannot reliably distinguish positions over arbitrarily long ranges (Du et al., 2026). Consequently, PoFormer does not guarantee unbounded extrapolation, and errors in its continuous pointer updates may accumulate across recurrent steps. Accordingly, our theory establishes finite-range expressivity

rather than guaranteed learnability or arbitrary-length generalization. Moreover, the advanced tasks require auxiliary pointer supervision, indicating that output supervision alone may not identify complex traversal policies. Recurrent unrolling also introduces sequential latency and additional training cost relative to parallel teacher-forced Transformers. Finally, our main evidence comes from synthetic algorithmic tasks. The LM experiment shows that pointer attention preserves language-modeling and long-context QA performance, but does not yet demonstrate improved length generalization in natural-language settings.

B Formal Definition and Semantics of P-RASP

This appendix defines the state space and operational semantics of P-RASP. Following RASP and its length-generalization-oriented variants (Weiss et al., 2021; Zhou et al., 2024a; Yang & Chiang, 2024; Huang et al., 2025), P-RASP represents intermediate sequence values as sequence operations (*s-ops*) over a length-L memory. It retains three standard classes of RASP-style operations: elementwise transformations apply a function independently at each position; selectors construct a binary relation between query and key positions using a predicate; and aggregations collect the values chosen by a selector. These operations are used to preprocess the input, construct content predicates, and compute memory readouts.

P-RASP extends this sequence-computation interface with a finite recurrent controller and persistent pointer variables over static memory. The $\mathrm{p_shift}$ primitive applies a position-independent relative displacement to a pointer, whereas $\mathrm{p_jump}$ uses a RASP-style content selector and aggregates the selected position as the pointer's next address. The read operation selects and aggregates the memory value at the current address, after which update and emit apply elementwise finite-state transformations to update the controller and produce an output. Thus, the standard RASP operations determine *what* is selected or computed, while the new pointer primitives preserve and update *where* recurrent computation proceeds.

The definitions below restate every inherited operation used by P-RASP and give separate semantics for the pointer extensions. The relationship between P-RASP and PoFormer is established separately by the simulation results in Section 4.2.

B.1 Program State

A P-RASP program operates on a static memory, a recurrent controller state, and one or more recurrent pointer states.

Static memory Let

$$M = (M_1, \ldots, M_L), \qquad M_i \in \Sigma, \tag{19}$$

be an input memory of length L, where Σ is the input alphabet. Each memory cell M_i is associated with a physical address

$$r_i = i, \qquad i \in [L] := \{1, \ldots, L\}. \tag{20}$$

The memory is initialized before recurrent execution and is not modified by the loop program. Thus, the same memory cells remain available at every iteration.

Although we write $M_i \in \Sigma$ for simplicity, a memory cell may contain multiple symbolic fields, such as a token identity, a delimiter indicator, a key, or a value. These fields can be extracted and transformed using the elementwise operations inherited from RASP-style languages.

Controller state At recurrent step t, the program maintains a controller state

$$s_t \in \mathcal{S}, \tag{21}$$

where $\mathcal{S}$ is a finite state space. The controller records non-positional information required by the algorithm, such as a parity bit, a traversal phase, an output mode, or a finite collection of flags.

The symbolic state s_t abstracts the continuous controller representation used by PoFormer. In particular, different elements of $\mathcal{S}$ correspond to distinguishable regions or representations in the neural hidden-state space.

Pointer state In the single-pointer setting, the active address at recurrent step t is

$$p_t \in [L]. \tag{22}$$

With H pointers, the address state is

$$\mathbf{p}_t = \left(p_t^{(1)}, \ldots, p_t^{(H)}\right) \in [L]^H. \tag{23}$$

Each pointer identifies one currently active memory location. Unlike a static positional selector, the pointer is part of the recurrent program state and therefore persists across loop iterations.

A complete P-RASP configuration is consequently

$$\mathcal{C}_t = (M, s_t, \mathbf{p}_t)\,. \tag{24}$$

The recurrent loop leaves M unchanged while updating s_t and $\mathbf{p}_t$.

B.2 Reading and Controller Computation

Before updating its state, the program may retrieve values from the memory locations addressed by its pointers.

Read For a single pointer, the read operation is

$$\mathrm{read}(M, p_t) = M_{p_t}. \tag{25}$$

With H pointers, the joint readout is

$$x_t = \mathrm{read}(M, \mathbf{p}_t) = \left(M_{p_t^{(1)}}, \ldots, M_{p_t^{(H)}}\right). \tag{26}$$

Elementwise transformations and aggregation operations may subsequently be applied to these values. The read operation therefore specifies the addressed memory cells, rather than prescribing a particular representation of their contents.

The distinction between an address and a read value is important. The pointer $p_t^{(h)}$ records *where* head h is located, whereas $M_{p_t^{(h)}}$ specifies *what* is stored at that location. Two memory locations may contain identical symbols while still being distinguished by their pointer addresses.

Controller update The controller is updated using its current state and the retrieved memory values:

$$s_{t+1} = \mathrm{update}(s_t, x_t), \tag{27}$$

where

$$\mathrm{update} : \mathcal{S} \times \Sigma^H \to \mathcal{S} \tag{28}$$

is a finite-state transition function. If memory cells contain structured fields, the domain Σ^H may be replaced by the corresponding finite readout domain.

This operation covers computations such as toggling a parity state, recording whether a boundary has been encountered, or selecting the next phase of a multi-stage traversal. It does not itself modify the pointer addresses.

Emit At an output checkpoint, the program produces a symbol using an emission function

$$y_t = \mathrm{emit}(s_t, x_t), \qquad \mathrm{emit} : \mathcal{S} \times \Sigma^H \to \Gamma, \tag{29}$$

where Γ is the output alphabet. The emission may depend only on the controller state, only on a retrieved value, or on both. Emission does not modify the static memory and does not append the predicted symbol to it.

The emission operation need not be applied at every recurrent iteration. For a PoFormer that decodes one token every k loop steps, it is evaluated only at iterations $t \in \{k, 2k, \ldots\}$.

B.3 Pointer-Update Primitives

The principal extension introduced by P-RASP is that selection may depend on a persistent address state. Pointer dynamics are expressed through two primitive operations: a translation-invariant relative shift and a content-conditioned jump.

B.3.1 Constant Relative Shift

For a constant integer offset $\Delta \in \mathbb{Z}$, the shift primitive is defined as

$$\mathrm{p_shift}(p_t, \Delta) = p_t + \Delta. \tag{30}$$

The offset Δ is independent of the current address p_t. Consequently, applying the same primitive at different memory locations produces the same relative displacement:

$$\mathrm{p_shift}(p + a, \Delta) = \mathrm{p_shift}(p, \Delta) + a \tag{31}$$

whenever both sides are valid memory addresses. The operation is therefore translation-invariant.

Typical instances include

$$\mathrm{p_shift}(p_t, +1) = p_t + 1, \qquad \text{forward traversal,} \tag{32}$$
$$\mathrm{p_shift}(p_t, -1) = p_t - 1, \qquad \text{backward traversal,} \tag{33}$$
$$\mathrm{p_shift}(p_t, 0) = p_t, \qquad \text{remaining at the current address.} \tag{34}$$

Offsets with $|\Delta| > 1$ represent fixed-stride traversal.

A single invocation of $\mathrm{p_shift}$ uses a fixed constant Δ. A program may nevertheless contain multiple shift primitives and use its finite controller state to select which one is applied. For example, one controller state may invoke $\mathrm{p_shift}(\cdot, +1)$ and another may invoke $\mathrm{p_shift}(\cdot, -1)$. Thus, the choice of operation may be state-dependent even though each individual shift remains position-independent.

The primitive is defined only when

$$p_t + \Delta \in [L]. \tag{35}$$

Clipping, circular wrapping, and reflection at the memory boundary are not implicitly included in $\mathrm{p_shift}$. If such behavior is required, it must be specified explicitly using controller transitions, boundary predicates, or a jump to a distinguished boundary location. This convention prevents the primitive from introducing an unstated boundary-specific inductive bias.

B.3.2 Content-Conditioned Jump

Whereas $\mathrm{p_shift}$ moves relative to the current address, $\mathrm{p_jump}$ selects a new address by matching a predicate against the memory. Let

$$\phi : \Sigma \times \mathcal{S} \times [L] \times [L] \to \{0, 1\} \tag{36}$$

be a selection predicate. For a candidate memory cell M_i, the value

$$\phi(M_i, s_t, p_t, r_i) \tag{37}$$

may depend on its content, the current controller state, the current pointer, and the candidate's physical address. The jump primitive is defined by

$$\mathrm{p_jump}(M, s_t, p_t, \phi) = r_{i^*}, \tag{38}$$

where i^* is the unique index satisfying

$$\phi(M_{i^*}, s_t, p_t, r_{i^*}) = 1. \tag{39}$$

Unlike a shift, the displacement produced by a jump need not be constant:

$$\mathrm{p_jump}(M, s_t, p_t, \phi) - p_t \tag{40}$$

may vary with the memory contents and the current program state. The operation can therefore express non-local transitions that cannot be represented by a single fixed relative offset.

For example, resetting a pointer to a unique beginning-of-sequence token can be written using

$$\phi_{\text{bos}}(M_i, s_t, p_t, r_i) \equiv \mathbf{1}[M_i = \texttt{<bos>}]. \tag{41}$$

Similarly, if each memory cell contains a key–value pair $M_i = (k_i, v_i)$ and the controller stores a target key q_t, a linked-memory transition can use

$$\phi_{\text{key}}(M_i, s_t, p_t, r_i) \equiv \mathbf{1}[k_i = q_t]. \tag{42}$$

The resulting jump moves the pointer to the memory cell whose key matches the query stored in the controller.

The uniqueness requirement in Equation 39 makes the symbolic operation deterministic. If multiple locations satisfy a predicate, the program must supply an additional selection rule, such as choosing the first or last matching position. Alternatively, the matching symbol can be paired with a unique identifier. If no candidate or more than one candidate is selected and no tie-breaking rule is specified, $\mathrm{p_jump}$ is undefined.

Although the general notation permits dependence on p_t and r_i, common content-conditioned jumps use the simpler predicate

$$\phi(M_i, s_t), \tag{43}$$

which selects a target using only its content and the controller state. Pointer-dependent predicates are useful when the admissible target is defined relative to the previously visited location.

B.3.3 Multiple Pointers

For H pointers, each pointer may be updated independently:

$$p_{t+1}^{(h)} = \Psi_h\left(M, s_t, \mathbf{p}_t, x_t\right), \qquad h \in [H], \tag{44}$$

where Ψ_h is instantiated by a shift, a jump, or a finite composition of pointer primitives. Different pointers may serve different algorithmic roles; for example, one may traverse the input while another records an output boundary or follows a linked-memory chain.

The number of pointers H is fixed by the program and does not grow with the input length. Therefore, pointers provide a fixed-dimensional recurrent address state rather than an input-length-sized writable memory.

B.4 Semantics of a Recurrent Step

A recurrent P-RASP step first reads the locations addressed by the current pointers and then updates the controller and address states. Using simultaneous update semantics, one loop iteration is

$$x_t = \text{read}(M, \mathbf{p}_t), \tag{45}$$

$$y_t = \text{emit}(s_t, x_t), \tag{46}$$

$$s_{t+1} = \text{update}(s_t, x_t), \tag{47}$$

$$p_{t+1}^{(h)} = \Psi_h(M, s_t, \mathbf{p}_t, x_t), \qquad h \in [H]. \tag{48}$$

Here, both s_{t+1} and $\mathbf{p}_{t+1}$ are computed from the state at iteration t. This convention avoids an ambiguity about whether a pointer transition observes the old or newly updated controller state. A program that requires the latter behavior can represent it using an additional loop iteration or by composing the relevant finite transition into Ψ_h.

Equivalently, the loop defines a transition

$$P_{\text{loop}} : (M, s_t, \mathbf{p}_t) \mapsto (M, s_{t+1}, \mathbf{p}_{t+1}, y_t). \tag{49}$$

The memory appears unchanged on both sides of this transition.

B.5 Complete P-RASP Programs

As defined in Section 4.1, a complete P-RASP program has the form

$$P = P_{\text{post}} \circ \left(P_{\text{loop}}\right)^{T(L)} \circ P_{\text{pre}}. \tag{50}$$

The three components have the following roles:

- P_{pre} constructs the static memory and initializes the controller and pointer states.
- P_{loop} repeatedly applies the read, controller-update, emission, and pointer-update operations.
- P_{post} maps the final state or the sequence of emitted symbols to the required output format.

The iteration count $T(L)$ may depend on the input length but the definition of P_{loop} and its parameters remain fixed. For example, a sequential scan may use $T(L) = L$, while an algorithm that performs a fixed number of full passes may use $T(L) = cL$ for a constant c.

The essential difference between P-RASP and existing RASP-style formalisms is not merely the repeated application of a fixed program. It is the inclusion of $\mathbf{p}_t$ in the recurrent state. This enables a memory location selected at iteration t to become the positional reference for iteration $t + 1$, thereby representing an explicit traversal trajectory over the static memory.

C Constructive Expressivity Proofs

This appendix provides the component constructions underlying Theorem 4.4. The goal is to establish architectural expressivity rather than learnability under gradient descent.

C.1 Assumptions

Assumption C.1 (Finite-range sharp selection)**.** *Consider a finite set of memory positions and a finite displacement range. Symbolic token and controller predicates admit linearly separable representations with positive margins. Moreover, query–key scores can be scaled so that softmax attention approximates any uniquely selected target position arbitrarily closely.*

We additionally assume that the RoPE frequencies used for positional selection distinguish all required displacements in the considered range. This is a finite-range condition: the results do not imply alias-free addressing at arbitrarily large positions according to the limitations suggested in (Du et al., 2026).

C.2 Address Resolution

Lemma C.2 (Pointer-based address resolution)**.** *Suppose that active pointer p is aligned with one physical memory position r_{j^*}. Under Assumption C.1, there exist query and key projections such that a pointer head attends arbitrarily closely to the unique position satisfying*

$$r_{j^*} = p \tag{51}$$

and retrieves its associated value.

Proof. For each memory position j, let

$$\delta_j = r_j - p \tag{52}$$

denote its displacement from the active pointer. Consider one two-dimensional RoPE block with frequency θ:

$$R_{\delta_j}^{(\theta)} = \begin{bmatrix} \cos(\delta_j\theta) & -\sin(\delta_j\theta) \\ \sin(\delta_j\theta) & \cos(\delta_j\theta) \end{bmatrix}. \tag{53}$$

Choose the projected query and key in this block as

$$q^{(\theta)} = k^{(\theta)} = \begin{bmatrix} \sqrt{c} \\ 0 \end{bmatrix}, \qquad c > 0. \tag{54}$$

The resulting positional score is

$$\begin{aligned} \left(q^{(\theta)}\right)^\top R_{\delta_j}^{(\theta)} k^{(\theta)} &= \begin{bmatrix} \sqrt{c} & 0 \end{bmatrix} \begin{bmatrix} \cos(\delta_j\theta) & -\sin(\delta_j\theta) \\ \sin(\delta_j\theta) & \cos(\delta_j\theta) \end{bmatrix} \begin{bmatrix} \sqrt{c} \\ 0 \end{bmatrix} \\ &= c\cos(\delta_j\theta). \end{aligned} \tag{55}$$

In particular, when $r_j = p$, we have $\delta_j = 0$ and the score equals c, its maximum possible value.

Using D RoPE blocks with frequencies $\theta_1, \ldots, \theta_D$ and choosing $q^{(m)} = k^{(m)} = (\sqrt{c_m}, 0)^\top$ in block m gives

$$K(\delta_j) = q^\top R_{\delta_j} k = \sum_{m=1}^{D} c_m \cos(\delta_j \theta_m), \qquad c_m > 0. \tag{56}$$

At the addressed position j^*, where $\delta_{j^*} = 0$,

$$K(0) = \sum_{m=1}^{D} c_m. \tag{57}$$

For every nonzero displacement in the considered finite range, Assumption C.1 guarantees that the RoPE features do not alias with those at zero. Hence at least one cosine term is strictly less than one, while none exceeds one, so

$$K(0) > K(\delta_j), \qquad j \neq j^*. \tag{58}$$

Because the set of admissible displacements is finite, there exists a uniform margin $\gamma > 0$ such that

$$s_{j^*} - s_j \geq \gamma, \qquad j \neq j^*. \tag{59}$$

After scaling the logits by $\beta > 0$, the attention probability assigned to j^* satisfies

$$a_{j^*} = \frac{e^{\beta s_{j^*}}}{\sum_{j=1}^{L} e^{\beta s_j}} \geq \frac{1}{1 + (L-1)e^{-\beta\gamma}} \xrightarrow[\beta \to \infty]{} 1. \tag{60}$$

Finally, choosing the value projection to preserve the required memory value makes the attention output converge to the value stored at j^*. Therefore, the pointer head implements pointer-based address resolution to arbitrary accuracy. □

Lemma C.3 (Controller-conditioned relative shift)**.** *Let $\mathcal{D} \subset \mathbb{R}$ be a finite set of offsets, and let $\Delta : \mathcal{S} \to \mathcal{D}$ map each finite controller state $s_t \in \mathcal{S}$ to an offset. Suppose that the current pointer p_t is aligned with a physical memory position and that the target address*

$$p_t + \Delta(s_t) \tag{61}$$

coincides with a valid memory position that is uniquely distinguishable within the considered RoPE range. Then a pointer head can implement

$$p_{t+1} = p_t + \Delta(s_t) \tag{62}$$

to arbitrary accuracy.

Although P-RASP restricts pointer offsets to integers because its memory is indexed by discrete positions, PoFormer's RoPE-based addressing mechanism is well-defined for real-valued pointer positions and offsets. The P-RASP shift primitive is therefore recovered as the special case $\mathcal{D} \subset \mathbb{Z}$.

Proof. Let

$$\delta_j = r_j - p_t \tag{63}$$

be the displacement of memory position j from the current pointer. Consider one two-dimensional RoPE block with frequency θ:

$$R_{\delta_j}^{(\theta)} = \begin{bmatrix} \cos(\delta_j\theta) & -\sin(\delta_j\theta) \\ \sin(\delta_j\theta) & \cos(\delta_j\theta) \end{bmatrix}. \tag{64}$$

For the controller state s_t, write

$$\Delta_t := \Delta(s_t). \tag{65}$$

Choose the key vector and the controller-conditioned query vector as

$$k^{(\theta)} = \begin{bmatrix} \sqrt{c} \\ 0 \end{bmatrix}, \qquad q^{(\theta)}(s_t) = R_{\Delta_t}^{(\theta)} k^{(\theta)} = \begin{bmatrix} \sqrt{c}\cos(\Delta_t\theta) \\ \sqrt{c}\sin(\Delta_t\theta) \end{bmatrix}, \qquad c > 0. \tag{66}$$

The pointer-relative attention score at position j is therefore

$$\begin{aligned} \big(q^{(\theta)}(s_t)\big)^\top R_{\delta_j}^{(\theta)} k^{(\theta)} &= \begin{bmatrix} \sqrt{c}\cos(\Delta_t\theta) & \sqrt{c}\sin(\Delta_t\theta) \end{bmatrix} \begin{bmatrix} \cos(\delta_j\theta) & -\sin(\delta_j\theta) \\ \sin(\delta_j\theta) & \cos(\delta_j\theta) \end{bmatrix} \begin{bmatrix} \sqrt{c} \\ 0 \end{bmatrix} \\ &= c\,[\cos(\Delta_t\theta)\cos(\delta_j\theta) + \sin(\Delta_t\theta)\sin(\delta_j\theta)] \\ &= c\cos\big((\delta_j - \Delta_t)\theta\big). \end{aligned} \tag{67}$$

Hence this score is maximized when

$$\delta_j = \Delta_t, \tag{68}$$

or equivalently,

$$r_j = p_t + \Delta(s_t). \tag{69}$$

For D RoPE blocks with frequencies $\theta_1, \dots, \theta_D$, choose

$$k^{(m)} = \begin{bmatrix} \sqrt{c_m} \\ 0 \end{bmatrix}, \qquad q^{(m)}(s_t) = R_{\Delta_t}^{(\theta_m)} k^{(m)}, \qquad c_m > 0. \tag{70}$$

The resulting score is

$$K_{\Delta_t}(\delta_j) = \sum_{m=1}^{D} c_m \cos\big((\delta_j - \Delta_t)\theta_m\big). \tag{71}$$

At the target position j^*, where $\delta_{j^*} = \Delta_t$, all cosine terms equal one:

$$K_{\Delta_t}(\Delta_t) = \sum_{m=1}^{D} c_m. \tag{72}$$

By the finite non-aliasing assumption, this is the unique maximum over the admissible displacement range. Since that range is finite, there exists a margin $\gamma > 0$ such that

$$s_{j^*} - s_j \geq \gamma, \qquad j \neq j^*. \tag{73}$$

Scaling the logits by $\beta > 0$ gives

$$a_{j^*} \geq \frac{1}{1 + (L-1)e^{-\beta\gamma}} \xrightarrow[\beta\to\infty]{} 1. \tag{74}$$

If the updated pointer is obtained from the attended physical positions,

$$p_{t+1} = \sum_{j=1}^{L} a_j r_j, \tag{75}$$

then

$$p_{t+1} \xrightarrow[\beta\to\infty]{} r_{j^*} = p_t + \Delta(s_t). \tag{76}$$

Finally, because both $\mathcal{S}$ and $\mathcal{D}$ are finite, the controller representation can select the corresponding query vector $q(s_t)$ through a finite-state lookup implemented by the controller projection. Thus, the pointer head implements the desired controller-conditioned relative shift to arbitrary accuracy. □

Lemma C.4 (Finite-range content-conditioned jump)**.** *Let the controller define an unrotated content score*

$$\bar{s}_j = q_\phi(s_t)^\top k_\phi(M_j). \tag{77}$$

Suppose this score uniquely selects a memory position j^* with margin $\gamma_\phi > 0$:

$$\bar{s}_{j^*} - \bar{s}_j \geq \gamma_\phi, \qquad j \neq j^*. \tag{78}$$

Assume that all relevant pointer-relative displacements satisfy

$$|r_j - p_t| \leq D. \tag{79}$$

Define

$$C_\phi = \|q_\phi(s_t)\|_2 \max_j \|k_\phi(M_j)\|_2 \tag{80}$$

and

$$\epsilon_\phi(D) = \sup_{|\delta| \leq D} \|R_\delta^\phi - I\|_2, \tag{81}$$

where R_δ^ϕ denotes the RoPE operator restricted to the subspace used by the content predicate. If

$$2C_\phi \epsilon_\phi(D) < \gamma_\phi, \tag{82}$$

then the pointer-relative attention score preserves j^* as its unique maximizer. Consequently, a pointer head can implement the content-conditioned jump

$$p_{t+1} = r_{j^*} \tag{83}$$

to arbitrary accuracy.

Proof. Let

$$\delta_j = r_j - p_t \tag{84}$$

be the displacement of memory position j from the current pointer. The pointer-relative score used by PoFormer is

$$s_j^\phi = q_\phi(s_t)^\top R_{\delta_j}^\phi \, k_\phi(M_j), \tag{85}$$

whereas the desired content-only score is

$$\bar{s}_j = q_\phi(s_t)^\top k_\phi(M_j). \tag{86}$$

To quantify the effect of the positional rotation, consider one two-dimensional RoPE block with frequency θ:

$$R_{\delta_j}^{(\theta)} = \begin{bmatrix} \cos(\delta_j\theta) & -\sin(\delta_j\theta) \\ \sin(\delta_j\theta) & \cos(\delta_j\theta) \end{bmatrix}. \tag{87}$$

The difference between the rotated and unrotated operators is

$$R_{\delta_j}^{(\theta)} - I = \begin{bmatrix} \cos(\delta_j\theta) - 1 & -\sin(\delta_j\theta) \\ \sin(\delta_j\theta) & \cos(\delta_j\theta) - 1 \end{bmatrix}, \tag{88}$$

whose spectral norm is

$$\left\| R_{\delta_j}^{(\theta)} - I \right\|_2 = 2 \left| \sin\left(\frac{\delta_j \theta}{2} \right) \right|. \tag{89}$$

Thus, a sufficiently low-frequency RoPE block behaves approximately as the identity over a bounded displacement range.

For the complete predicate subspace, the score perturbation satisfies

$$\begin{aligned} \left| s_j^\phi - \bar{s}_j \right| &= \left| q_\phi(s_t)^\top \left(R_{\delta_j}^\phi - I \right) k_\phi(M_j) \right| \\ &\le \|q_\phi(s_t)\|_2 \left\| R_{\delta_j}^\phi - I \right\|_2 \|k_\phi(M_j)\|_2 \\ &\le C_\phi \epsilon_\phi(D). \end{aligned} \tag{90}$$

Therefore, for every $j \neq j^*$,

$$\begin{aligned} s_{j^*}^\phi - s_j^\phi &= (\bar{s}_{j^*} - \bar{s}_j) + \left(s_{j^*}^\phi - \bar{s}_{j^*} \right) - \left(s_j^\phi - \bar{s}_j \right) \\ &\ge \gamma_\phi - \left| s_{j^*}^\phi - \bar{s}_{j^*} \right| - \left| s_j^\phi - \bar{s}_j \right| \\ &\ge \gamma_\phi - 2C_\phi \epsilon_\phi(D). \end{aligned} \tag{91}$$

By Condition 82, the remaining margin

$$\widetilde{\gamma}_\phi := \gamma_\phi - 2C_\phi \epsilon_\phi(D) \tag{92}$$

is strictly positive. Hence j^* remains the unique maximizer of the rotated attention score.

After scaling the logits by $\beta > 0$, the attention probability assigned to j^* satisfies

$$a_{j^*} \ge \frac{1}{1 + (L-1)e^{-\beta\widetilde{\gamma}_\phi}} \xrightarrow[\beta\to\infty]{} 1. \tag{93}$$

Under the expected-position pointer update,

$$p_{t+1} = \sum_{j=1}^{L} a_j r_j, \tag{94}$$

we consequently obtain

$$p_{t+1} \xrightarrow[\beta\to\infty]{} r_{j^*}. \tag{95}$$

Thus, the pointer head implements the desired content-conditioned jump to arbitrary accuracy. □

Finite validity range Suppose the predicate representation is supported on RoPE blocks indexed by $\mathcal{I}_\phi$, with frequencies $\{\theta_m\}_{m\in\mathcal{I}_\phi}$, and define

$$\Theta_\phi = \max_{m\in\mathcal{I}_\phi} |\theta_m|. \tag{96}$$

Because R_δ^ϕ is block diagonal,

$$\left\| R_\delta^\phi - I \right\|_2 = \max_{m\in\mathcal{I}_\phi} 2\left|\sin\left(\frac{\delta\theta_m}{2}\right)\right|. \tag{97}$$

When $D\Theta_\phi \le \pi$, this gives

$$\epsilon_\phi(D) \le 2\sin\left(\frac{D\Theta_\phi}{2}\right). \tag{98}$$

A sufficient condition for preserving the content-selected position is therefore

$$4C_\phi \sin\left(\frac{D\Theta_\phi}{2}\right) < \gamma_\phi. \tag{99}$$

The construction is consequently guaranteed only over a bounded displacement range. Using lower-frequency RoPE blocks enlarges this range, but no fixed periodic positional representation guarantees an arbitrarily large non-distorting range.

C.3 Finite-State Controller Updates

Lemma C.5 (Finite-state controller update)**.** *Let $\mathcal{S}$ and $\mathcal{X}$ be finite controller-state and readout spaces, with injective encodings*

$$e_\mathcal{S} : \mathcal{S} \to \mathbb{R}^{d_s}, \qquad e_\mathcal{X} : \mathcal{X} \to \mathbb{R}^{d_x}. \tag{100}$$

Suppose that the controller state and memory readout are represented in separable subspaces, so that the feed-forward module receives

$$u(s,x) = e_\mathcal{S}(s) \oplus e_\mathcal{X}(x). \tag{101}$$

If the feed-forward module belongs to an MLP family with the universal approximation property, then a sufficiently wide MLP can approximate any finite transition

$$U : \mathcal{S}\times\mathcal{X} \to \mathcal{S} \tag{102}$$

and any finite output-emission function to arbitrary accuracy.

Proof. Enumerate the finite set $\mathcal{S}\times\mathcal{X}$ as

$$(s_1,x_1),\dots,(s_N,x_N), \qquad N = |\mathcal{S}||\mathcal{X}|, \tag{103}$$

and let

$$u_i = e_\mathcal{S}(s_i) \oplus e_\mathcal{X}(x_i), \qquad v_i = e_\mathcal{S}\big(U(s_i,x_i)\big). \tag{104}$$

Because the encodings are injective, the joint representations $u_1,\dots,u_N$ are pairwise distinct.

For each i, define the polynomial

$$\psi_i(u) = \prod_{\substack{j=1\\ j\ne i}}^{N} \frac{\|u-u_j\|_2^2}{\|u_i-u_j\|_2^2}. \tag{105}$$

These functions satisfy

$$\psi_i(u_j) = \begin{cases} 1, & i=j,\\ 0, & i\ne j. \end{cases} \tag{106}$$

Consequently, the continuous function

$$F(u) = \sum_{i=1}^{N} v_i\psi_i(u) \tag{107}$$

interpolates the desired transition exactly on the finite set of valid representations:

$$F(u_i) = v_i = e_{\mathcal{S}}\big(U(s_i, x_i)\big). \tag{108}$$

Let $\mathcal{K}$ be a compact neighborhood containing all valid joint representations. By the universal approximation property, for every $\epsilon > 0$, there exists a sufficiently wide MLP F_θ such that

$$\sup_{u \in \mathcal{K}} \|F_\theta(u) - F(u)\|_2 < \epsilon. \tag{109}$$

In particular,

$$\big\|F_\theta(u_i) - e_{\mathcal{S}}\big(U(s_i, x_i)\big)\big\|_2 < \epsilon \tag{110}$$

for every valid state–readout pair.

For a residual Transformer block, define the desired correction at u_i as

$$d_i = e_{\mathcal{S}}\big(U(s_i, x_i)\big) - e_{\mathcal{S}}(s_i). \tag{111}$$

Applying the same interpolation argument to the finite mapping $u_i \mapsto d_i$ yields an MLP whose residual update satisfies

$$e_{\mathcal{S}}(s_i) + \mathrm{MLP}_\theta(u_i) \approx e_{\mathcal{S}}\big(U(s_i, x_i)\big) \tag{112}$$

to arbitrary accuracy.

Since $\mathcal{S}$ is finite and its encodings are distinct, their minimum separation

$$\rho_{\mathcal{S}} = \min_{\substack{s, s' \in \mathcal{S} \\ s \neq s'}} \|e_{\mathcal{S}}(s) - e_{\mathcal{S}}(s')\|_2 \tag{113}$$

is strictly positive. Choosing $\epsilon < \rho_{\mathcal{S}}/2$ ensures that the approximate next-state representation remains closer to the correct state encoding than to any other state encoding.

The same construction applies to a finite emission function by replacing v_i with a target logit vector whose unique maximum corresponds to the required output symbol. Scaling the target logits makes the resulting softmax probability arbitrarily close to one. □

C.4 Proof of the Simulation Theorem

Proof of Theorem 4.4. The preprocessing stage constructs the static memory using standard C-RASP (Yang & Chiang, 2024) operations. Lemma C.2 realizes address resolution and content readout. Lemma C.3 realizes local forward, backward, stay, and controller-conditioned transitions. Lemma C.4 realizes unique content-conditioned jumps within the valid displacement range. Lemma C.5 realizes finite-state controller transitions and output emissions.

Because a P-RASP program contains only a finite composition of these operations, PoFormer can allocate attention heads or representational subspaces to the required reads, shifts, jumps, and controller updates. Sufficient layer depth composes these operations within one recurrent step, while recurrent parameter sharing applies the same transition across iterations. For any fixed finite execution range and prescribed $\varepsilon > 0$, the attention logits can be scaled by a sufficiently large fixed inverse temperature so that each softmax selection approximates its corresponding hard selection within the required accuracy. Composing the resulting approximations therefore yields a PoFormer that approximates the controller states, pointer trajectories, and output emissions of the P-RASP program to the accuracy stated in Theorem 4.4. □

D P-RASP Programs for Algorithmic Tasks

We give explicit P-RASP programs for the tasks in Corollary 4.5. Each program uses a constant number of controller states and pointers, and its recurrent transition does not depend on the input length.

D.1 Duplicate-Token Copy

Consider an input memory of the form

$$M = (\texttt{<bos>}, x_1, \ldots, x_n, \texttt{<sep>}),$$

together with a controller token initialized as <bot>. The target output is $(x_1, \ldots, x_n, \texttt{<eos>})$.

Proposition D.1 (Duplicate-Token Copy). *There exists a* P-RASP *program that solves Duplicate-Token Copy in $n + 1 = O(L)$ recurrent iterations.*

```
def update(s, x):
    if x == SEP:
        # Enter the termination state.
        return EOS
    else:
        # Store the currently read symbol.
        return x


def _layer1(M, s, p):
    if s == BOS:
        # Move to the first content token.
        p1 = p_shift(p, +1)
    else:
        # Keep the address from layer 0.
        p1 = p_shift(p, 0)

    x1 = read(M, p1)
    s1 = update(s, x1)
    emit(s1)

    if s1 == EOS:
        halt()

    return s1, p1
```

```
def _layer0(M, s, p):
    if s == BOT:
        # Initialize at the left boundary.
        p0 = p_jump(M, query=BOS)
    else:
        # Advance by a constant offset.
        p0 = p_shift(p, +1)

    x0 = read(M, p0)
    s0 = update(s, x0)
    return s0, p0

def duplicate_token_copy(M):
    s = BOT
    p = bot_position(M)

    while s != EOS:
        s, p = _layer0(M, s, p)
        s, p = _layer1(M, s, p)

    return
```

Listing 1: A P-RASP program for Duplicate-Token Copy. The first layer initializes or advances the pointer, and the second layer reads and emits the selected memory value.

D.2 Prefix Parity

For $x_1, \ldots, x_n \in \{0, 1\}$, define

$$q_i = \bigoplus_{j=1}^{i} x_j.$$

The target output is $(q_1, \ldots, q_n, \texttt{<eos>})$.

Proposition D.2 (Prefix Parity). *There exists a* P-RASP *program that solves Prefix Parity in $n + 1 = O(L)$ recurrent iterations.*

D.3 Shuffled Linked-List Traversal

The input contains a shuffled collection of records of the form

$$k_i \;\rightarrow\; v_i,$$

where every key k_i is unique and v_i is the key of its logical successor. The distinguished record $\texttt{<begin>} \rightarrow k_{\text{start}}$ identifies the first node, and the final node points to <end>. A schematic input sequence is shown below:

<bos>	<begin>	→	x_1	;	x_2	→	x_3	;	...	x_n	→	<end>	<sep>	<bot>

In the flattened memory representation used below, the value of each record is stored at a fixed offset of $+2$ from its key.

Proposition D.3 (Shuffled Linked-List Traversal). *There exists a* P-RASP *program that traverses a shuffled linked list containing n nodes in $n + 1 = O(L)$ recurrent iterations.*

```
def update(s, x):
    if s == BOT:
        # Enter the initialization phase.
        return BOS

    if s == BOS:
        # Initialize parity with x_1.
        return x

    if x == SEP:
        # Enter the termination state.
        return EOS

    # Accumulate the running parity.
    return xor(s, x)

def _layer1(M, s, p):
    if s == BOS:
        # Move to the first input bit.
        p1 = p_shift(p, +1)
    else:
        # Keep the address from layer 0.
        p1 = p_shift(p, 0)

    x1 = read(M, p1)
    s1 = update(s, x1)
    emit(s1)

    if s1 == EOS:
        halt()

    return s1, p1
```

```
def _layer0(M, s, p):
    if s == BOT:
        # Initialize at the left boundary.
        p0 = p_jump(M, query=BOS)
        x0 = read(M, p0)
        s0 = update(s, x0)
    else:
        # Advance to the next memory cell.
        s0 = s
        p0 = p_shift(p, +1)

    return s0, p0

def prefix_parity(M):
    s = BOT
    p = bot_position(M)

    while s != EOS:
        s, p = _layer0(M, s, p)
        s, p = _layer1(M, s, p)

    return
```

Listing 2: A P-RASP program for Prefix Parity. The pointer performs a sequential scan, while the finite controller maintains the running parity.

E Experimental Details

E.1 Task Definition and Configuration

We provide the complete definitions and data-generation settings for the six basic algorithmic tasks. Unless otherwise stated, an input consists of a static memory prefix

$$M = (\texttt{<bos>}, x_1, \ldots, x_n, \texttt{<sep>})$$

followed by the recurrent controller token <bot>. The controller token is not included in the target sequence, and every output terminates with <eos>. Table 2 summarizes the task-specific generation settings.

Duplicate-Token Copy Duplicate-Token Copy is adapted from the Copy with Repeating Tokens setting (Zhou et al., 2024a; Huang et al., 2025; Fan et al., 2025) and tests stable left-to-right traversal. Its input and target are

<bos>	x_1	x_2	$\cdots$	x_n	<sep>	<bot>	$\longrightarrow$	x_1	x_2	$\cdots$	x_n	<eos>

Content tokens are sampled independently from a finite vocabulary consisting of the 26 lowercase alphabet symbols $\{a, \ldots, z\}$ together with a set of special control tokens. For each generated sequence, content tokens are sampled independently and uniformly from the alphabet subset. To control the ambiguity of content-based retrieval, we introduce a repetition rate of 0.5: with probability 0.5, a newly sampled content token is copied from a previously generated position, while with probability 0.5, it is sampled as a new symbol from the alphabet. This procedure creates sequences where identical symbols may appear at multiple memory locations while preserving a controllable amount of redundancy.

Reverse Reverse uses the same input distribution as Duplicate-Token Copy but requires right-to-left traversal:

<bos>	x_1	x_2	$\cdots$	x_n	<sep>	<bot>	$\longrightarrow$	x_n	x_{n-1}	$\cdots$	x_1	<eos>

```python
def update(s, x):
    if x == BEGIN:
        # Enter the traversal phase.
        return BEGIN

    if x == END:
        # Enter the termination state.
        return EOS

    # Store the successor key.
    return x

def _layer1(M, s, p):
    if s == BEGIN:
        # Preserve the initialization address.
        p1 = p_shift(p, 0)
    else:
        # Select the record whose key is s.
        p1 = p_jump(
            M,
            query=s,
            predicate=is_key
        )

    x1 = read(M, p1)
    s1 = update(s, x1)
    return s1, p1

def shuffled_linked_list(M):
    s = BOT
    p = bot_position(M)

    while s != EOS:
        s, p = _layer0(M, s, p)
        s, p = _layer1(M, s, p)
        s, p = _layer2(M, s, p)

    return
```

```python
def _layer0(M, s, p):
    if s == BOT:
        # Find the distinguished start key.
        p0 = p_jump(M, query=BEGIN)
    else:
        # Preserve the current address.
        p0 = p_shift(p, 0)

    x0 = read(M, p0)
    s0 = update(s, x0)
    return s0, p0

def _layer2(M, s, p):
    # Move from a key to its value field.
    p2 = p_shift(p, +2)

    # Read the logical successor key.
    x2 = read(M, p2)
    s2 = update(s, x2)
    emit(s2)

    if s2 == EOS:
        halt()

    return s2, p2
```

Listing 3: A P-RASP program for Shuffled Linked-List Traversal. Each iteration jumps to the record matching the current key and then shifts to its successor field.

The input again has repetition rate 0.5, so the model must distinguish different occurrences of the same symbol. In contrast to Duplicate-Token Copy, the required pointer transition moves backward through memory, testing whether the architecture can learn traversal against the causal left-to-right direction. Standard Transformers typically exhibit lower performance on Reverse than on Copy (Kazemnejad et al., 2023).

Prefix Parity Prefix Parity combines forward traversal with recurrent finite-state accumulation. For a binary input sequence, the task is

<bos>	x_1	x_2	$\cdots$	x_n	<sep>	<bot>	$\longrightarrow$	s_1	s_2	$\cdots$	s_n	<eos>

Here $x_i, s_i \in \{0, 1\}$, and each target token is the parity of the corresponding input prefix:

$$s_i = \bigoplus_{j=1}^{i} x_j. \tag{114}$$

The pointer follows the same left-to-right trajectory as in Duplicate-Token Copy, while the controller maintains and updates a single parity bit.

Flip-Flop Flip-Flop tests persistent finite-state updates during sequential traversal. Each input token is an operation from

$$\mathcal{X}_{\text{ff}} = \{\texttt{<ignore>}, \texttt{<read>}, \texttt{<write_0>}, \texttt{<write_1>}\}.$$

The input and densely supervised target have the form

<bos>	x_1	x_2	$\cdots$	x_n	<sep>	<bot>	$\longrightarrow$	s_1	s_2	$\cdots$	s_n	<eos>

Starting from a fixed initial binary state s_0, the transition is

$$s_i = \begin{cases} 0, & x_i = \texttt{<write_0>}, \\ 1, & x_i = \texttt{<write_1>}, \\ s_{i-1}, & x_i \in \{\texttt{<ignore>}, \texttt{<read>}\}. \end{cases} \tag{115}$$

The target symbol at every step represents the resulting state s_i. Thus, the task separates stable positional traversal from command-conditioned state overwriting and persistence.

Unique Set Unique Set requires the model to identify the first occurrence of each symbol while preserving the original sequence length and order:

<bos>	x_1	x_2	$\cdots$	x_n	<sep>	<bot>	$\longrightarrow$	y_1	y_2	$\cdots$	y_n	<eos>

The output at position i is

$$y_i = \begin{cases} x_i, & x_i \notin \{x_1, \ldots, x_{i-1}\}, \\ \texttt{<pad>}, & \text{otherwise.} \end{cases} \tag{116}$$

For example,

a	b	a	c	b	$\longrightarrow$	a	b	<pad>	c	<pad>

We use repetition rate 0.25. Solving the task requires combining left-to-right traversal with a content-based test of whether the currently addressed symbol has appeared earlier.

Shuffled Linked-List Traversal Shuffled Linked-List Traversal evaluates content-conditioned, non-local address transitions. The memory contains records of the form $u \to v$, where every key u is unique and v identifies its logical successor. The records are randomly permuted, so logical neighbors are generally non-adjacent in physical memory. A schematic example is

<bos>	<begin>	$\to$	i_1	;	$i_{\pi(1)}$	$\to$	next($i_{\pi(1)}$)	;	$\cdots$	$i_{\pi(n)}$	$\to$	next($i_{\pi(n)}$)	<sep>	<bot>

Here π is a random permutation of the node records, $\mathrm{next}(i_n) = \texttt{<end>}$, and the target is

i_1	i_2	$\cdots$	i_n	<eos>

At each step, the model must read the successor identifier associated with the current node and jump to the key position of the matching record. Because a node identifier may occur both as a record key and as another record's value, selection must depend jointly on token identity and structural role.

E.2 Model and Optimization

We compare PoFormer against five baseline settings. All models use decoder-only Transformer architectures with RMSNorm pre-normalization, SwiGLU feed-forward networks. Unless otherwise specified, all models are trained with next-token prediction (NTP).

Table 2: Data-generation and evaluation configurations for the basic and advanced tasks. Vocabulary sizes include task-specific special tokens.

Task	Train Length	Evaluation Length	Vocabulary	Repetition Rate
Duplicate-Token Copy	2–60	2–200	31	0.50
Reverse	2–60	2–200	31	0.50
Parity	2–60	2–120	7	–
Flip-Flop	2–60	2–120	11	–
Unique Set	2–60	2–120	105	0.25
Shuffled Linked-List	2–30	2–60	265	–
Bidirectional Copy	2–60	2–200	33	0.25
Binary Addition	1–20	2–40	8	–

- **NoPE NTP Transformer** A standard decoder-only Transformer trained with next-token prediction and no explicit positional encoding. This baseline evaluates how far purely content-based attention can solve the considered algorithmic tasks without an external positional prior.
- **APE NTP Transformer** A standard decoder-only Transformer trained with next-token prediction and learned absolute positional embeddings. In both training and evaluation, we allocate positional embeddings up to the maximum sequence length used in the experiments. To strengthen the length-generalization ability of this baseline, we additionally apply a random positional-offset strategy (Huang et al., 2025) during training: each training example is assigned a randomly sampled offset before its absolute positions are looked up.
- **RoPE NTP Transformer** (Su et al., 2024) A standard decoder-only Transformer trained with next-token prediction and Rotary Positional Embeddings. The RoPE configuration, including the base frequency, is matched to the corresponding PoFormer model. This baseline evaluates whether improved length generalization can be explained solely by the use of rotary position encoding.
- **ALiBi NTP Transformer** (Press et al., 2022) A standard decoder-only Transformer trained with next-token prediction and ALiBi attention biases. This baseline introduces a monotonic distance-dependent inductive bias directly into attention logits and provides an alternative extrapolatable positional mechanism.
- **RoPE Looped Transformer** A recurrent weight-sharing Transformer using the same recurrent decoding procedure as PoFormer and RoPE-based positional initialization. The only difference is that it does not maintain an explicit address channel or pointer state during execution. This ablation isolates the contribution of explicit pointer-based latent addressing from the effect of recurrent decoding alone.

Optimization and Evaluation Protocol All models use AdamW optimization with weight decay 0.01, batch size 64, and gradient clipping with maximum norm 1.0. Unless otherwise specified, models are trained with a constant learning rate. The learning rate and total number of optimization steps are selected according to model and task difficulty.

For each sequence length, we generate a training bucket containing 4,096 examples and an evaluation bucket containing 512 examples. Curriculum schedules are described in Appendix E.2.1. During training, we evaluate models every 200 optimization steps on fixed validation sets spanning all sequence lengths in the current training range. We record aggregate validation accuracy and loss, and retain the best checkpoint achieving the highest aggregate validation accuracy, using validation loss as a secondary selection criterion.

APE and RoPE use the same random positional-offset augmentation during training. RoPE-based models use $\theta_{\text{base}} = 10^6$ and $d_{\text{rot}} = \min(128, d_{\text{head}})$ unless otherwise specified. ALiBi uses head-specific linear attention biases.

For PoFormer, pointer supervision is **NOT** provided for any basic task. The model learns latent pointer dynamics solely through the task objective. During training, pointer updates use a differentiable expected-position update, whereas during evaluation PoFormer uses a pointer-attention temperature of 0.75 and replaces the expected-position update with an argmax address update. This difference is intentional: the differentiable update facilitates gradient-based optimization, while the evaluation-time discrete address update aligns the learned pointer dynamics with the hard-selection semantics assumed by P-RASP.

To clarify the effect of this train–test difference, we use Duplicate-Token Copy and Reverse as a representative example and compare the evaluation-time argmax update with the train–test-matched expected-position update. As shown in Fig. 6, the train–test-matched soft configuration already generalizes beyond the maximum training length, showing that the learned recurrent pointer dynamics themselves acquire an extrapolatable relative-address transition. Its accuracy nevertheless degrades at longer execution horizons, consistent with the accumulation of fractional pointer errors under repeated soft updates. In contrast, evaluation-time argmax addressing substantially extends the reliable extrapolation range by projecting each update onto a discrete physical address. We therefore view discretization as a robustness mechanism for long-horizon execution rather than as the source

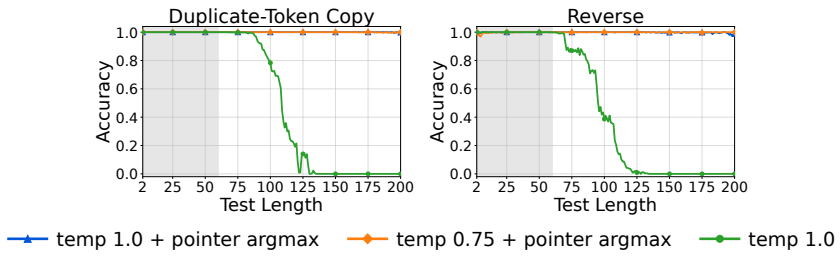

Figure 6: Effect of pointer-update discretization on Duplicate-Token Copy and Reverse. The shaded region denotes the training-length range. In the legend, `pointer argmax` indicates that the pointer is updated to the physical position with the maximum attention weight, while `temp` specifies the softmax temperature used to compute the pointer-attention distribution. The train–test-matched expected-position configuration (`temp 1.0`, without `pointer argmax`) already generalizes beyond the training range, while evaluation-time argmax addressing substantially extends the reliable extrapolation horizon, consistent with discretization preventing the accumulation of fractional pointer errors.

of the learned traversal rule. This execution regime also more closely matches the discrete address transitions characterized by P-RASP.

$$\underbrace{p_{t+1} = \sum_i a_i i}_{\text{Training: differentiable expected-position update}} \longrightarrow \underbrace{p_{t+1} = \arg\max_i a_i}_{\text{Evaluation: discrete hardmax address update}}$$

Table 3: Task-specific model configurations. All baseline models use the same backbone scale as PoFormer within each task; differences are limited to the attention mechanism and explicit pointer-address components. These algorithmic tasks are difficult for baselines to learn with this small scale.

Task	Layers	Heads	d_{model}	d_{ff}	Pointer Heads	Slave Heads
Duplicate-Token Copy	2	2	256	512	1	1
Reverse	2	2	256	512	1	1
Parity	2	2	256	512	1	1
Flip-Flop	2	2	256	512	1	1
Unique Set	2	2	256	512	1	1
Shuffled Linked List	3	2	256	512	1	1
Bidirectional Copy	2	2	256	512	1	1
Binary Addition	2	4	512	1024	2	2

E.2.1 Curriculum Schedule

We use a two-stage length curriculum that separates task acquisition from length expansion. In the first stage, models are trained on short sequences, typically of lengths 2–10 or 2–20, using a relatively large task-dependent learning rate usually between 1×10^{-3} and 3×10^{-4}. This stage facilitates learning the controller's finite-state transition rules before optimization is dominated by the more difficult attention patterns required over long sequences. Empirically, training directly on the full length range often entangles state-transition learning with unstable addressing, making both components more difficult to acquire.

Within each curriculum epoch, the model visits length buckets from short to long, moving to the next length after processing the prescribed samples for the current bucket. Importantly, we do not train each individual length to convergence before advancing, as doing so can encourage length-specific overfitting. Instead, over multiple epochs, we either repeat the entire short-length curriculum or gradually expand its maximum training length until accuracy converges across all included lengths. We also avoid continuing this stage substantially beyond convergence.

After the model has acquired the task on short inputs, the second stage expands the curriculum to the complete training-length range and reduces the learning rate to a task-dependent value usually between 1×10^{-4} and 1×10^{-5}. The model again cycles through the length buckets from short to long and typically converges across the full training range within only one or two epochs. The lower learning rate allows the learned state-transition rules to be retained while the attention and pointer dynamics adapt to longer execution horizons.

The changing length distribution produces characteristic optimization dynamics. During the first stage, the loss may oscillate as training repeatedly moves between length buckets of different difficulty. After several curriculum epochs, models that discover the appropriate addressing rule often exhibit a sharp transition to near-perfect accuracy across the short-length range. During the second stage, the loss usually remains low, with occasional increases when longer lengths are introduced. Such fluctuations are expected provided that the loss remains bounded and accuracy subsequently recovers across the expanded length range.

$$\underbrace{\text{short lengths + larger LR}}_{\text{state-transition acquisition}} \longrightarrow \underbrace{\text{full range + smaller LR}}_{\text{address stabilization}}$$

E.3 Controlling for Baseline In-Distribution Fit

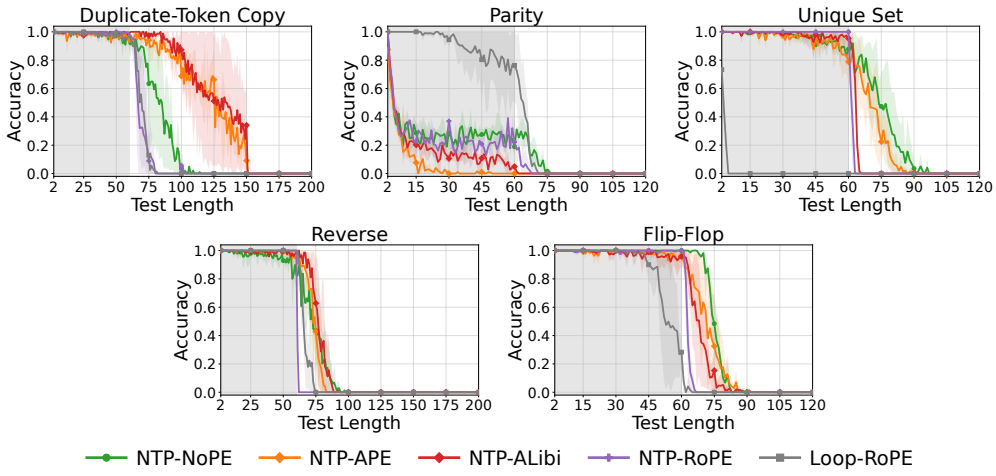

Figure 7: Length generalization of the scaled baseline models. Curves show mean exact-match accuracy ± SD over three seeds; the background shading marks the training-length range. The scaled baselines obtain strong ID performance on all displayed tasks except Prefix Parity, but their accuracy generally declines rapidly beyond the training range.

To determine whether the weak length generalization of the baselines is merely a consequence of limited model capacity or incomplete in-distribution (ID) learning, we repeat the experiments with substantially enlarged baseline models. We omit Shuffled Linked-List Traversal because the compact baselines already achieve strong ID performance on this task. For Duplicate-Token Copy, Parity, Unique Set, Reverse, and Flip-Flop, we increase each baseline to four layers and four attention heads. These scaled models achieve strong performance throughout the training-length range, but their exact-match accuracy still declines rapidly once the input length exceeds the maximum training length (Figure 7). APE and ALiBi exhibit better OOD generalization on Duplicate-Token Copy, but remain substantially less robust than PoFormer over longer extrapolation ranges.

Prefix Parity is an exception: even after scaling the baselines, they remain difficult to optimize to consistently strong ID performance. Overall, the remaining tasks show that insufficient capacity or incomplete ID fitting is not a sufficient explanation for the baseline OOD failures: conventional architectures can learn the training distribution well while still failing to preserve the learned computation beyond it.

E.4 Advanced-Task and Pointer-Supervision Details

We provide the complete definitions and pointer-supervision trajectories for Bidirectional Copy and Binary Addition. In preliminary experiments, we found that output supervision alone was insufficient for learning the intended addressing policies. We therefore construct a P-RASP program for each task and translate its symbolic execution trace into target pointer positions.

Bidirectional Copy The input consists of a content sequence followed by a direction command:

<bos>	x_1	x_2	$\cdots$	x_n	<sep>	<fwd|bwd>

The command token belongs to $\{\texttt{<fwd>}, \texttt{<bwd>}\}$ and simultaneously serves as the recurrent controller token. The target is

$$y^* = \begin{cases} (x_1, x_2, \ldots, x_n, \texttt{<eos>}), & c = \texttt{<fwd>}, \\ (x_n, x_{n-1}, \ldots, x_1, \texttt{<eos>}), & c = \texttt{<bwd>}. \end{cases} \tag{117}$$

We use a two-layer, two-head PoFormer containing one pointer head and its corresponding slave head. Hence, the same pointer head must represent both transitions

$$p_{t+1} = p_t + 1 \quad \text{and} \quad p_{t+1} = p_t - 1 \tag{118}$$

within shared query–key projections and select between them according to the controller command. The model performs $T = n + 1$ decoding steps, including the final <eos> prediction.

Index <bos> by 0, the content tokens by $1, \ldots, n$, and <sep> by $n + 1$. At the final-layer read checkpoints, the P-RASP program produces the forward targets

$$j_t^{*,\mathrm{fwd}} = \begin{cases} t, & 1 \le t \le n, \\ n+1, & t = n+1, \end{cases} \tag{119}$$

and the backward targets

$$j_t^{*,\mathrm{bwd}} = \begin{cases} n - t + 1, & 1 \le t \le n, \\ 0, & t = n+1. \end{cases} \tag{120}$$

Thus, the same supervised pointer head follows opposite trajectories depending on the input command, terminating at <sep> in the forward case and <bos> in the backward case.

Binary Addition The input contains two n-bit binary numbers in most-significant-bit-first order:

<bos>	x_1	x_2	$\cdots$	x_n	+	y_1	y_2	$\cdots$	y_n	<sep>	<bot>

The output contains $n + 1$ sum bits in least-significant-bit-first order, followed by <eos>:

$$y^* = (z_1, z_2, \ldots, z_{n+1}, \texttt{<eos>}). \tag{121}$$

The additional bit represents a possible final carry. The model therefore performs $T = n+2$ decoding steps.

We use a two-layer, four-head PoFormer containing two pointer heads and their corresponding slave heads. Pointer $p_t^{(x)}$ addresses the current bit of the first operand, pointer $p_t^{(y)}$ addresses the corresponding bit of the second operand, and the recurrent controller stores the carry c_t. During the arithmetic steps,

$$p_{t+1}^{(x)} = p_t^{(x)} - 1, \qquad p_{t+1}^{(y)} = p_t^{(y)} - 1. \tag{122}$$

Index <bos> by 0, x_i by i, the separator + by $n + 1$, and y_i by $n + 1 + i$. At the operand-read checkpoints, the logical P-RASP targets for $1 \le t \le n$ are

$$j_t^{(x),*} = n - t + 1, \qquad j_t^{(y),*} = 2n - t + 2. \tag{123}$$

Thus, the two pointers begin at x_n and y_n and move synchronously toward x_1 and y_1. The P-RASP trace additionally specifies the layer-specific initialization and boundary positions used for the final carry and termination steps.

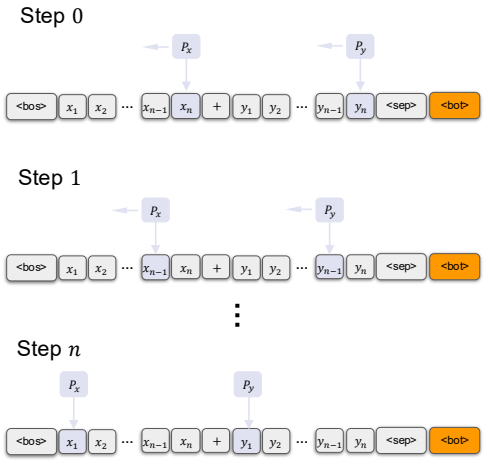

Figure 8: Dual-pointer execution of Binary Addition. The two pointers scan the operands synchronously from right to left, while the recurrent controller maintains the carry and emits the sum in reverse digit order.

Pointer-supervision objective Let $a_{t,j}^{(\ell,h)}$ denote the attention probability assigned to position j by pointer head h in layer ℓ at recurrent step t, and let $j_t^{(\ell,h),*}$ be the corresponding target obtained from the P-RASP trace. The supervised tuples form the set $\mathcal{I}_{\text{ptr}}$, and the auxiliary objective is

$$\mathcal{L}_{\text{ptr}} = -\frac{1}{|\mathcal{I}_{\text{ptr}}|} \sum_{(t,\ell,h)\in\mathcal{I}_{\text{ptr}}} \log a_{t,j_t^{(\ell,h),*}}^{(\ell,h)}. \tag{124}$$

The full objective is

$$\mathcal{L} = \mathcal{L}_{\text{task}} + \lambda_{\text{ptr}}\mathcal{L}_{\text{ptr}}. \tag{125}$$

For both advanced tasks, we use $\lambda_{\text{ptr}} = 0.3$. Pointer supervision is applied exclusively to pointer heads and is imposed at every layer and every recurrent step, i.e., $\mathcal{I}_{\text{ptr}}$ contains all (t, ℓ, h) for which h is a pointer head. For Bidirectional Copy, the target trajectory is conditioned on the input command. With <fwd>, the pointer follows the same left-to-right trajectory as in Duplicate-Token Copy; with <bwd>, it follows the same right-to-left trajectory as in Reverse. For Binary Addition, the two pointer heads are assigned to the first and second operands, respectively, with layer-specific supervision. At recurrent step $t = 0$, the layer-0 pointers target the two separator positions + and <sep>; the corresponding layer-1 targets are one position to the left, namely the least-significant bits x_n and y_n. For every subsequent recurrent step, the layer-0 targets shift one position to the left, while the layer-1 targets retain the addresses produced by layer 0. Thus, the auxiliary loss directly supervises only address selection. Output-token prediction, carry computation, controller-state transitions, and termination remain governed solely by $\mathcal{L}_{\text{task}}$.

E.5 Additional Attention Maps and Failure Cases

Learned pointer trajectories To determine whether PoFormer solves the traversal tasks through the intended address dynamics, we inspect its pointer-attention maps. Figure 9 shows both layers of the trained models on Duplicate-Token Copy and Reverse. These tasks isolate the two directions of local traversal: Duplicate-Token Copy requires a left-to-right scan, whereas Reverse requires a right-to-left scan.

The resulting attention maps exhibit clean diagonal trajectories. For Duplicate-Token Copy, the active address advances in the natural input order, corresponding to the transition

$$p_{t+1} = p_t + 1. \tag{126}$$

Reverse exhibits the opposite diagonal, corresponding to

$$p_{t+1} = p_t - 1. \tag{127}$$

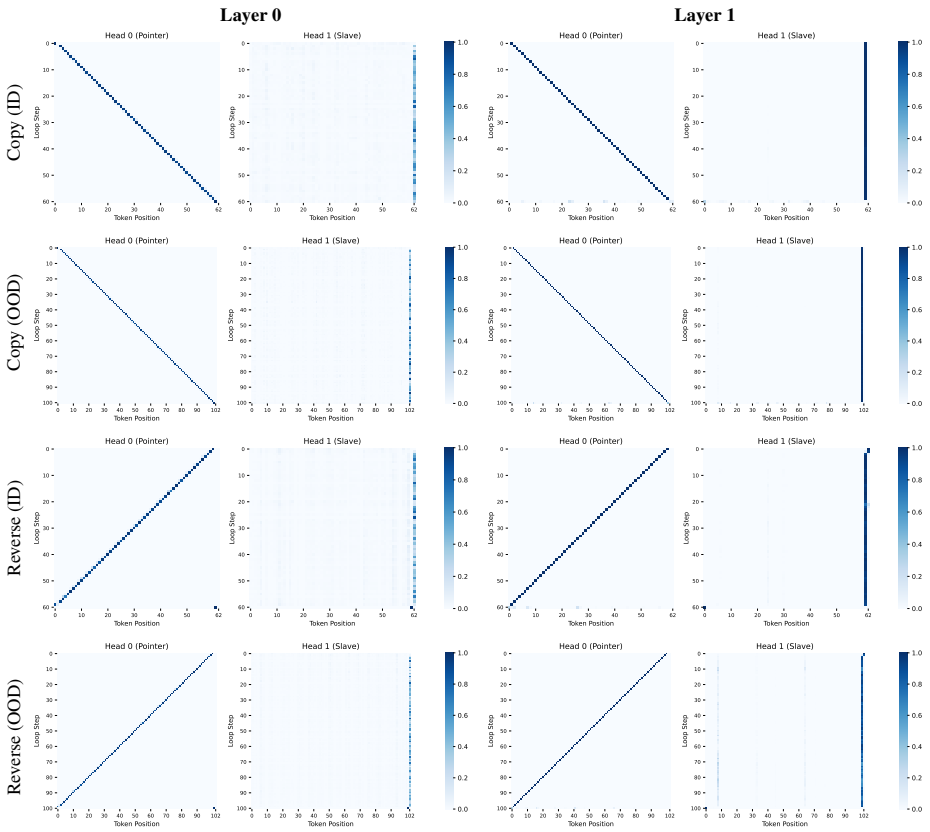

Figure 9: Layer-wise pointer-attention maps for Duplicate-Token Copy and Reverse. ID examples have length $L = 60$, while OOD examples have length $L = 100$. The forward and backward diagonal trajectories remain stable outside the training-length range.

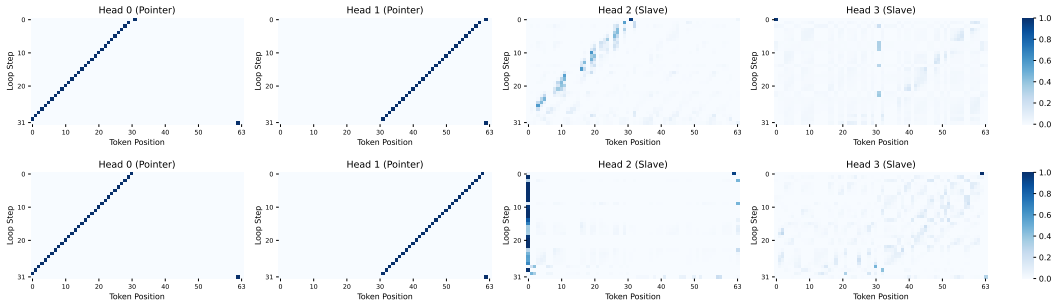

Figure 10: Pointer-attention maps for Binary Addition at $L = 30$. The two pointer-head trajectories correspond to the two operands and move synchronously from right to left. Top: Layer 0; bottom: Layer 1.

Importantly, these structures remain visible at the OOD length $L = 100$. Their persistence beyond the training range supports the P-RASP interpretation in Section 4.1: PoFormer learns an extrapolatable relative address transition rather than a lookup rule tied to the input lengths observed during training.

Multi-pointer behavior on Binary Addition Figure 10 shows the pointer trajectories learned with auxiliary supervision. The two pointer channels track the corresponding operands from right to left across both layers, consistent with the symbolic P-RASP trace. Meanwhile, the controller combines the retrieved bits with its recurrent carry state to produce the output sequence.

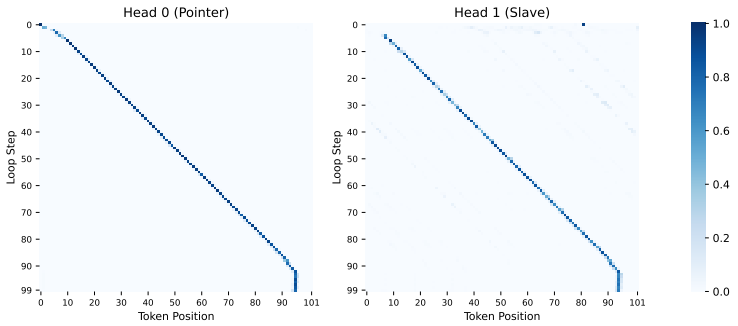

Figure 11: A representative OOD failure on Prefix Parity at $L = 100$. The pointer trajectory remains globally diagonal, but local boundary misalignment produces an incorrect read and subsequently corrupts the recurrent parity state.

OOD address aliasing The remaining failures are primarily address-level errors rather than a complete loss of the learned traversal rule. Figure 11 shows a representative OOD **Prefix Parity** example. Although the pointer trajectory remains globally diagonal, it becomes locally misaligned near the beginning and end of the recurrent execution. At unseen lengths, RoPE phases outside the training range can perturb the learned positional geometry and split attention between nearby or aliased addresses.

Such local errors are particularly harmful for state-sensitive computations. In Prefix Parity, one skipped or duplicated read changes the controller state and can therefore corrupt every subsequent output, even though the remaining pointer trajectory is approximately correct. This explains why output accuracy can deteriorate faster than the global attention geometry.

Accumulation of local output errors **Unique Set** exhibits a related but less state-sensitive failure mode. PoFormer typically identifies the correct unique elements, but occasional duplicate decisions or ordering errors invalidate the complete sequence under exact match. This is reflected by the large gap between its average OOD exact-match accuracy (45.1%), step accuracy (97.3%), and bag-level F1 (98.8%). At $L = 120$, exact match is nearly zero while step accuracy and bag-level F1 remain 92.0% and 95.6%, respectively. Thus, the model continues to recover most target elements, but the strict sequence-level metric amplifies sparse local errors as the execution horizon grows.

E.6 Language Modeling Details

Both 0.6B models use a GPT-2-style decoder-only architecture with 28 layers, 16 attention heads per layer, hidden dimension 1,024, and feed-forward dimension 4,096. In PoFormer, four of the 16 attention heads in each layer are designated as pointer heads, while the remaining 12 retain standard attention; the Standard LM uses conventional attention for all 16 heads. Both models are pretrained from scratch on FineWeb-Edu (Penedo et al., 2024) using the standard next-token-prediction objective. Their pretraining hyperparameters are summarized in Table 4. Pretraining required a total of 387.6 80GB A100 GPU-hours. Figure 12 shows that the Standard LM and PoFormer follow nearly identical optimization trajectories throughout pretraining, consistent with their similar validation perplexities of 11.69 and 11.72, respectively.

Downstream evaluation follows the same instruction-tuning and task-specific fine-tuning pipeline for both architectures. RULER QA-2 uses a maximum sequence length of 4,096, while SCROLLS Qasper and QuALITY use maximum lengths of 8,192 and 16,384, respectively.

We apply the same two-stage fine-tuning protocol to both architectures. Each model is first instruction-tuned for three epochs on *databricks-dolly-15k* (Conover et al., 2023) and subsequently fine-tuned for three epochs on 2,000 training examples from each downstream benchmark. In both stages, the objective is computed only over target response tokens.

All fine-tuning runs use AdamW with $\beta_1 = 0.9$, $\beta_2 = 0.95$, learning rate 10^{-5}, weight decay 0.01, and an effective batch size of 16. The learning rate is linearly warmed up over the first 5% of

Table 4: Pretraining configuration shared by the Standard LM and PoFormer.

Hyperparameter	**Value**
Sequence length	2,048
Micro-batch size	8 sequences
Global batch size	524,288 tokens
Peak learning rate	3×10^{-4}
Minimum learning rate	1×10^{-4}
Warmup fraction	2%
Plateau fraction	83%
RoPE base	10^6

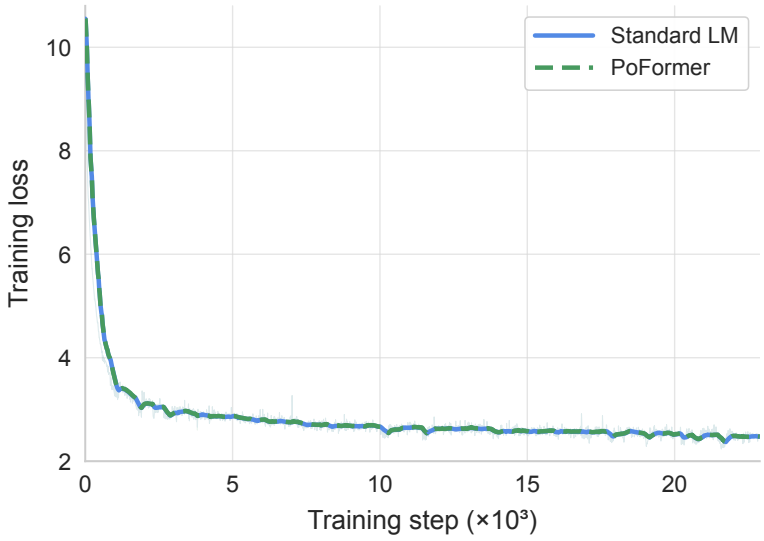

Figure 12: Pretraining loss of the matched 0.6B Standard LM and PoFormer. Thick curves show an exponential moving average over 120 logged points; faint curves show the raw measurements.

optimization steps and then decayed according to a cosine schedule. We clip the gradient norm at 1.0 and use BF16 mixed-precision training with gradient checkpointing. At evaluation time, responses are generated using greedy decoding with a maximum generation length of 64 tokens. The Standard LM and PoFormer use identical data splits, optimization settings, and decoding configurations.