An empirical study on the limitation of Transformers in program trace generation

Simeng Sun
simengs@nvidia.com

Abstract

We study Transformers on the task *program trace generation* (PTG), where models produce step-by-step execution traces for synthetic programs. Unlike existing algorithmic problems, PTG externalizes reasoning through long traces where each step is trivial. We train small Transformers with diverse modifications, including alternative position encodings, softmax replacements, hybrid model, and short convolutions. While these models achieve strong in-distribution accuracy, they exhibit systematic failures when generalizing to various factors (e.g., program length, trace steps), though some designs significantly improve generalization.

1 Introduction

We study Transformers on the task *program trace generation* (PTG): given a synthetic program and inputs, models must output the complete step-by-step execution trace. This setup offloads the *internal* reasoning (e.g., sorting, tracking permutations) onto long, structured traces/scratchpads that explicitly store the states of intermediate steps. Following prior work (Sun et al., 2025), we generate simple, executable programs with additional constraints to reduce difficulty at each step. Complex execution thus emerges from composition of simple steps, which enable evaluation of models' contextual flexibility at executing arbitrary programs, contrasting with fixed-procedure tasks (e.g., large number multiplication) whose complexity is tightly coupled with input size. With synthetic data, we train small Transformers ($\sim$154M parameters) with diverse modifications. Evaluation is limited below training sequence length to avoid confounding of length extrapolation. While all models achieve strong in-distribution accuracy, they struggle to reliably generalize to various factors. Experimental results reveal substantial impacts of architectural choices on out-of-distribution performance, with NAPE (mixing NoPE and ALiBi heads) on average outperforming other modifications. Our results expose limitations in existing Transformers at executing programs step by step, an ability crucial to reasoning tasks and for precise instruction-following in large language models.

2 Program trace generation

The synthetic programs are standalone functions varying code segments among assignment, binary operations,[1] if block, while block, list append or pop. We generate these examples following the methodology of L0-Bench (Sun et al., 2025). More details are provided in Appendix A. With constraints on the simplicity at each step, execution of complex programs becomes sequentially composing basic operations, and the trace steps grow at least linearly with the input size. Therefore, compared to tasks requiring direct production of final answers, we focus on the case where models leverage computations that change with problem complexity. The research question thus shifts from "Can Transformers execute a fixed program *internally* given unseen/longer input?" to "Can Transformers execute simple operations consistently over long sequence given arbitrary programs?"

[1]We further restrict arithmetic ops to be only $x + 1$ and $x - 1$, comparison ops to only $==$ and $!=$.

39th Conference on Neural Information Processing Systems (NeurIPS 2025) Workshop: What Can('t) Transformers Do?.

3 Experimental setup

Data & Evaluation metrics. Training and evaluation data are generated while controlling four factors: program length, trace steps, number of variables, and input size. Training data contains programs of 5 to 40 lines, 5 to 103 trace steps, maximum 10 variables, and input list of 5 to 10 integers. Short examples are concatenated together, while longer examples do not exceed 4096 tokens. For out-of-distribution evaluation, we vary each factor independently while keeping others fixed: we test on *longer programs*, *longer traces*, *more variables*, and *longer lists* (i.e., larger input size). Motivation of studying these factors is in Appendix. We report whole-trace accuracy: a trace is correct only if every step matches the ground truth exactly. More details are provided in Appendix A.

Model & Training. We train small Transformers of size $\sim$154M parameters on a mixture of short and medium length programs. In addition to standard RoPE-based Transformers, we also experiment with other position encodings (NoPE, ALiBi, NaPE, Fox, PaTH), softmax replacements (STB, α-entmax), short 1d-convolution (Canon) and hybrid model (SWAN). Description of these methods are provided in Appendix B. We perform fixed-budget model comparison with 20k training steps, which is sufficient for the baseline (NoPE) to achieve near perfect in-distribution performance. We focus on OOD performance to explore the compatibility of various inductive biases with PTG. All models are trained with 4k-token sequences and vocabulary size of 176. We report the average and standard error of each model across 10 random seeds. More details are provided in Appendix C.

4 Results

Table 1: Whole-trace accuracy (%) across test conditions. ID: in-distribution test splits.

	ID (short)	**ID (mid)**	**Longer lists**	**More variables**	**Longer traces - 1**	**Longer traces - 2**	**Longer programs**
NoPE	$99.8_{\pm 0.2}$	$99.0_{\pm 0.4}$	$90.3_{\pm 2.5}$	$94.1_{\pm 2.0}$	$91.6_{\pm 2.7}$	$14.2_{\pm 2.5}$	$46.5_{\pm 2.2}$
RoPE	$98.0_{\pm 1.1}$	$92.6_{\pm 3.1}$	$65.5_{\pm 4.2}$	$92.9_{\pm 3.5}$	$78.5_{\pm 5.2}$	$3.9_{\pm 1.0}$	$40.8_{\pm 3.3}$
ALiBi	$99.4_{\pm 0.4}$	$96.8_{\pm 1.9}$	$86.2_{\pm 3.5}$	$92.1_{\pm 1.7}$	$93.1_{\pm 1.5}$	$39.5_{\pm 4.7}$	$50.8_{\pm 2.2}$
NaPE	$99.7_{\pm 0.3}$	$98.3_{\pm 1.6}$	$93.4_{\pm 1.6}$	$98.4_{\pm 0.9}$	$97.9_{\pm 1.3}$	$93.5_{\pm 1.4}$	$57.4_{\pm 2.1}$
Fox	$99.2_{\pm 0.4}$	$95.8_{\pm 2.1}$	$90.0_{\pm 3.3}$	$95.8_{\pm 1.8}$	$77.9_{\pm 1.5}$	$19.2_{\pm 1.7}$	$42.9_{\pm 1.1}$
PaTH	$100.0_{\pm 0.0}$	$99.2_{\pm 0.2}$	$89.7_{\pm 3.8}$	$90.7_{\pm 3.1}$	$88.8_{\pm 2.4}$	$30.9_{\pm 5.3}$	$52.0_{\pm 4.5}$
SWAN	$97.9_{\pm 1.1}$	$85.3_{\pm 3.6}$	$29.8_{\pm 3.9}$	$80.5_{\pm 2.6}$	$49.5_{\pm 3.9}$	$0.5_{\pm 0.2}$	$21.9_{\pm 2.5}$
NoPE+Canon	$100.0_{\pm 0.0}$	$100.0_{\pm 0.0}$	$58.0_{\pm 8.1}$	$98.4_{\pm 0.5}$	$95.3_{\pm 1.4}$	$35.3_{\pm 6.9}$	$64.8_{\pm 2.4}$
STB	$99.1_{\pm 0.5}$	$82.2_{\pm 4.4}$	$81.9_{\pm 4.4}$	$50.7_{\pm 3.8}$	$66.4_{\pm 2.7}$	$11.2_{\pm 1.5}$	$21.6_{\pm 3.7}$
NoPE+α-Entmax	$100.0_{\pm 0.0}$	$99.1_{\pm 0.1}$	$41.6_{\pm 2.2}$	$98.2_{\pm 0.3}$	$98.0_{\pm 0.4}$	$27.1_{\pm 1.8}$	$49.5_{\pm 0.7}$

Table 1 shows whole-trace accuracy across architectures and test conditions. All models achieve strong in-distribution accuracy but vary significantly in generalization performance. Reliably generating long traces remains challenging across models with large degradation in whole-trace accuracy when increasing output trace steps (*longer traces - 1* vs. *- 2*). Models also struggle with more complex/longer instructions in the context: none reaches above 70% accuracy. Dealing with larger entity size (*longer lists*), a factor often studied by existing work, is heavily impacted by positional encoding, with standard RoPE falling behind recent variants, such as Fox and PaTH. Despite its simplicity, NaPE is surprisingly powerful at generating longer traces, being the only variant achieving over 90% accuracy while others are below 40%. Finally, Canon layers improve performance on longer programs (64.8% for NoPE+Canon) but lead to degradation on longer lists (90.3% vs. 58.0%).

5 Discussion

We empirically studied Transformers on program trace generation, a new task requiring precise rule-following over long sequences of simple computational steps. We compare multiple modifications to standard Transformers, including alternative position encoding methods, softmax replacements and novel layer designs. Experimental results show that while these models achieve near-perfect in-distribution performance, they struggle to maintain global accuracy, especially for longer programs and longer traces unseen during training. Architectural choices also significantly impact performance, with NaPE outperforming others. In the future, we plan to extend our work to latest architectures potentially including non-Transformers and MoE models. Finally, we also plan to explore the transduction setting, which may require additional modeling capabilities atop contextual flexibility.

References

Zeyuan Allen-Zhu. 2025. Physics of Language Models: Part 4.1, Architecture Design and the Magic of Canon Layers. *SSRN Electronic Journal*. `https://ssrn.com/abstract=5240330`.

Zeyuan Allen-Zhu and Yuanzhi Li. 2025. Physics of language models: Part 1, learning hierarchical language structures.

Gregoire Deletang, Anian Ruoss, Jordi Grau-Moya, Tim Genewein, Li Kevin Wenliang, Elliot Catt, Chris Cundy, Marcus Hutter, Shane Legg, Joel Veness, and Pedro A Ortega. 2023. Neural networks and the chomsky hierarchy. In *The Eleventh International Conference on Learning Representations*.

Amirhossein Kazemnejad, Inkit Padhi, Karthikeyan Natesan, Payel Das, and Siva Reddy. 2023. The impact of positional encoding on length generalization in transformers. In *Thirty-seventh Conference on Neural Information Processing Systems*.

Zhixuan Lin, Evgenii Nikishin, Xu He, and Aaron Courville. 2025. Forgetting transformer: Softmax attention with a forget gate. In *The Thirteenth International Conference on Learning Representations*.

Bingbin Liu, Jordan T. Ash, Surbhi Goel, Akshay Krishnamurthy, and Cyril Zhang. 2023. Exposing attention glitches with flip-flop language modeling. In *Thirty-seventh Conference on Neural Information Processing Systems*.

William Merrill, Jackson Petty, and Ashish Sabharwal. 2024. The illusion of state in state-space models. In *Forty-first International Conference on Machine Learning*.

William Merrill and Ashish Sabharwal. 2024. The expressive power of transformers with chain of thought. In *The Twelfth International Conference on Learning Representations*.

Ofir Press, Noah Smith, and Mike Lewis. 2022. Train short, test long: Attention with linear biases enables input length extrapolation. In *International Conference on Learning Representations*.

Krishna C. Puvvada, Faisal Ladhak, Santiago Akle Serrano, Cheng-Ping Hsieh, Shantanu Acharya, Somshubra Majumdar, Fei Jia, Samuel Kriman, Simeng Sun, Dima Rekesh, and Boris Ginsburg. 2025. Swan-gpt: An efficient and scalable approach for long-context language modeling.

Jianlin Su, Yu Lu, Shengfeng Pan, Ahmed Murtadha, Bo Wen, and Yunfeng Liu. 2023. Roformer: Enhanced transformer with rotary position embedding.

Simeng Sun, Cheng-Ping Hsieh, Faisal Ladhak, Erik Arakelyan, Santiago Akle Serano, and Boris Ginsburg. 2025. L0-reasoning bench: Evaluating procedural correctness in language models via simple program execution.

Shawn Tan, Songlin Yang, Aaron Courville, Rameswar Panda, and Yikang Shen. 2025. Scaling stick-breaking attention: An efficient implementation and in-depth study. In *The Thirteenth International Conference on Learning Representations*.

Pavlo Vasylenko, Marcos Treviso, and André F. T. Martins. 2025. Long-context generalization with sparse attention.

Jie Wang, Tao Ji, Yuanbin Wu, Hang Yan, Tao Gui, Qi Zhang, Xuanjing Huang, and Xiaoling Wang. 2024. Length generalization of causal transformers without position encoding.

Songlin Yang, Yikang Shen, Kaiyue Wen, Shawn Tan, Mayank Mishra, Liliang Ren, Rameswar Panda, and Yoon Kim. 2025. Path attention: Position encoding via accumulating householder transformations.

Wojciech Zaremba and Ilya Sutskever. 2015. Learning to execute.

Hattie Zhou, Arwen Bradley, Etai Littwin, Noam Razin, Omid Saremi, Josh Susskind, Samy Bengio, and Preetum Nakkiran. 2023. What algorithms can transformers learn? a study in length generalization.

Łukasz Kaiser and Ilya Sutskever. 2016. Neural gpus learn algorithms.

A Program Trace Generation

Program generation & Trace format. We follow L0-Bench to generate simple synthetic Python programs using generative grammar. Each program is a standalone Python function with coding segments varying among assignment, if block, while block and list operations (append/pop). We further reduce the difficulty of each line of code by enforcing the binary arithmetic operations to be only increment or decrement by 1; the scope depth is set to 1 to avoid nested loops; list indexing is also excluded. An example of PTG instance is shown in Figure 1. The traces are generated by executing the programs given sampled inputs and post-processing Python execution bytecodes. Specifically, each trace is of format `line_number,var_name:var_value`. For code lines that do not have value updates, only `line_number,` is required.

Data generation & statistics. We list the distribution of training and test data configs in Table 2. Each program is unique. Training and in-distribution (ID) test splits do not have any overlap. During training, with $p = 0.5$, we shift the first line of the program (and correspondingly the trace) by an offset drawn between 1 and 100. Without doing so models fail to achieve non-zero accuracy for the *longer programs* split due to unseen line number in the programs and traces. Preliminary experiments also show the importance of including examples of diverse program/trace lengths to avoid overfitting to certain trace steps; training data are generated by binning short (5-20 lines of code) programs and mid (21-40 lines of code) programs by their trace steps, details listed in Table 3.

Task comparison & Four factors. The task program trace generation essentially forces models to perform reliable sequential operations like a computer (Łukasz Kaiser and Sutskever, 2016; Zaremba and Sutskever, 2015) with *guaranteed correctness*. However, PTG differs from existing algorithmic tasks (such as string reversal, modular addition, permutation composition, or in general recognizing regular languages) (Zhou et al., 2023; Kazemnejad et al., 2023; Deletang et al., 2023; Liu et al., 2023; Merrill et al., 2024; Allen-Zhu and Li, 2025) in two key ways. First, PTG provides the algorithm *explicitly* in the context, while existing tasks require models internalizing the fixed algorithms into parameters during training. In other words, the context window now serves as memory: the beginning part hosting the "instruction memory" and the trace/scratchpad/chain-of-thought (Merrill and Sabharwal, 2024) part " data memory." Second, the inputs and algorithms are decoupled, i.e., programs now have additional complexity terms independent of input size. We further refine the setup by Sun et al. (2025) to encompass four factors for more targeted evaluation of program execution. Specifically, we focus on::

- **Program length**: Given a fixed max window size (4096 in our experiments), the split *longer programs* tests how models behave when the size of "instruction memory" increases. Longer programs also correspond to more complex procedures which often have higher branching factors.
- **Trace steps**: Given a fixed max window size (4096 in our experiments), the split *longer traces* tests how models behave when the size of "data memory" increases. This can be achieved by increasing the while block size and/or the while loop iteration numbers without increasing the program length.
- **Number of variables**: The split *more variables* tests whether models can handle programs with larger number of input variables, some of which may serve as distractors.
- **Input size**: The split *longer lists* tests if models can apply in-distribution algorithm to larger entity size. This is commonly studied in existing works, such as training models up to n-digit addition and testing on inputs with $[n + 1, k]$ digit inputs.

Scratchpad size. Give a list of T elements and a program of N code lines with k while loop (scope-depth 1) iterations, PTG requires generation of $O(kNT)$ tokens, similar to executing a linear-time algorithm with explicit scratchpad generation. This contrasts with trace-free tasks that test whether models can perform equivalent computation entirely within hidden states.

Table 2: Distribution of training and eval data configs. OOD dimensions are highlighted in blue. Program length in number of lines; Trace length in number of steps; input size in the length of the list.

Split	Program length	Trace steps	Num of variables	Input size	Avg. Out Tokens
TRAIN					
Short program	[5,20]	[5,103]	[1,10]	[5,10]	-
Mid program	[21,40]	[5,103]	[1,10]	[5,10]	-
EVAL					
ID (short)	[5,20]	[59,79]	[1,10]	[5,10]	713.2
ID (mid)	(20,40]	[90,103]	[1,10]	[5,10]	1052.3
Longer lists	(20,40]	[5,103]	[1,10]	[10,15]	828.1
More variables	(20,40]	[5,103]	[10,50]	[5,10]	644.8
Longer traces - 1	(20,40]	[121,142]	[1,10]	[5,10]	1580.5
Longer traces - 2	(20,40]	[172,300]	[1,10]	[5,10]	2402.3
Longer programs	(40,60]	[5,103]	[1,10]	[5,10]	1279.3

Table 3: Data split statistics. **pl**, program length in number of code liens; **tr** trace size in number of trace steps.

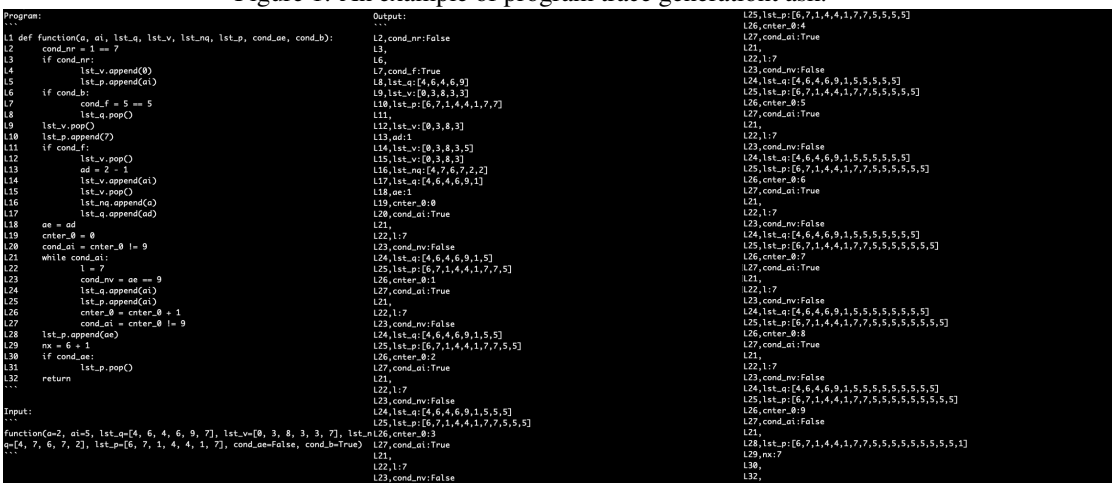
Train split	Avg. Token per example	Total num. tokens
pl [5, 20] tr (5,7]	204.06	8.14E+07
pl [5, 20] tr (7, 8]	208.19	8.31E+07
pl [5, 20] tr (8, 10]	268.01	1.07E+08
pl [5, 20] tr (10, 15]	341.37	1.36E+08
pl [5, 20] tr (15, 40]	548.64	2.19E+08
pl [5, 20] tr (40, 59]	749.28	2.99E+08
pl [5, 20] tr (59, 79]	986.17	3.93E+08
pl [20, 40] tr (14, 25]	615.42	2.46E+08
pl [20, 40] tr (25, 48]	860.15	3.43E+08
pl [20, 40] tr (48, 63]	1057.39	4.22E+08
pl [20, 40] tr (63, 77]	1244.97	4.97E+08
pl [20, 40] tr (77, 90]	1419.38	5.66E+08
pl [20, 40] tr (90, 103]	1600.17	6.38E+08
	Sum	4.03E+09

B Model description

C Experimental configurations

Detailed training configurations are provided in Table 5. Transformer variants evaluated in this work are listed in Table 4. For STB and ASEntmax, we experiments with both RoPE and NoPE, and report the best for each (STB+RoPE, ASEntmax+NoPE). All models are trained with A100 GPUs.

Figure 1: An example of program trace generationt ask.



```
Program:
```
def function(a, ai, lst_a, lst_v, lst_na, lst_p, cond_ae, cond_b):
    cond_nr = 1 == 7
    if cond_nr:
        lst_v.append(0)
        lst_p.append(ai)
    if cond_b:
        cond_f = 5 == 5
        lst_q.pop()
    lst_v.pop()
    lst_p.append(7)
    if cond_f:
        lst_v.pop()
        ad = 2 - 1
        lst_v.append(ai)
        lst_v.pop()
        lst_na.append(a)
        lst_q.append(ad)
    ae = ad
    cnter_0 = 0
    cond_ai = cnter_0 != 9
    while cond_ai:
        l = 7
        cond_nv = ae == 9
        lst_q.append(ai)
        lst_p.append(ai)
        cnter_0 = cnter_0 + 1
        cond_ai = cnter_0 != 9
    lst_p.append(ae)
    nx = 6 + 1
    if cond_ae:
        lst_p.pop()
    return
```

Input:
```
function(a=2, ai=5, lst_a=[4, 6, 4, 6, 9, 7], lst_v=[0, 3, 8, 3, 3, 7], lst_na=[4, 7, 6, 7, 2], lst_p=[6, 7, 1, 4, 4, 1, 7], cond_ae=False, cond_b=True)
```

Output:
```
L2,cond_nr:False
L3,
L6,
L7,cond_f:True
L8,lst_q:[4,6,4,6,9]
L9,lst_v:[0,3,8,3,3]
L10,lst_p:[6,7,1,4,4,1,7,7]
L11,
L12,lst_v:[0,3,8,3]
L13,ad:1
L14,lst_v:[0,3,8,3,5]
L15,lst_v:[0,3,8,3]
L16,lst_na:[4,7,6,7,2,2]
L17,lst_a:[4,6,4,6,9,1]
L18,ae:1
L19,cnter_0:0
L20,cond_ai:True
L21,
L22,l:7
L23,cond_nv:False
L24,lst_q:[4,6,4,6,9,1,5]
L25,lst_p:[6,7,1,4,4,1,7,7,5]
L26,cnter_0:1
L27,cond_ai:True
L21,
L22,l:7
L23,cond_nv:False
L24,lst_a:[4,6,4,6,9,1,5,5]
L25,lst_p:[6,7,1,4,4,1,7,7,5,5]
L26,cnter_0:2
L27,cond_ai:True
L21,
L22,l:7
L23,cond_nv:False
L24,lst_a:[4,6,4,6,9,1,5,5,5]
L25,lst_p:[6,7,1,4,4,1,7,7,5,5,5]
L26,cnter_0:3
L27,cond_ai:True
L21,
L22,l:7
L23,cond_nv:False
L24,lst_a:[4,6,4,6,9,1,5,5,5,5]
L25,lst_p:[6,7,1,4,4,1,7,7,5,5,5,5]
L26,cnter_0:4
L27,cond_ai:True
L21,
L22,l:7
L23,cond_nv:False
L24,lst_q:[4,6,4,6,9,1,5,5,5,5,5]
L25,lst_p:[6,7,1,4,4,1,7,7,5,5,5,5,5]
L26,cnter_0:5
L27,cond_ai:True
L21,
L22,l:7
L23,cond_nv:False
L24,lst_q:[4,6,4,6,9,1,5,5,5,5,5,5]
L25,lst_p:[6,7,1,4,4,1,7,7,5,5,5,5,5,5]
L26,cnter_0:6
L27,cond_ai:True
L21,
L22,l:7
L23,cond_nv:False
L24,lst_q:[4,6,4,6,9,1,5,5,5,5,5,5,5]
L25,lst_p:[6,7,1,4,4,1,7,7,5,5,5,5,5,5,5]
L26,cnter_0:7
L27,cond_ai:True
L21,
L22,l:7
L23,cond_nv:False
L24,lst_q:[4,6,4,6,9,1,5,5,5,5,5,5,5,5]
L25,lst_p:[6,7,1,4,4,1,7,7,5,5,5,5,5,5,5,5]
L26,cnter_0:8
L27,cond_ai:True
L21,
L22,l:7
L23,cond_nv:False
L24,lst_q:[4,6,4,6,9,1,5,5,5,5,5,5,5,5,5]
L25,lst_p:[6,7,1,4,4,1,7,7,5,5,5,5,5,5,5,5,5]
L26,cnter_0:9
L27,cond_ai:False
L21,
L28,lst_p:[6,7,1,4,4,1,7,7,5,5,5,5,5,5,5,5,5,1]
L29,nx:7
L30,
L32,
```

Table 4: Model description

| Model | Description |
|---|---|
| **RoPE** (Su et al., 2023) | Rotate query/key vectors before softmax attention. |
| **NoPE** (Wang et al., 2024) | Not applying any explicit positional encoding. |
| **SWAN** (Puvvada et al., 2025) | Interweaving sliding window (RoPE) and global attention (NoPE) layers. |
| **ALiBi** (Press et al., 2022) | Attention logits added by $m_k(j-i)$ for $q_i$, $k_j$, and head-specific slope $m_k$. |
| **NaPE** (Vasylenko et al., 2025) | Slope of latter half of ALiBi is set to 0. |
| **Fox** (Lin et al., 2025) | Data-dependent forget gate added to attention logits. |
| **Canon** (Allen-Zhu, 2025) | Short 1-D depthwise causal convolution applied to position "abc." |
| **STB** (Tan et al., 2025) | Replace Softmax in attention with strong recency-biased stick-breaking. |
| $\alpha$**-Entmax** (Vasylenko et al., 2025) | Sparse attention allowing 0 weight on tokens whose scores below a threshold. |
| **PaTH** (Yang et al., 2025) | Data-dependent positional encoding via cumulative Householder transformation. |

Table 5: Training and model configuration summary.

| Config name | Value |
|---|---|
| **Number of layers** | 12 |
| **Number of heads** | 16 |
| **Embedding dim** | 1024 |
| **Feedforward** | SwiGLU |
| **Learning rate** | 3e-4 |
| **LR scheduler** | Cosine |
| **Adam Betas** | (0.95, 0.9) |
| **Weight decay** | 0.1 |
| **Warmup steps** | 1000 |
| **Total train steps** | 20000 |
| **Batch size** | 128 |
| **Sequence length** | 4096 |
| **Decoding** | Greedy |
| **Model-specific configurations** | |
| **RoPE base** | 10000 |
| **ALiBi slopes** | $\text{slope}_i = (2^{-\frac{8}{n}})^{i+1}$ for $i = 0 \dots n-1$ |
| **SWAN layers** | [1,2,3,5,6,7,9,10,11] |
| **SWAN window size** | 512 |
| **Canon conv window** | 4 |
| **Canon added position** | "abc" |
| **ASEntmax** $\alpha$ | 1.5 |
```

Figure 2: NoPE + Canon accuracy across evaluation splits throughout training (average over 10 random seeds). In-distribution performance saturates much earlier than OOD splits. The learning speed for longer programs is much slower than other splits.