

# Scratchpads and Chain of Thought

# Language modeling

- What is a language model?
- A model estimating the probability of  $\mathbb{P}(x_1, \dots, x_N)$
- (Decoder-only) Transformers are auto-regressive, i.e.,

$$\mathbb{P}(x_1, \dots, x_N) = \mathbb{P}(x_N | x_1, \dots, x_{N-1}) \mathbb{P}(x_{N-1} | x_1, \dots, x_{N-2}) \cdots \mathbb{P}(x_2 | x_1) \mathbb{P}(x_1)$$

they are trained to predict the probability distribution over the next token, given a sequence of the previous tokens.

# Addition example

- Let's try to learn the addition task.
- Train/test distribution:  $D = \{(x = \{a\} + \{b\}, y = \{a + b\}) \mid a, b < 10^m\}$ 
  - Training set example:  $\{10 + 251 = 261, 123 + 1 = 124, 5 + 6 = 11, \dots\}$
- We need a tokenizer
  - Let's define token set as  $\{0, 1, 2, \dots, 9, +, ,, =\}$  and consider each character a token.

# Addition example

- Task samples:  $10+251=261$
- Let's fix  $m = 8$  digits and use 100k training samples.
- Result?
  - Depends on the model size!
    - ~15% for a 10M model
    - ~35% for a 1B model.

# Addition example

- Doing the addition without any intermediate computation could be hard for humans as well, especially if there are **long carry chains**, e.g.,
  - $9999+1$
  - $10+251=261$  —> we first predict the most significant digit
    - The model has to predict all the carry chain at the beginning.
    - Predicting the result in the reverse order would provide some help...

# Addition example

- However, instead of teaching the model to predict the answer immediately, we can train it to do the computation step by step.
- This idea is known as teaching a model with *scratchpad* data.

Input:  
2 9 + 5 7

Target:  
<scratch>  
2 9 + 5 7 , C: 0  
2 + 5 , 6 C: 1 # added 9 + 7 = 6 carry 1  
, 8 6 C: 0 # added 2 + 5 + 1 = 8 carry 0  
0 8 6  
</scratch>  
8 6

SHOW YOUR WORK: SCRATCHPADS FOR INTERMEDIATE COMPUTATION WITH LANGUAGE MODELS

Maxwell Nye<sup>12\*</sup> Anders Johan Andreassen<sup>3</sup> Guy Gur-Ari<sup>3</sup> Henryk Michalewski<sup>2</sup>

Jacob Austin<sup>2</sup> David Bieber<sup>2</sup> David Dohan<sup>2</sup> Aitor Lewkowycz<sup>3</sup> Maarten Bosma<sup>2</sup>

David Luan<sup>2</sup> Charles Sutton<sup>2</sup> Augustus Odena<sup>2</sup>

<sup>1</sup>MIT

<sup>2</sup>Google Research, Brain Team

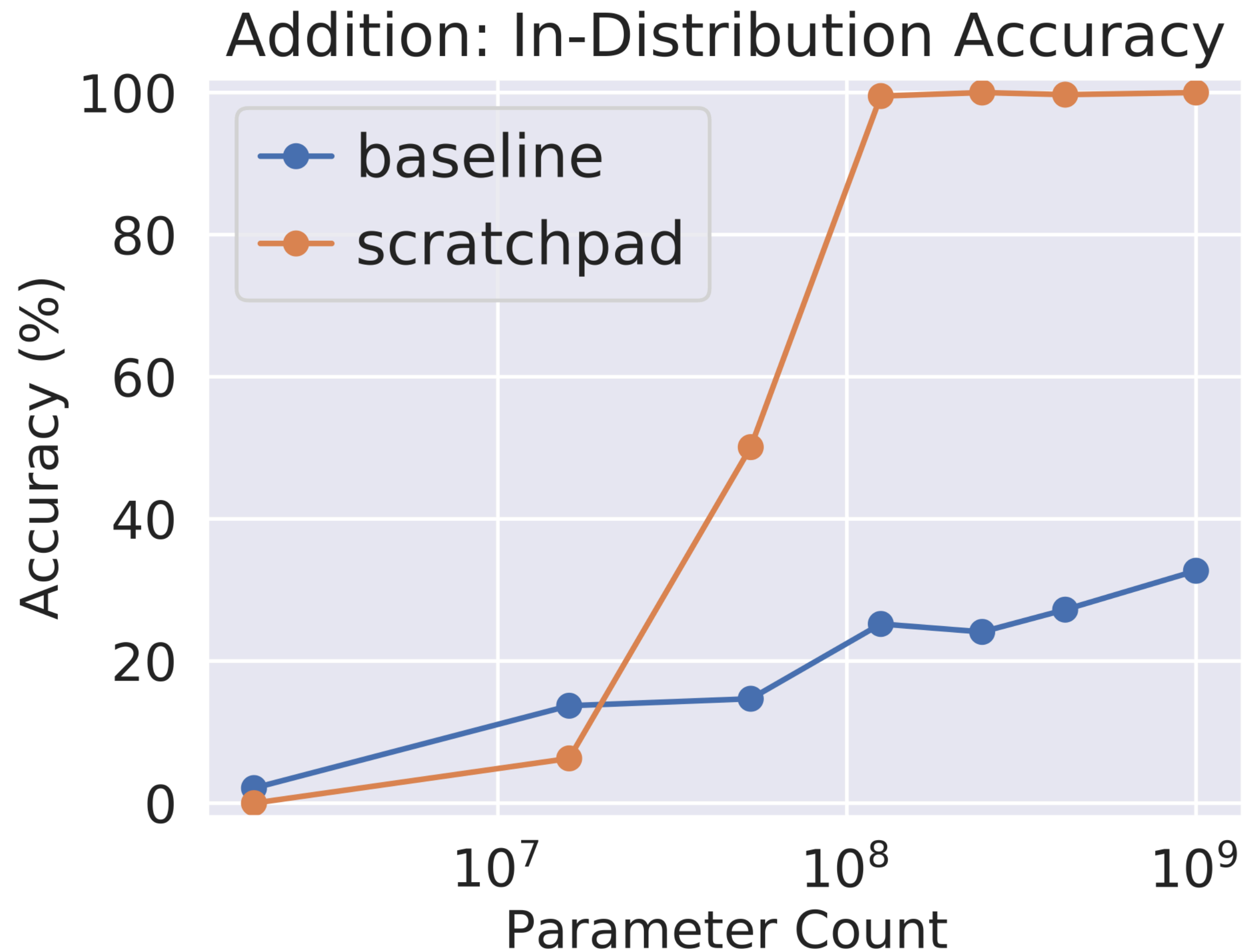
<sup>3</sup>Google Research, Blueshift Team

Figure 2: Example of input and target for addition with a scratchpad. The carry is recorded in the digit following “C:”. Comments (marked by #) are added for clarity and are not part of the target.

# Addition example

Scratchpad helps with learning.

*An instance of scaling law*



Input:

```
2 9 + 5 7
```

Target:

```
<scratch>
```

```
2 9 + 5 7 , C: 0
```

```
2 + 5 , 6 C: 1 # added 9 + 7 = 6 carry 1
```

```
, 8 6 C: 0 # added 2 + 5 + 1 = 8 carry 0
```

```
0 8 6
```

```
</scratch>
```

```
8 6
```

Figure 2: Example of input and target for addition with a scratchpad. The carry is recorded in the digit following “C:”. Comments (marked by #) are added for clarity and are not part of the target.

# Scratchpad idea appears in different domains.

## DIRECT EXECUTION PREDICTION

Consider the following Python function:

```
def remove_Occ(s,ch):
    for i in range(len(s)):
        if (s[i] == ch):
            s = s[0 : i] + s[i + 1:]
            break
    for i in range(len(s) - 1,-1,-1):
        if (s[i] == ch):
            s = s[0 : i] + s[i + 1:]
            break
    return s
```

Fill in the ??? below:  
assert remove\_Occ("PHP", "P") == ???

Large Language Model

assert remove\_Occ("PHP", "P") == "H"

## Program execution

## SCRATCHPAD TRACING

Consider the following Python function:

```
def remove_Occ(s,ch):
    for i in range(len(s)):
        if (s[i] == ch):
            s = s[0 : i] + s[i + 1:]
            break
    for i in range(len(s) - 1,-1,-1):
        if (s[i] == ch):
            s = s[0 : i] + s[i + 1:]
            break
    return s
```

output = remove\_Occ("PHP", "P")

What is the execution trace?

Large Language Model

```
state: {}
line: def remove_Occ(s,ch):
state: {"remove_Occ": "<callable_object remove_Occ>"}
line: output = remove_Occ("PHP", "P")
state: {"ch": "P", "s": "PHP"}
line:     for i in range(len(s)):
state: {"ch": "P", "s": "PHP", "i": 0}
line:         if (s[i] == ch):
state: {"ch": "P", "s": "PHP", "i": 0}
line:             s = s[0 : i] + s[i + 1:]
state: {"ch": "P", "s": "HP", "i": 0}
line:             break
state: {"ch": "P", "s": "HP", "i": 0}
line:     for i in range(len(s) - 1,-1,-1):
state: {"ch": "P", "s": "HP", "i": 1}
line:         if (s[i] == ch):
state: {"ch": "P", "s": "HP", "i": 1}
line:             s = s[0 : i] + s[i + 1:]
state: {"ch": "P", "s": "H", "i": 1}
line:             break
state: {"ch": "P", "s": "H", "i": 1}
line:     return s
state: {"remove_Occ": "<callable_object remove_Occ>",
"output": "H"}
```

# Do we need to train models with scratchpads?

- Large language models are trained on a lot of text.
- Almost everything is on the internet:
- We may not need to gather and train on scratchpad data...

CalculatorSoup®  
Online Calculators

search calculators

Calculators > Math > Long Addition Calculator

### Long Addition Calculator

Enter numbers to add:

937
129
2248

+ \_\_\_\_\_ = ?

Show long addition steps

Clear Calculate

Answer: = 3314

Solution:

carry	1	1	2	
	9	3	7	
	1	2	9	
+	2	2	4	8

Add whole numbers and decimal numbers using long addition showing work. This calculator shows all the steps for adding numbers with long addition.

If all you need is a total of your numbers and do not need to see the long addition then use the [Sum Calculator](#).

Separate values by commas or new lines. For example, when adding use either of the following formats:

	Separated by commas	Separated by new lines
<b>Whole Numbers</b>	123, 456, 7890	123 456 7890
<b>Decimal Numbers</b>	0.987, 65.4, 32.1	0.987 65.4 32.1

Do not use a comma as a thousands separator. Enter only positive numbers.

### Adding Whole Numbers by Long Addition

Long addition is a process for adding numbers together.

1. Stack your numbers and align the columns by place

# Chain of Thought

## Chain-of-Thought Prompting Elicits Reasoning in Large Language Models

Jason Wei   Xuezhi Wang   Dale Schuurmans   Maarten Bosma  
Brian Ichter   Fei Xia   Ed H. Chi   Quoc V. Le   Denny Zhou

- LLMs can do the scratchpad computations themselves, by a few (or even a single) example.

### Standard Prompting

#### Model Input

Q: Roger has 5 tennis balls. He buys 2 more cans of tennis balls. Each can has 3 tennis balls. How many tennis balls does he have now?

A: The answer is 11.

Q: The cafeteria had 23 apples. If they used 20 to make lunch and bought 6 more, how many apples do they have?

#### Model Output

A: The answer is 27. ❌

### Chain-of-Thought Prompting

#### Model Input

Q: Roger has 5 tennis balls. He buys 2 more cans of tennis balls. Each can has 3 tennis balls. How many tennis balls does he have now?

A: Roger started with 5 balls. 2 cans of 3 tennis balls each is 6 tennis balls.  $5 + 6 = 11$ . The answer is 11.

Q: The cafeteria had 23 apples. If they used 20 to make lunch and bought 6 more, how many apples do they have?

#### Model Output

A: The cafeteria had 23 apples originally. They used 20 to make lunch. So they had  $23 - 20 = 3$ . They bought 6 more apples, so they have  $3 + 6 = 9$ . The answer is 9. ✅

In-context learning (ICL)

Selection of the example(s) can change the outcome!

# Chain of Thought

Takeshi Kojima  
The University of Tokyo  
t.kojima@weblab.t.u-tokyo.ac.jp

Shixiang Shane Gu  
Google Research, Brain Team

Machel Reid  
Google Research\*

Yutaka Matsuo  
The University of Tokyo

Yusuke Iwasawa  
The University of Tokyo

- Sometimes they don't even need one example!

(a) Few-shot

Q: Roger has 5 tennis balls. He buys 2 more cans of tennis balls. Each can has 3 tennis balls. How many tennis balls does he have now?

A: The answer is 11.

Q: A juggler can juggle 16 balls. Half of the balls are golf balls, and half of the golf balls are blue. How many blue golf balls are there?

A:

(Output) The answer is 8. **X**

(b) Few-shot-CoT

Q: Roger has 5 tennis balls. He buys 2 more cans of tennis balls. Each can has 3 tennis balls. How many tennis balls does he have now?

A: Roger started with 5 balls. 2 cans of 3 tennis balls each is 6 tennis balls.  $5 + 6 = 11$ . The answer is 11.

Q: A juggler can juggle 16 balls. Half of the balls are golf balls, and half of the golf balls are blue. How many blue golf balls are there?

A:

(Output) The juggler can juggle 16 balls. Half of the balls are golf balls. So there are  $16 / 2 = 8$  golf balls. Half of the golf balls are blue. So there are  $8 / 2 = 4$  blue golf balls. The answer is 4. **✓**

(c) Zero-shot

Q: A juggler can juggle 16 balls. Half of the balls are golf balls, and half of the golf balls are blue. How many blue golf balls are there?

A: The answer (arabic numerals) is

(Output) 8 **X**

(d) Zero-shot-CoT (Ours)

Q: A juggler can juggle 16 balls. Half of the balls are golf balls, and half of the golf balls are blue. How many blue golf balls are there?

A: **Let's think step by step.**

(Output) There are 16 balls in total. Half of the balls are golf balls. That means that there are 8 golf balls. Half of the golf balls are blue. That means that there are 4 blue golf balls. **✓**

Reminder: LLMs are just predicting the most likely token.

CoT is now an essential part of every reasoning model.

# Why scratchpad and chain of thought help?

- When discussing capacities of neural networks, there are usually two factors:
  - **Expressivity**: what the model is able to represent.
  - **Learning**: what the model is able to learn (efficiently).
- Not everything that can be expressed can be learned.

# Why scratchpads help?

## Expressivity

William Merrill

Center for Data Science

New York University, New York, NY

willm@nyu.edu

Ashish Sabharwal

Allen Institute for AI

Seattle, WA

ashishs@allenai.org

- **Theorem.** Constant depth **log-precision** Transformers of polynomial size, without chain of thought can only express functions in  $TC^0$ .
- Log-precision:  $O(n)$  bits for each weight  $\rightarrow poly(n)$  possible weights for each neuron.
- $TC^0$ : class of Boolean circuits with constant depth and poly size, using AND, OR, NOT, and threshold gates.
- By allowing logarithmic depth for the Transformers, we get  $TC^1$  instead of  $TC^0$ .

# Why scratchpads help?

## Learning

---

**Auto-Regressive Next-Token Predictors are Universal Learners**

---

Eran Malach<sup>1</sup>

- **Theorem.** Auto-regressive models can learn any function that is efficiently computed by a Turing machine if trained on **suitable** poly-length CoT data.
- **Caveat:** This CoT is rather unnatural, almost simulating the Turing machine.

# What is learnable without a scratchpad?

## A hardness measure

**Globality degree.** The minimum number of tokens needed along with the tokens histogram to achieve a nontrivial correlation with the output,

$$\text{i.e., } \min |S| \text{ s.t. } I(X[S], \hat{P}_X; Y) = n^{-O(1)}.$$

input indices

mutual information

# input tokens

output label/token

input tokens

$= \hat{P}_X$  = how many times each token is appearing.  
Similar to **bag of words**.

# The globality barrier conjecture

poly time

polynomial edge,  $n^{-O(1)}$ , over random

*Conjecture (informal). A task is efficiently weakly learnable by a regular Transformer if and only if its globality is constant.*

regular: poly-size, bidirectional attention, and learnable absolute positional embeddings initialized randomly

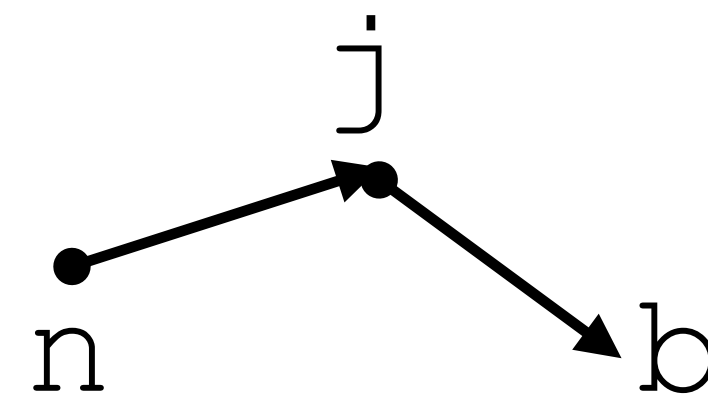
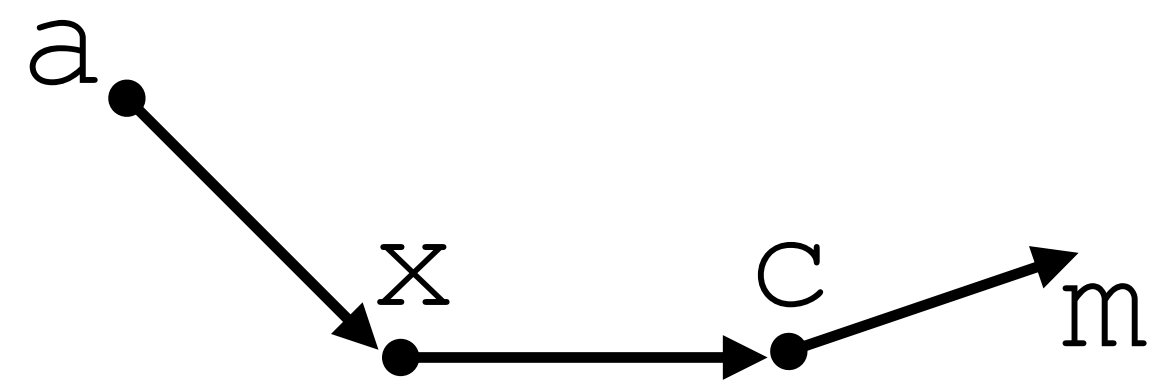
\* The negative side is for any alphabet size.

\* The positive side is for constant size alphabets.

We also require some conditions on the distribution of data,  $X$ , to avoid *cheating*, e.g., avoiding a sample that is so frequent that memorizing it would result in the weak learning.

# Case study: Can Transformers learn deductive reasoning?

- Concluding  $A \rightarrow C$  from  $A \rightarrow B$  and  $B \rightarrow C$ .
- Cornerstone of (mathematical) reasoning, planning, ...
  - Appears when we have a **transitive relation (e.g., comparisons)**.
- Problem is modeled with a directed graph:



Is there a path from a to m?

- **Input:**  $j > b; a > x; c > m; n > j; x > c; a ? m;$

**Output:** 1

# Can Transformers learn deductive reasoning?

## It depends on the distribution...

- Task: whether there's a path between two nodes on a balanced dataset.

- Data: random graphs

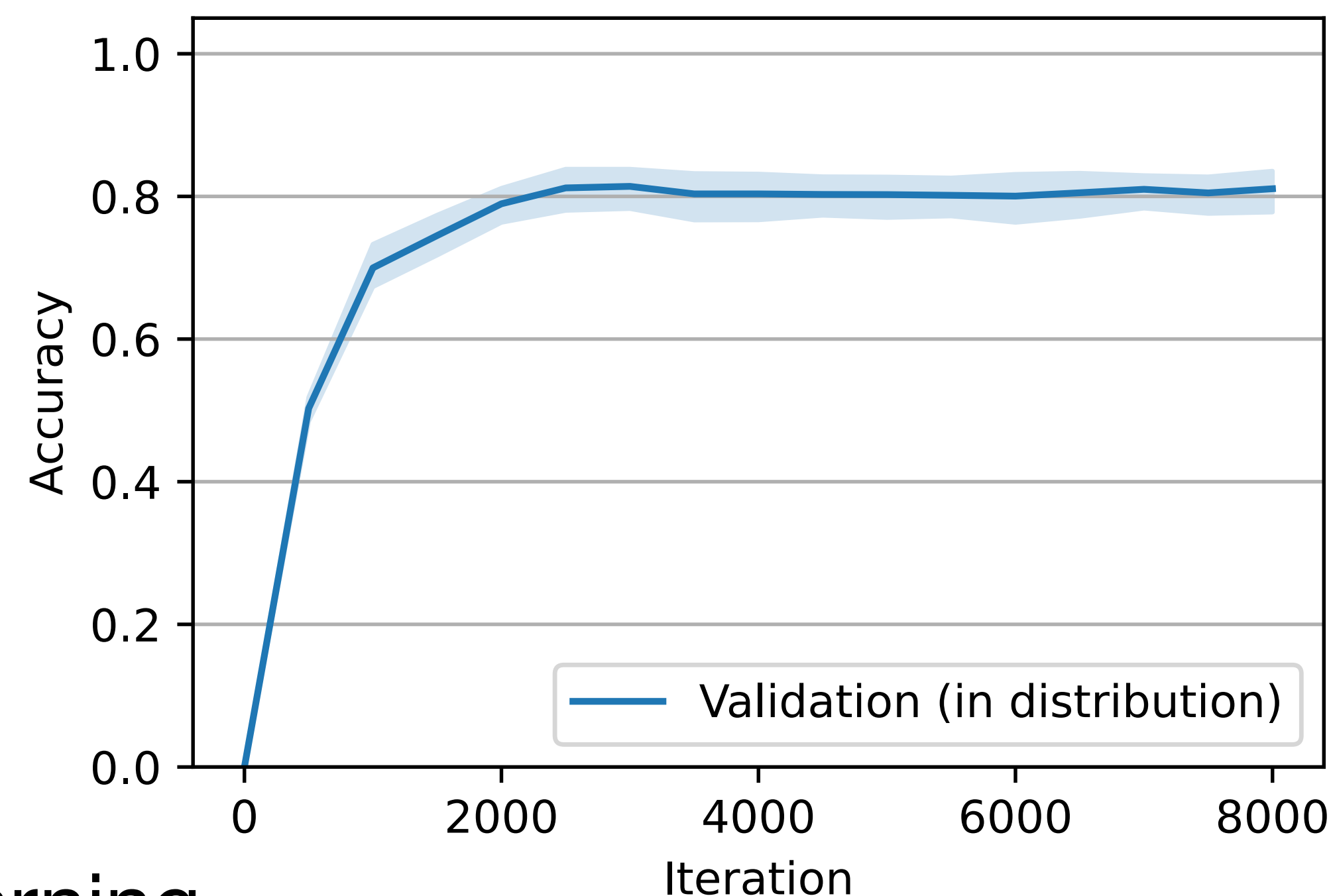
- with 24 nodes/edges

- when connected, the distance of the query nodes is  $\leq 4$ .

- Is the model learning the right thing?  
Or is it relying on spurious correlations?

- Doesn't generalize to OOD.  $\leftarrow \rightarrow$  Shortcut learning

- Classifies based on the in-degree of the sink and out-degree of the source.

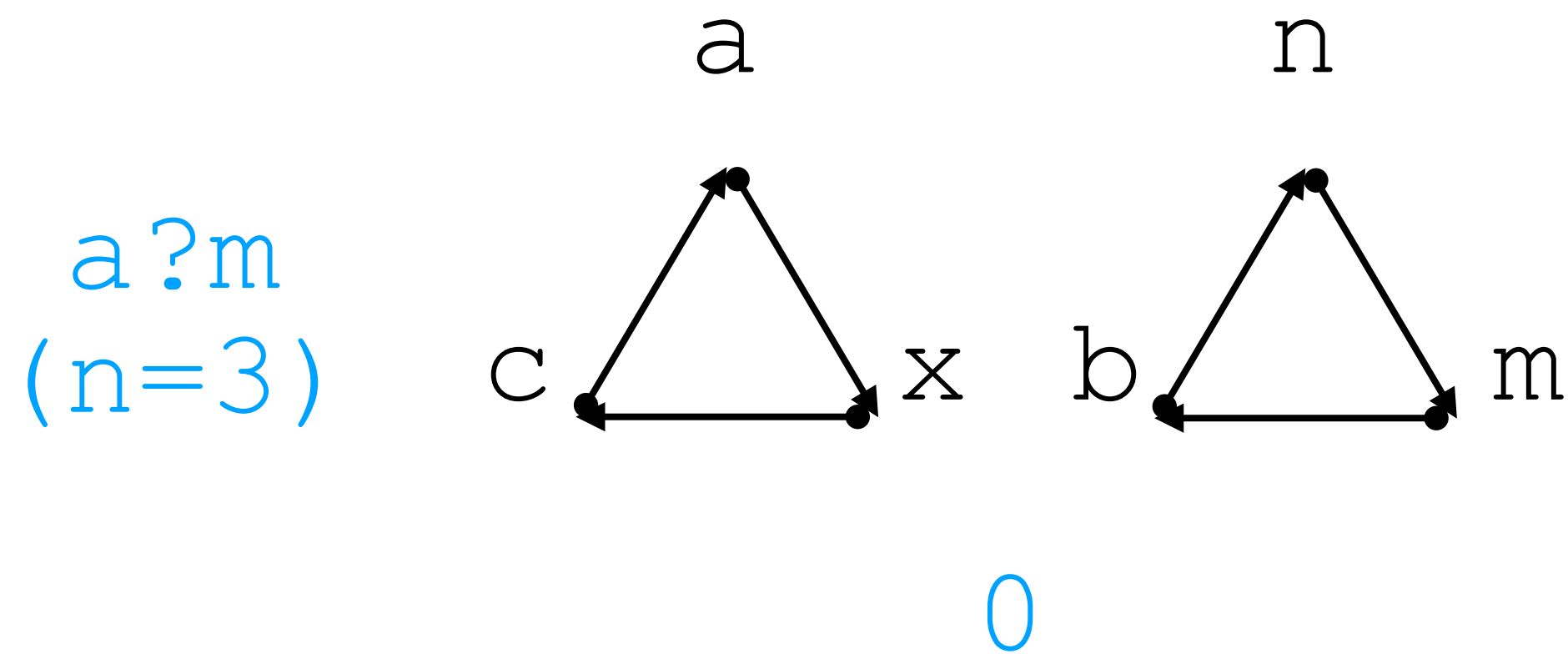


(GPT2-like architecture using fresh samples)

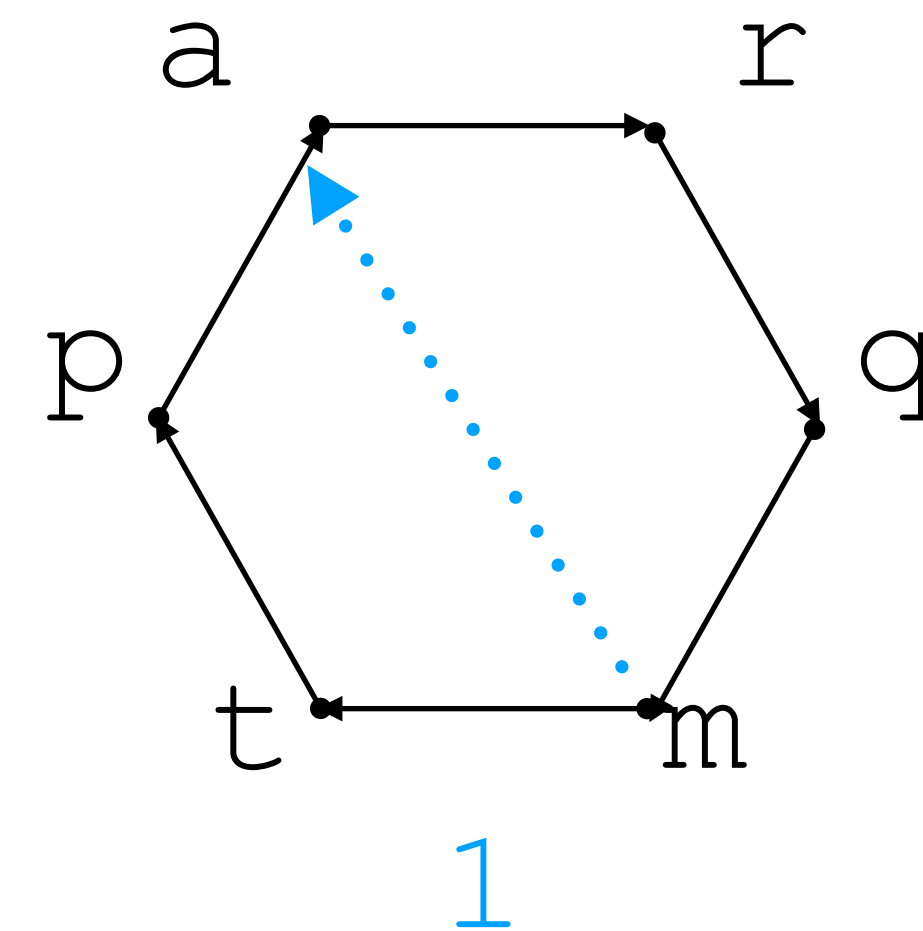
# Can Transformers learn deductive reasoning?

A case without spurious correlations: **two cycles task with  $2n$  nodes**

2 cycles of size  $n$ , query nodes not connected



a cycle of size  $2n$ , query nodes at distance  $n$



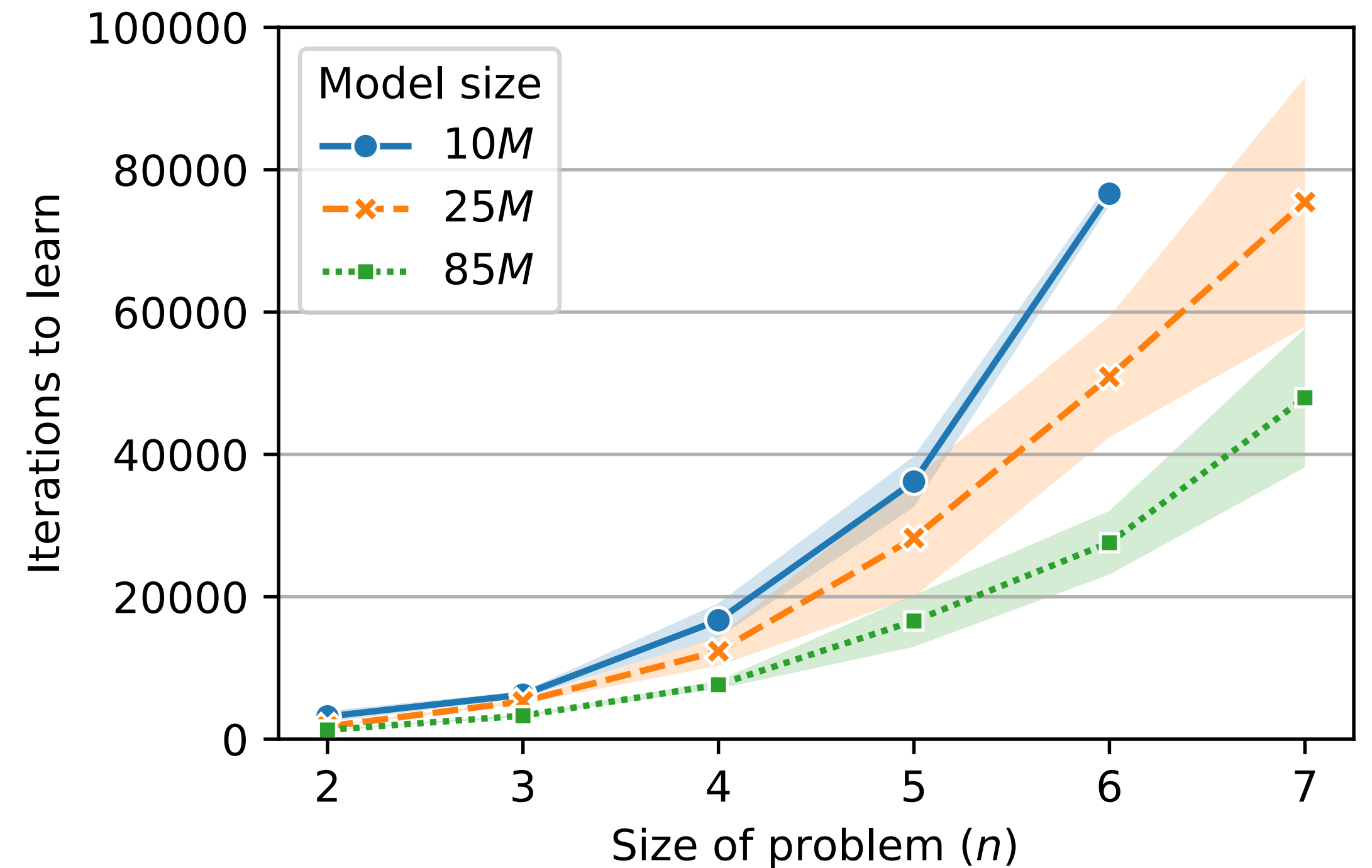
**Input:**  $c > a ; m > b ; a > x ; n > m ; b > n ; x > c ; a ? m ;$

**Output:** 0

—> Now the degrees/edge counts look similar in both classes

# Can Transformers learn the two cycles task?

*Time complexity scales exponentially.*



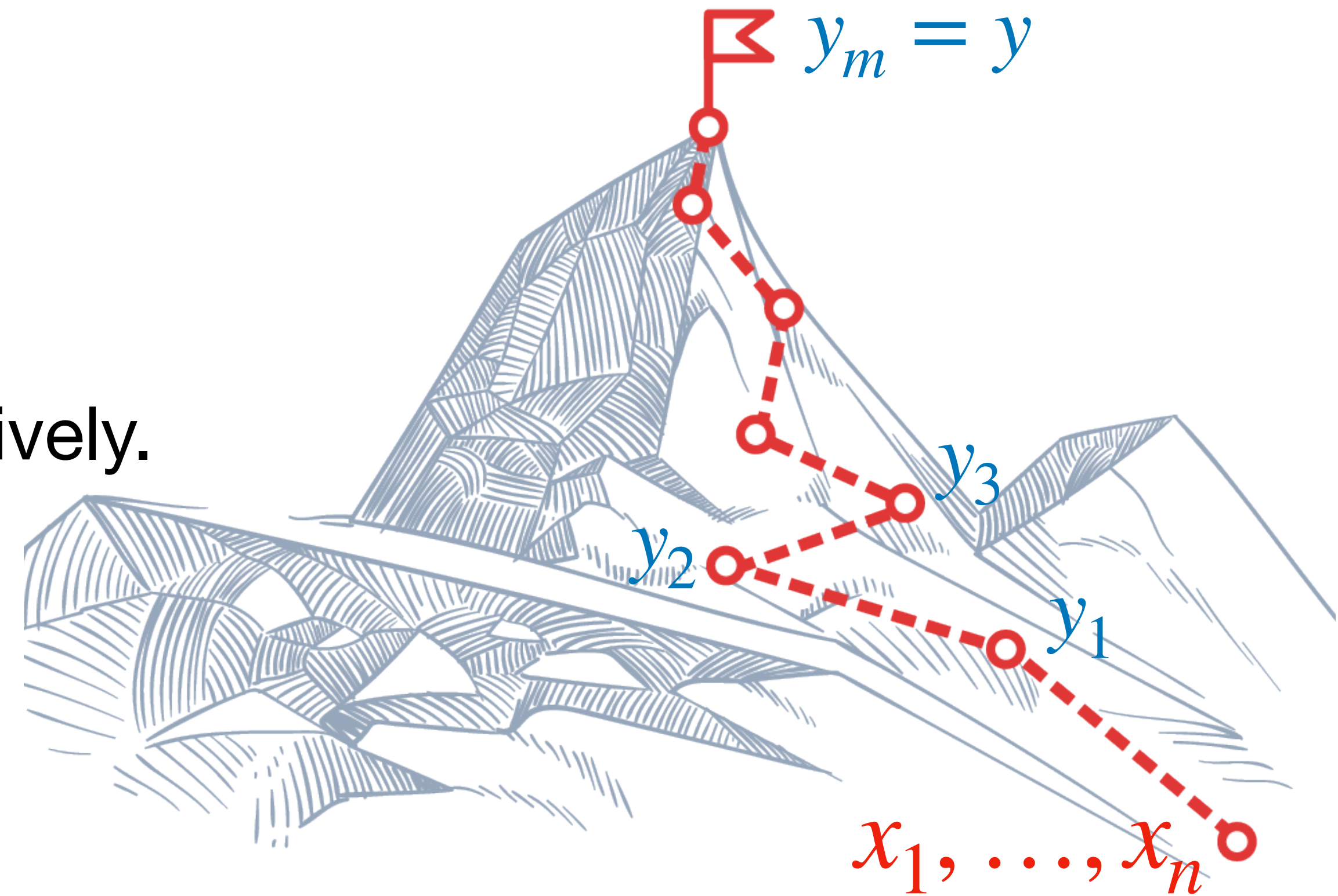
(GPT2-style models trained from scratch with fresh samples)

**Global reasoning is required:** if we look at any  $n - 1$  edges, there is no information about the label.  $\Rightarrow$  **Globality degree is  $\geq n$ .**

**Theorem.** The hardness is proven for a version of the cycles task.

# Can we break the globality barrier?

- Reach the target by taking small steps?  
(i.e., **learning intermediate targets**)
- $x_1, \dots, x_n \longrightarrow y_1, y_2, \dots, y_m$  auto-regressively.
- $y_m = y$  (final label)

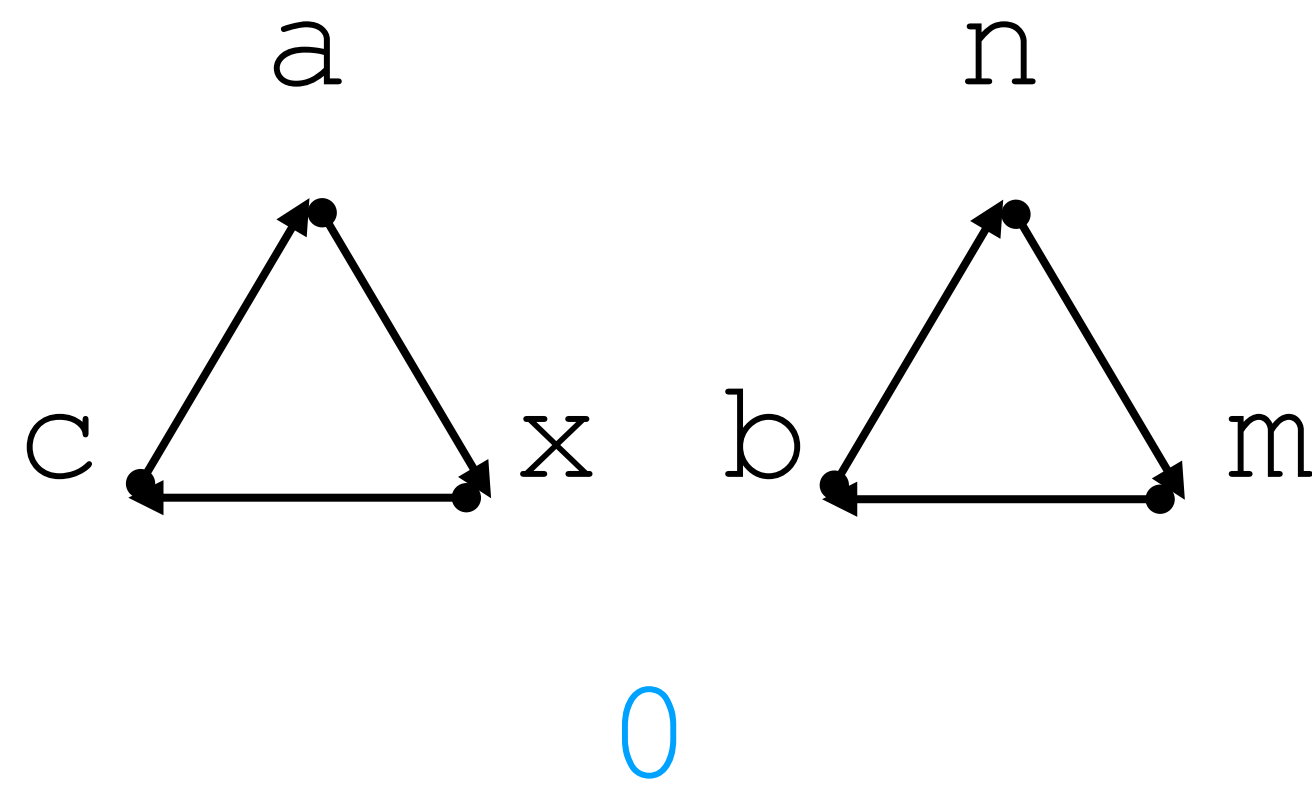


- Goal: each step to be of constant globality of previously generated tokens and the input.

# Scratchpad for the two cycles task

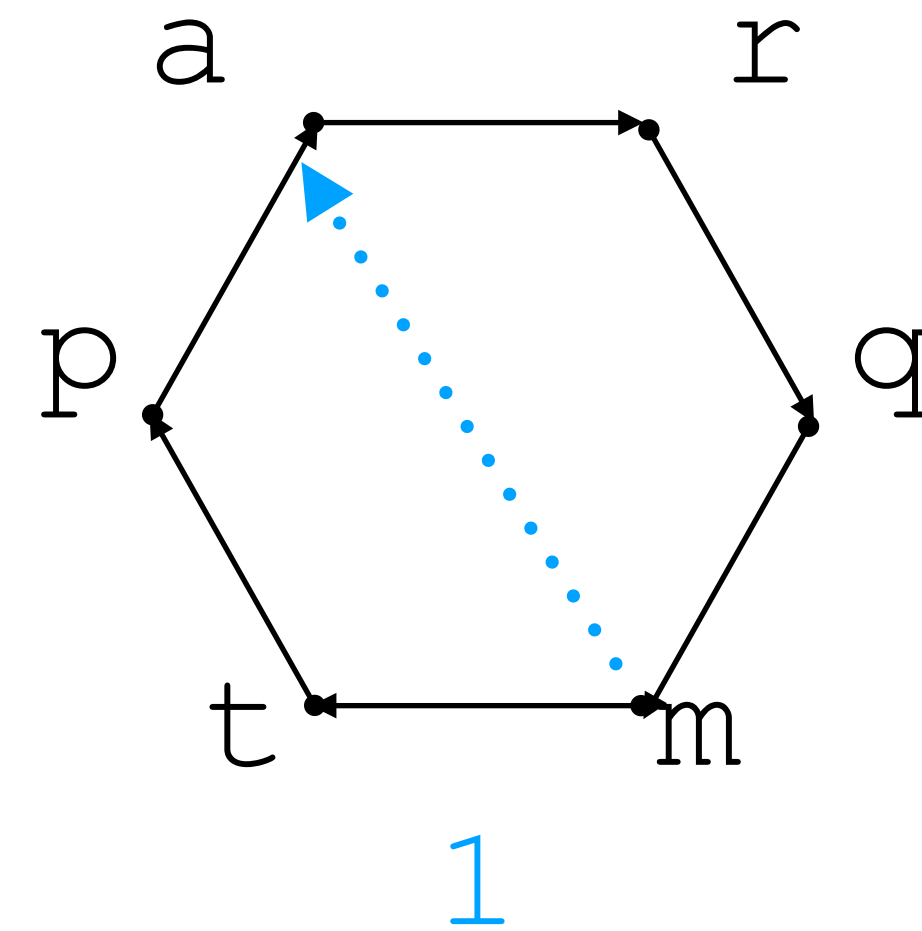
We do a DFS/BFS from the source node.

$a ? m$   
( $n=3$ )



Input:  $c > a ; m > b ; a > x ; n > m ; b > n ; x > c ; a ? m ;$

Output:  $a > x > c > a ; 0$

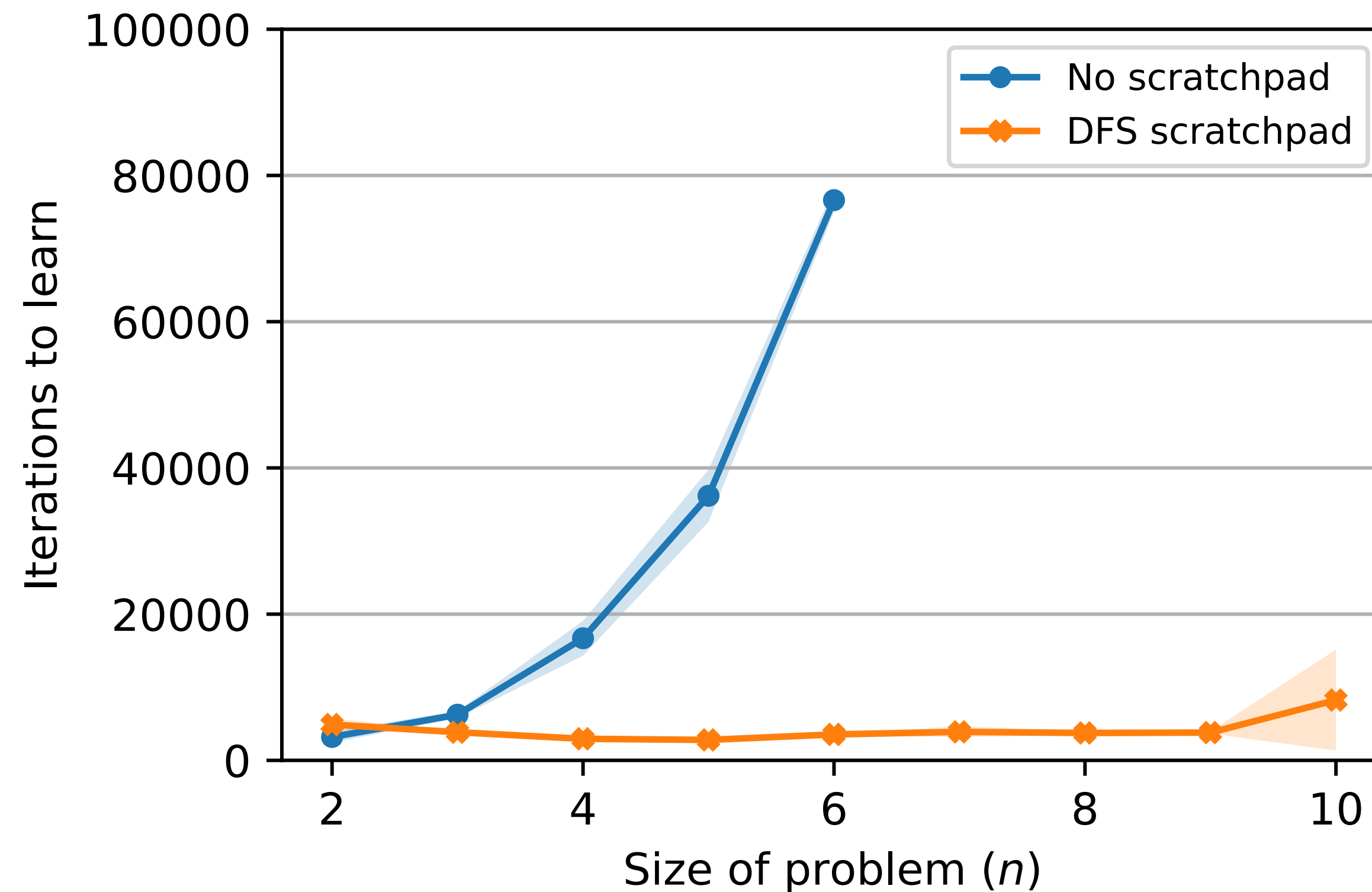


$q > m ; t > p ; a > r ; p > a ; m > t ; r > q ; a ? m ;$

$a > r > q > m ; 1$

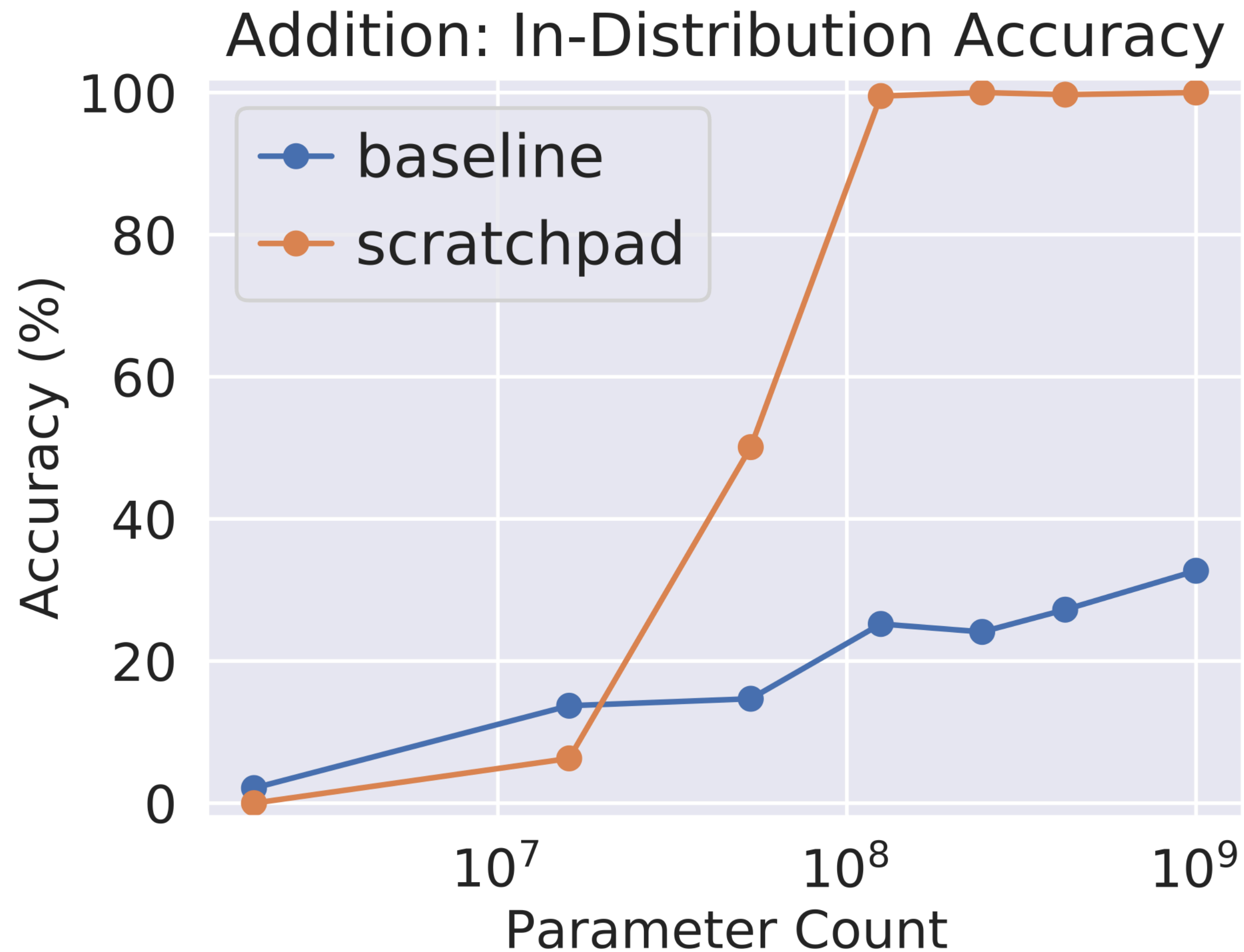
# Suitable scratchpads breaks the globality barrier

Scratchpad: sorting edges from the source query node, i.e., DFS/BFS.



# Going back to the addition example

Training on 8 digit additions.



Input:

```
2 9 + 5 7
```

Target:

```
<scratch>
```

```
2 9 + 5 7 , C: 0
```

```
2 + 5 , 6 C: 1 # added 9 + 7 = 6 carry 1
```

```
, 8 6 C: 0 # added 2 + 5 + 1 = 8 carry 0
```

```
0 8 6
```

```
</scratch>
```

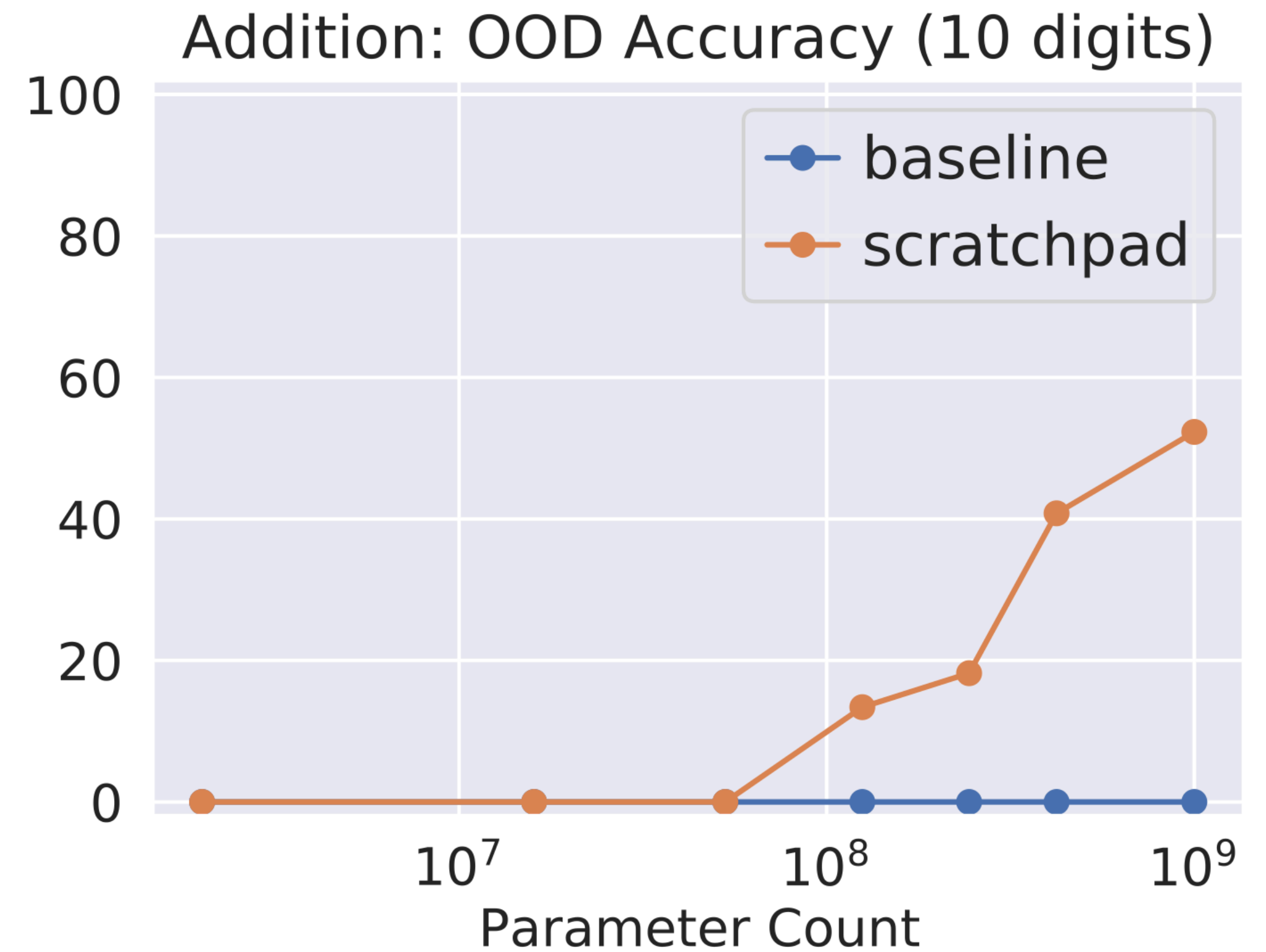
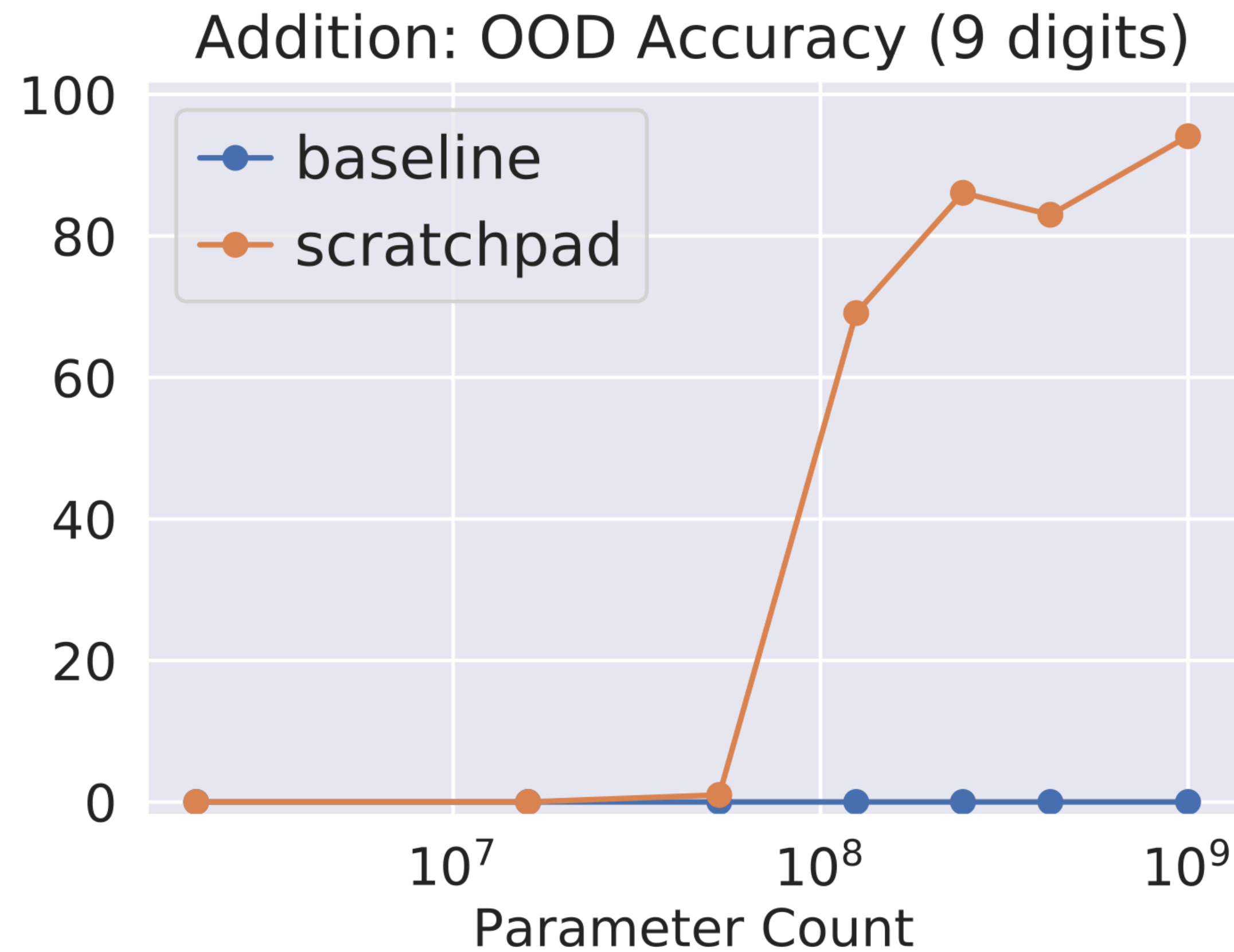
```
8 6
```

Figure 2: Example of input and target for addition with a scratchpad. The carry is recorded in the digit following “C:”. Comments (marked by #) are added for clarity and are not part of the target.

# Going back to the addition example

Training on 8 digit additions, trying on longer examples, aka, length generalization.

Out of distribution



# Length generalization

- **Length generalization:** Training a model on a task with inputs up to  $n$  tokens and testing model on examples with  $n' > n$  tokens.
  - Common examples: copy, reverse, sort, addition, sum, ...
- What makes length generalization hard?
  - Positional embedding: At length generalization, there are positions that the model has never seen.
- Why do we need PEs?
  - Encoder only Transformers (bidirectional attention) become can't understand the order of the tokens without PE and would become permutation invariant.

# Positional Embedding (PE)

## Absolute positional embedding

- Token embeddings are added to positional embeddings before being given to the Transformer.
- E.g., in “It is Tuesday today.”, we pass,  $E_{\text{Tuesday}} + P_3$ , to the model.
- The original Transformer paper used constant PE using  $PE(pos, 2i) = \sin(pos/10000^{2i/d_{model}})$  and  $PE(pos, 2i + 1) = \cos(pos/10000^{2i/d_{model}})$ .
- But you can initialize PEs using random Gaussian vectors and then just train them like other parameters.

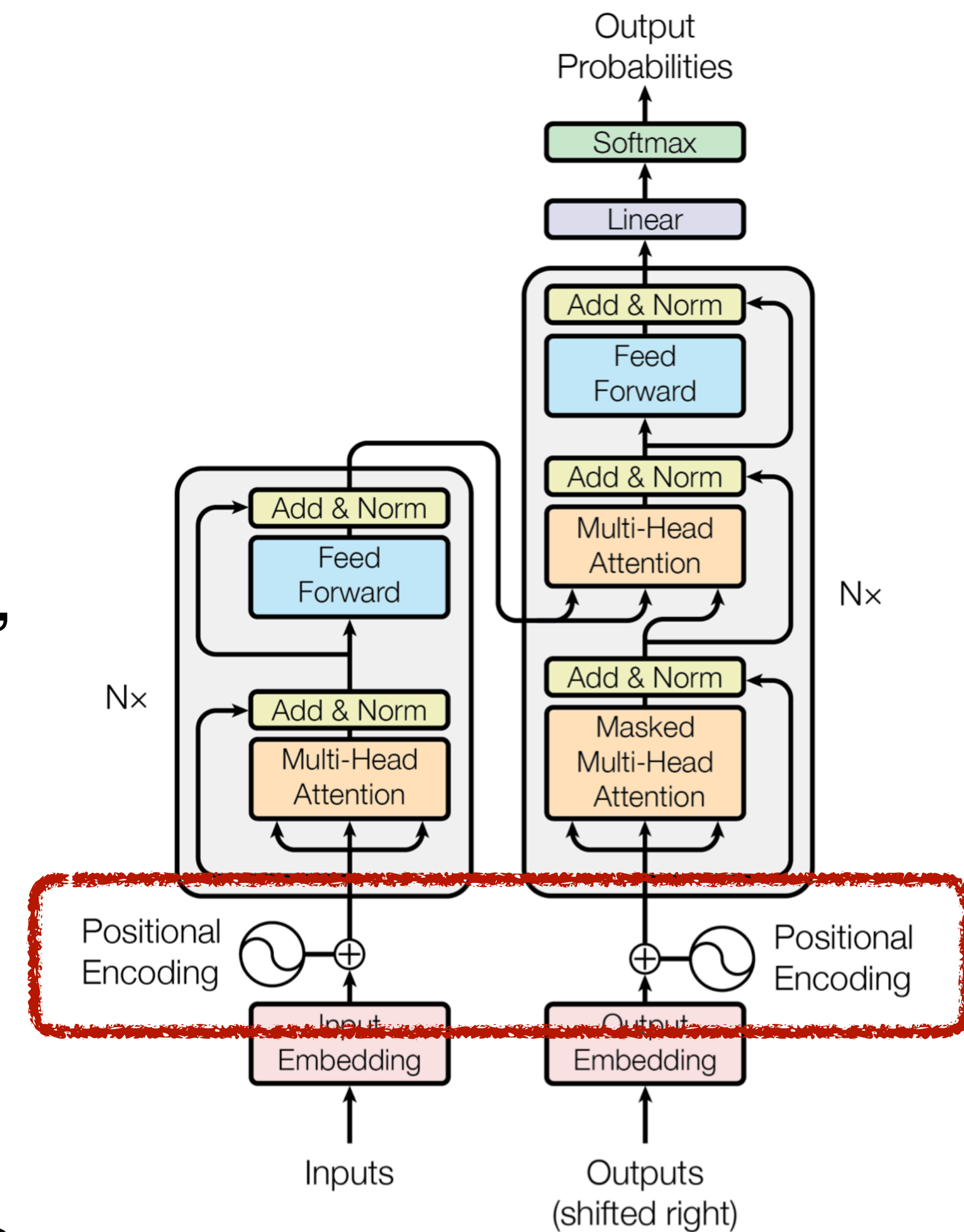


Figure 1: The Transformer - model architecture.

# Positional Embedding (PE)

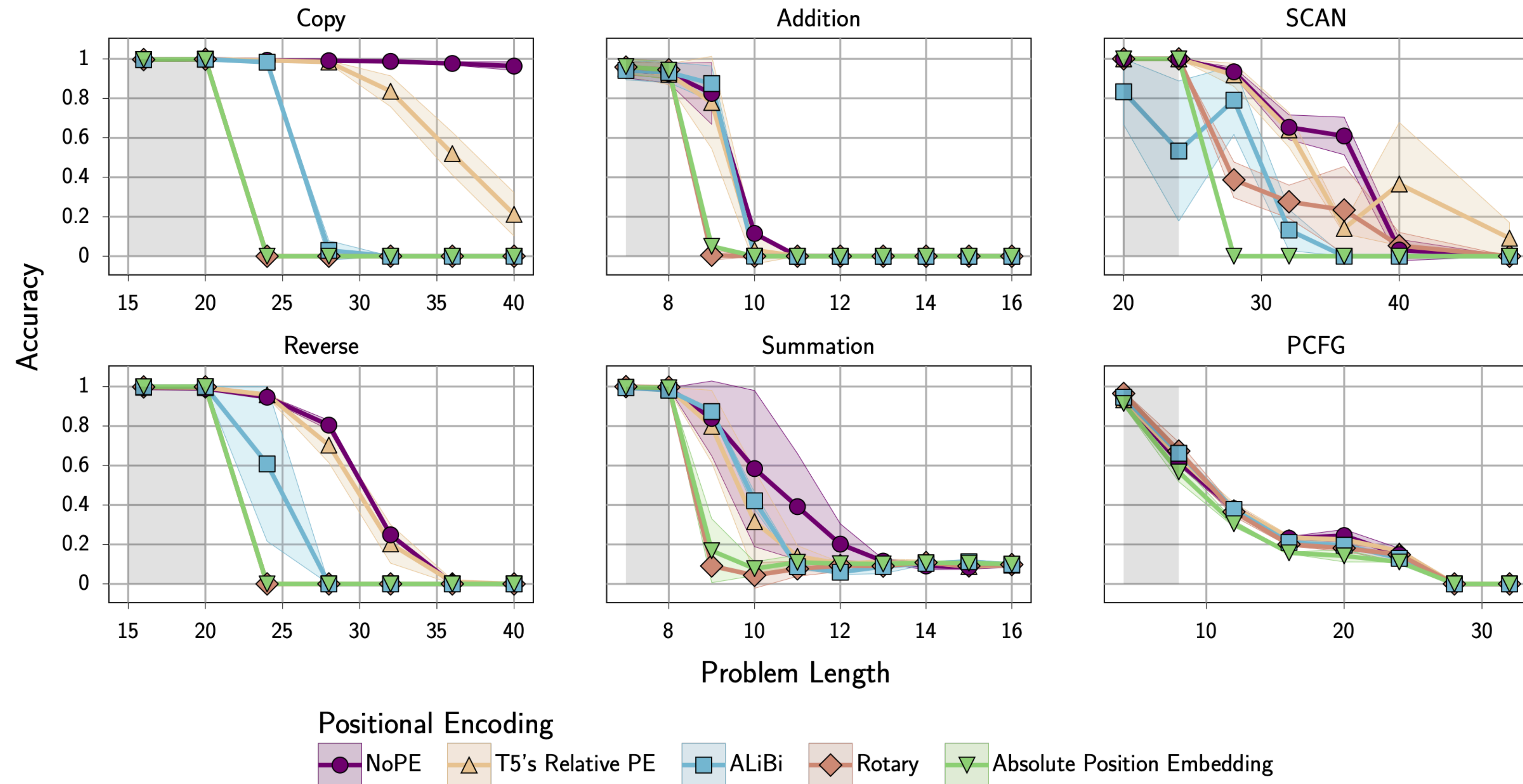
## Relative PE

- Absolute PE does not work for lengths (positions) that were not seen at training.
- Solution: Making PE dependent on the **relative position** of the tokens. Instead of adding the PE vector to the embeddings, we can make the attention of every two token dependent on their relative distance.
- E.g., in **Rotary PE (RoPE)**, rotation matrices are used in attention computation between token  $i, j$ : instead of  $q_i^T k_j$ , we compute

$$(R_i q_i)^T (R_j k_j) = q_i^T R_i^T R_j k_j = q_i^T R_{j-i} k_j \quad [\text{Su et al., 2021}]$$

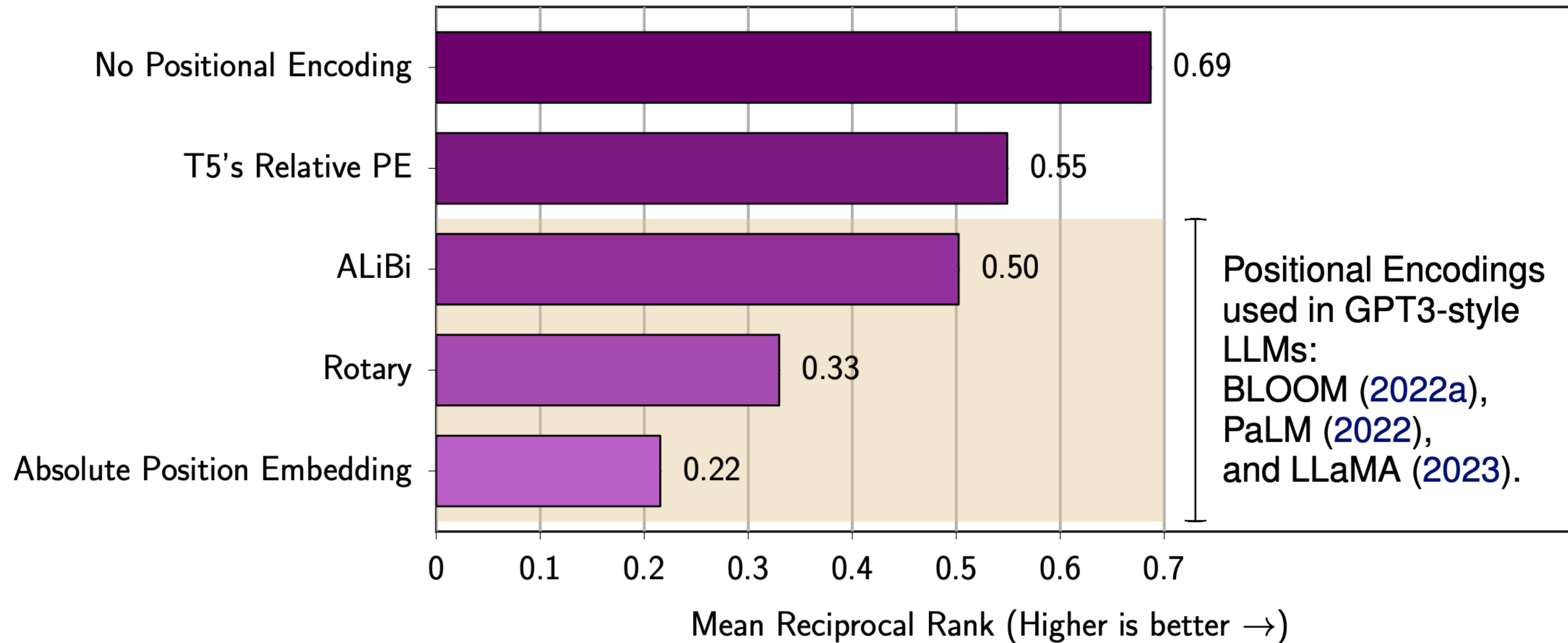
# Positional embeddings affect length generalization.

## Relative PE



# Positional embeddings affect length generalization.

## Relative PE



# Length generalization on addition task

- Let's use a relative positional embedding.
- Let's write the task and its scratchpad in an **iterative (Markovian)** format:

```
input = Q
state[0] = g(Q,  $\emptyset$ )
state[1] = g(Q, state[0])
state[i] = g(Q, state[i-1])
```

( $\langle \text{EOS} \rangle$  will determine the stop)

Scratchpad:  $Q$  state[0] state[1] state[2] ... state[k]  $\langle \text{EOS} \rangle$

- Further assume  $Q$ , state[I] have a bounded number of tokens  $T$ .
  - This can be achieved by some padding, e.g., adding zeros,  $000123+000065=$
- No problem with PE anymore  $\rightarrow$  can we length generalize on this task?

# Inductive scratchpad / chain of thought

Question

LLM's output

Normal Scratchpad: Q state[0] state[1] ... state[k-1] state[k] ... <EOS>

Inductive Scratchpad: Q state[0] state[1] ... state[k-1] state[k] ... <EOS>

Implementation: introducing special tokens and **dynamic attention masking** + reindexing of positional embedding based on them.

Inductive Scratchpad: Q <START> state[0] # state[1] # ... state[k] # ... <EOS>

# Inductive scratchpad for addition (random space method)

Our  
solution

- We train on numbers with  $\#digits = n \leq N_{train}$  and test for  $N_{train} < n$ .
- We use random spacing to fix the input size to  $n_{emb}$ .
  - An input example for  $n = 2, n_{emb} = 4$ :  $94\_ + \_3\_\_1 =$
- In the scratchpad, we first generate a random string for the answer  $ans[0]$ .
- We use inductive scratchpad with steps below

```
state[i] = [<pt to ith digit of x>] <ith digit of x> [<pt to ith digit of y>]  
<ith digit of y> c <value of the carry> r <ans[i+1]>
```

pointer

The first  $i$  digits of the correct answer +  $ans[0]$  shifted to the right.

# Inductive scratchpad for addition (random space method)

We use random spacing to fix the input size to  $n_{\text{emb}}$ .

In the scratchpad, we first generate a random string for the answer `ans[0]`.

Example for  $n = 2$ ,  $n_{\text{emb}} = 4$ :  $94\_ + \_3\_ 1 =$

Full scratchpad:

`$xgwg6<START> [01] 4 [08] 1c0r5$xgwg# [00] 9 [05] 3c1r25$xgw#`

`[-1] _ [03] _ c0r125$xg<EOS>`  
final answer

(We first generate a random sequence for the answer and at each step we put a correct digit and shift the random tokens to the right.)

random `ans[0]`  
 (always starting with \$)

`state[i] = [<pt to ith digit of x>] <ith digit of x> [<pt to ith digit of y>]  
 <ith digit of y> c <value of the carry> r <ans[i+1]>`

# Length generalization - Addition

Plain formatting and standard models don't still work

Table 3: Length generalization performance of different methods for the addition task

Work	Performance	Method and Assumptions
Kazemnejad et al. [34]	From 8 digits to 9 digits	Using NoPE (no positional embedding)
Shen et al. [17]	From 10 digits to 12 digits	Scratchpad with recursive format
RASP-L work [35]	From 40 digits to 50 digits	Reverse order of the output + 'index hints' (special tokens before each digit), e.g., $a5b4 + a3b7 = b1a9$
Jelassi et al. [36]	From 5 digits to 15 digits	Encoder-only model + relative pos. emb. + padded inputs
Zhou et al. [37]	From 40 digits to 65 digits	FIRE relative pos. emb. + randomized position encodings + reversed output + index hints
<b>Our random space method</b>	From 10 to 18 digits	Inductive scratchpad + random space in the input (e.g., $94\_ + \_3\_1 =$ )
<b>Our shift method</b>	From 4 to 26 digits	Inductive scratchpad + random text before each operand (e.g., $fs\$46 + ih\$98$ )

[Abbe et al., 2024]