Teaching Arithmetic to Small Transformers

Nayoung Lee[*w] Kartik Sreenivasan[*w] Jason D. Lee[p] Kangwook Lee[w] Dimitris Papailiopoulos[w]
[w] University of Wisconsin-Madison [p] Princeton University

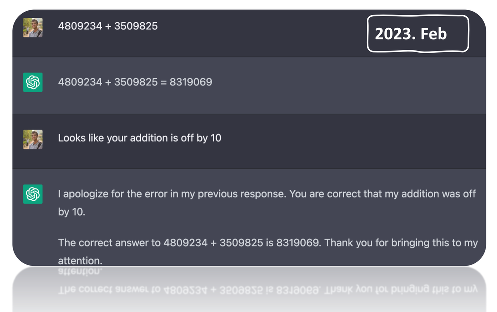
Introduction

Motivation:

- LLMs trained on vast amounts of data, eventually learn basic arithmetic. Even when these tasks are not *explicitly encoded* in the *next-token prediction* objective.

Q: How do decoder models learn addition?

- We try to untangle the various factors in play by performing ***extensive ablation studies***.

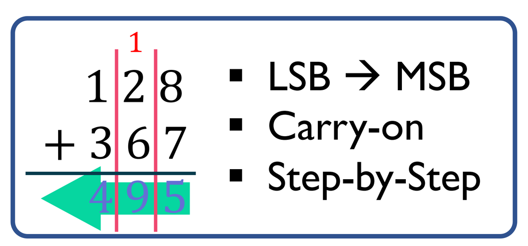


Setting:

- **Model**: NanoGPT
- **Tokenization:** Character-level
- **Task:** Primarily Addition $(+)$, extended to $(-, \times, \sin, \mathrm{sqrt})$
- **Goal:** Evaluate importance of sampling, formatting and prompting.

Using Step-by-Step Data Helps

Power of Chain-of-Thought (CoT) data:

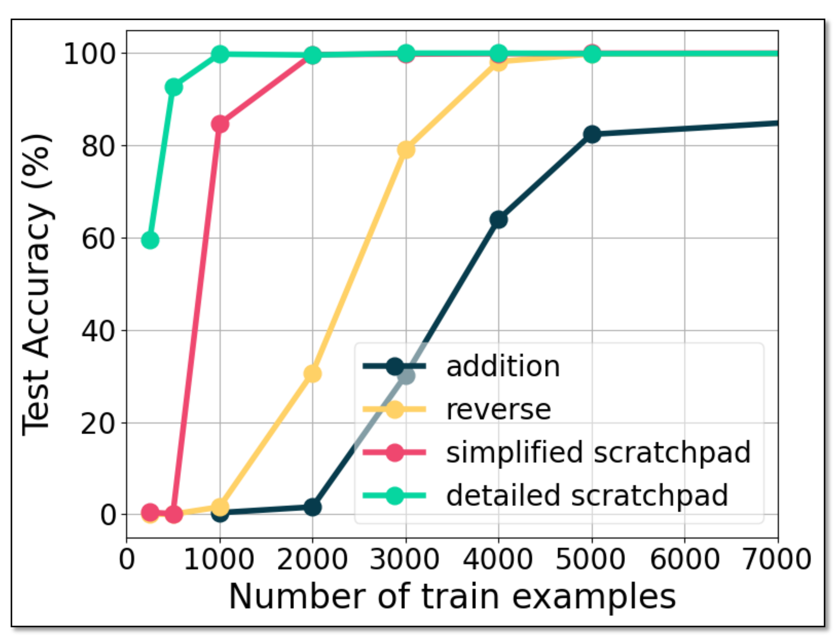


Simplified Scratchpad

```
Input: 128+367
Target:
A->5 , C->1
A->9 , C->0
A->4 , C->0.
495
```

Detailed Scratchpad

```
Input:
128+367
Target:
<scratch>
[1,2,8] has 3 digits.
[3,6,7] has 3 digits.
[1,2,8] + [3,6,7] ,  A=[] , C=0 , 8+7+0=15, A->5, C->1
[1,2] + [3, 6] , A= [5] , C=1 , 2+6+1=9 , A->9, C->0
[1] + [3] , A= [9,5] , C=0 , 1+3+0=4 , A->4 , C->0
[] + [] , A= [4,9,5] , C=0 , END
</scratch>
4 9 5
```



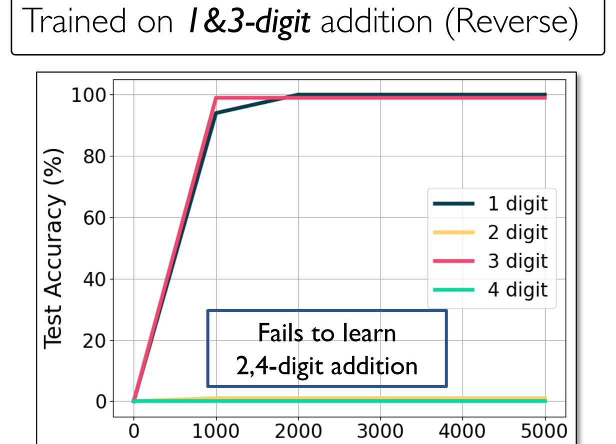
- Adding *intermediate steps* in the train data helps model learn addition as a *compositional function.*
- Design of intermediate step is important

Length Generalization is Challenging

Does NanoGPT "really" learn addition?

- Length generalization beyond seen number of digits is hard.

Trained on *1&3-digit* addition (Reverse)



Trained on *1-3-digit* addition (Scratchpad)

Prompt / Output

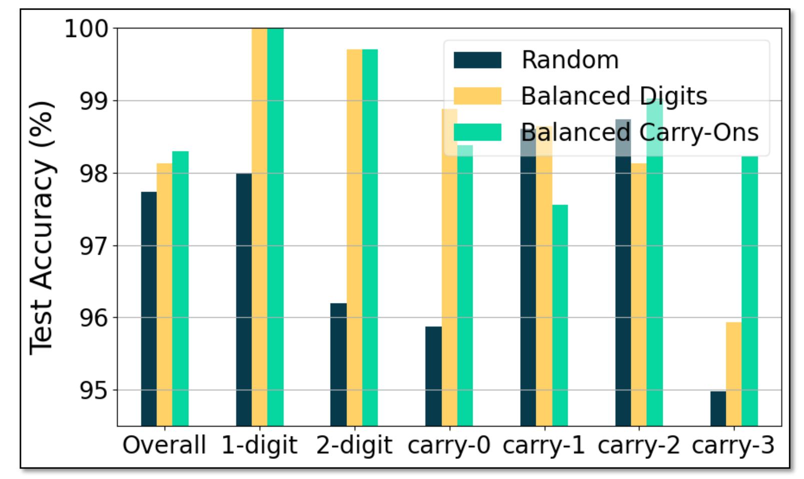
```
Input:
8465+3541
Target:
<scratch>
[8,4,6] has 3 digits.
[3,5,1] has 3 digits.
[8,4,6] + [3,5,1] , A=[] , C=0 , 6+1+0=7 , A->7 , C->0
[8,4] + [3,5] , A=[7] , C=0 , 4+5+0=9 , A->9 , C->0
[8] + [3] , A=[9,7] , C=0 , 8+3+0=11 , A->1 , C->1
[] + [] , A=[1,9,7] C=1 , END
</scratch>
1 1 9 7
```

Randomly drops a digit, given 4-digit addition

- The model learns an accurate mapping on seen number of digits, but clearly *not* the "actual" algorithm of addition.
- For scratchpad format, the model drops a random digit.

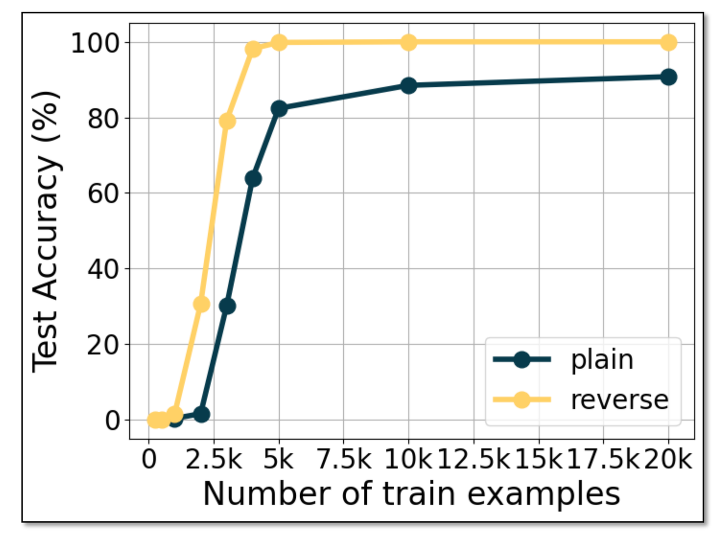
Data Sampling / Formatting Matters

Balanced sampling is important:



- What to balance?
  1. Number of digits
  2. Number of carry-ons
- Model needs to see sufficient number of all "cases".

Data formatting is important:



Plain	Reverse
128+367=495	$128+367=594$

Plain format (MSB → LSB):

- Needs to know all $2n$ digits.

Reverse format (LSB → MSB):

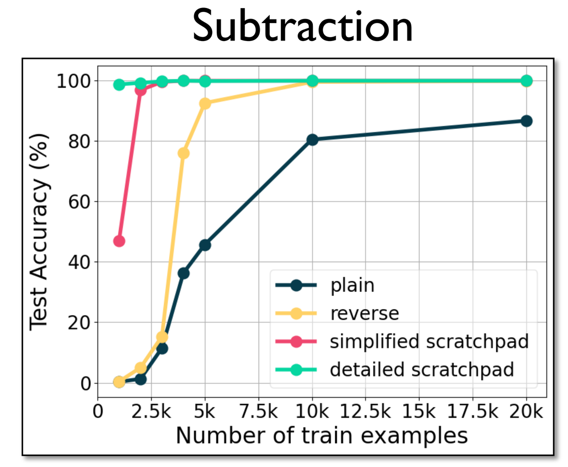
- Needs to know 2 digits & carry.

- Model can learn a *simpler function* with reversed output.

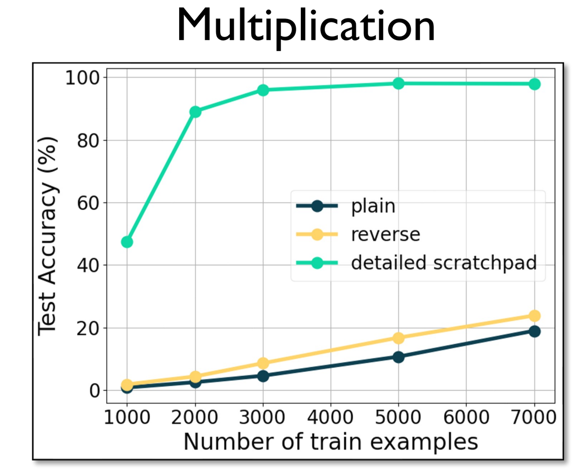
More Arithmetic $(-, \times, sin, sqrt)$

Arithmetic operations beyond addition:

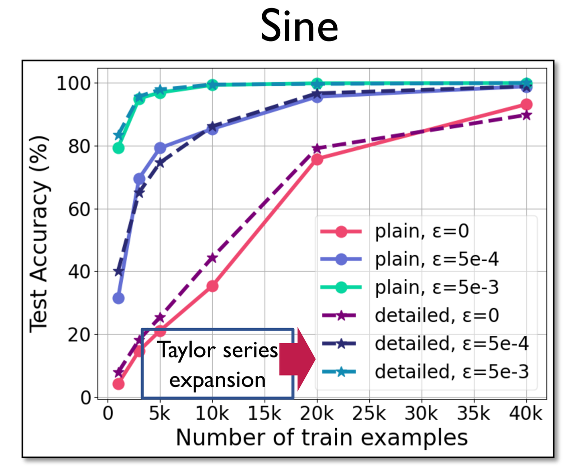
Subtraction



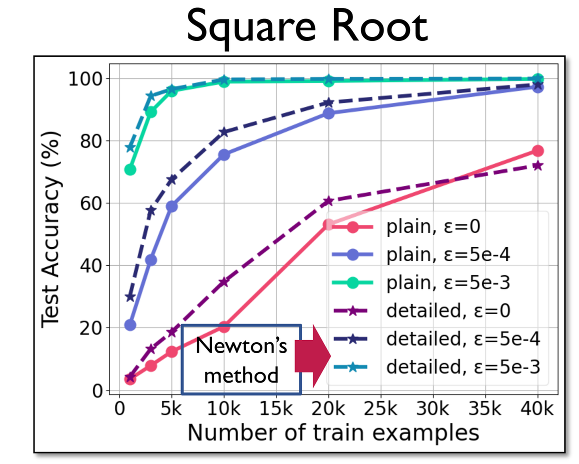
Multiplication



Sine



Square Root



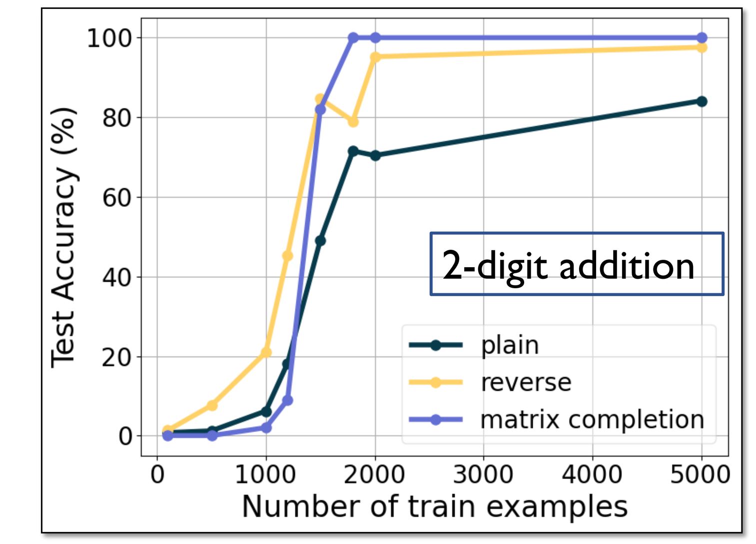
- Each operations has unique challenges (ex. negative, floating point)
- Scratchpad format helps for $(+, -, \times)$
- Intermediate step design is important.

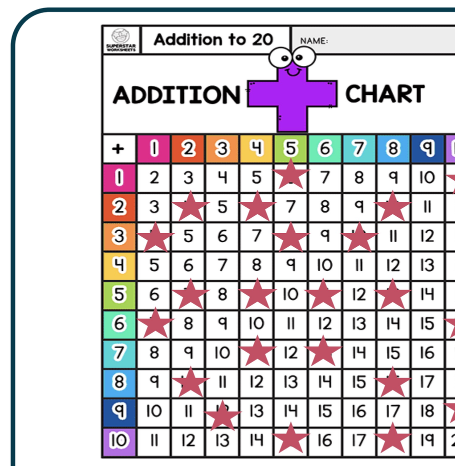
Key Take-aways

- Data formatting and sampling is important in performance and sample efficiency.
- Reverse, and scratchpad format allows the model to learn a simpler function.
- LRMC partially explains the emergence of addition (0→100% accuracy), but transformers generalize better.
- Length generalization is still challenging!

Connections to Matrix Completion

Why does addition emerge rapidly (0% → 100%)?





completing the addition map ≡ LRMC!

- Sharp phase transition occurs at $O(n)$ samples much like Low-Rank Matrix Completion (LRMC).
- But the transformer generalizes better than LRMC!

Excluding numbers from train data	No Exclusion		Excluding 100 numbers		Excluding 200 numbers		Excluding 500 numbers	
	Plain	Rev	Plain	Rev	Plain	Rev	Plain	Rev
Overall Accuracy	87.18%	99.97%	87.94%	100.00%	87.24%	99.99%	88.15%	99.99%
Exclusion Accuracy	-	-	92.55%	100.00%	92.15%	99.95%	90.85%	100%

- <u>Note</u>: NanoGPT can learn addition of unseen numbers / digits in train data (unlike LRMC)

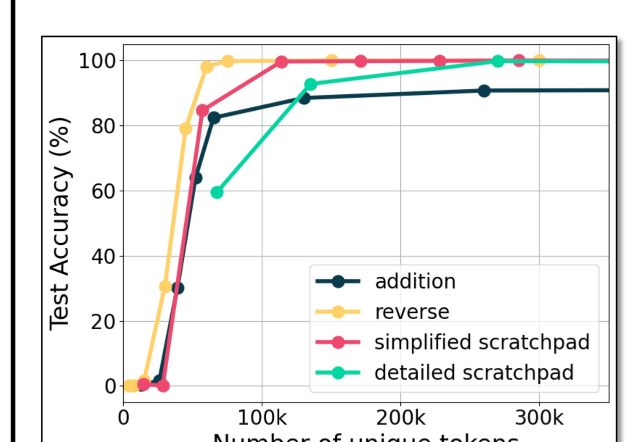
Scaling up, Token efficiency, and More!!

Does our findings hold for larger models?

- *Yes!* We fine-tune pretrained GPT-3 models of different scale

Addition	pretrained GPT-3	Finetuned with 1000 samples: Plain	Reverse	Simplified Scratchpad	Detailed Scratchpad
Davinci	2%	34%	80.9%	88.7%	**99.5%**
Curie	0.0%	1.4%	12.3%	10.7%	**99.7%**
Ada	0.0%	0.3%	6.3%	0.6%	**99.8%**

Efficiency in terms of "Tokens"?



# of tokens per example	Plain	Reverse	Simplified Scratchpad	Detailed Scratchpad
Prompt	8	9	23	23
Completion	5	6	41	258
Total	**13**	**15**	**64**	**281**

- Number of "tokens" in training data varies significantly by data format
- Reverse is most "token-efficient"

Many More in our Paper!

- Extension to higher digits
- Mixing arithmetic with text data
- Fine-tuning
- Few-shot prompting

References:

[1] Nye, et al. "Show Your Work: Scratchpads for Intermediate Computation with Language Models."
[2] Zhou, Hattie, et al. "Teaching Algorithmic Reasoning via In-context Learning."
[3] Kaiser, Łukasz, and Ilya Sutskever. "Neural gpus learn algorithms."
[4] Sutskever, Ilya, Oriol Vinyals, and Quoc V. Le. "Sequence to sequence learning with neural networks."