Exploring Length Generalization in Large Language Models

Cem Anil [*1,3], **Yuhuai Wu**[2], **Anders Andreassen**[1], **Aitor Lewkowycz**[1]
Vedant Misra[1], **Vinay Ramasesh**[1], **Ambrose Slone**[1], **Guy Gur-Ari**[1],
Ethan Dyer[1], **Behnam Neyshabur**[1]
[1] Google Research, Blueshift Team
[2] Google Research
[3] University of Toronto, Vector Institute

Abstract

The ability to extrapolate from short problem instances to longer ones is an important form of out-of-distribution generalization in reasoning tasks, and is crucial when learning from datasets where longer problem instances are rare. These include theorem proving, solving quantitative mathematics problems, and reading/summarizing novels. In this paper, we run careful empirical studies exploring the length generalization capabilities of transformer-based language models. We first establish that naively finetuning transformers on length generalization tasks shows significant generalization deficiencies independent of model scale. We then show that combining pretrained large language models' in-context learning abilities with scratchpad prompting (asking the model to output solution steps before producing an answer) results in a dramatic improvement in length generalization. We run careful failure analyses on each of the learning modalities and identify common sources of mistakes that highlight opportunities in equipping language models with the ability to generalize to longer problems.

1 Introduction

Many natural problems, such as theorem proving and program synthesis, have a notion of length that strongly correlates with the difficulty of the task. However, in these domains, the number of available problems typically drops rapidly as a function of problem length (e.g. Figure 2). Hence, it is desirable to learn from examples of shorter lengths to generalize to longer ones or at least reduce the number of samples required for longer examples. We refer to this type of problem as *length generalization*.

Recent work on large language models (LLMs) has shown consistent improvement in their performance by scaling model and dataset size. However, such models are still incapable of length generalization. For example, [1] shows that even though scale helps with solving arithmetic problems, scale alone is likely insufficient for learning to solve instances of arbitrary lengths. This implies that models fail to learn the general algorithms that would enable this kind of generalization. Indeed, Razeghi et al. [2] showed that the performance of LLMs on mathematical calculations correlates with term frequency in the training data. This suggests that LLMs might have gained their current performance from surface-level memorization instead of learning to apply the correct algorithm.

A recent line of work proposes to use a scratchpad, or chain-of-thought reasoning, when prompting LLMs [3, 4, 5] on multi-step tasks. Breaking down tasks into multiple small steps and presenting these steps to the model leads to improved performance across a variety of reasoning tasks including word problems, arithmetic, and code execution.

We perform a systematic study of length generalization with transformer-based large language models. We consider problems in which learning an algorithm can in principle enable a model to extrapolate

[*]Work done while interning at Google Research, Blueshift Team. Correspondence to: `anilcem@cs.toronto.edu`, `yuhuai@google.com`, `neyshabur@google.com`.

36th Conference on Neural Information Processing Systems (NeurIPS 2022).

In Distribution Lengths

```
# Length 3
x = True
y = False
if x == True:
      y = True
print(y)
```

```
# Length 4
a = True
b = False
if a == False:
      b = True
if b == True:
      a = False
print(a)
```

```
# Length 5
k = True
l = False
m = True
k = k and l
k = not k
print(k)
```

Out-of-Distribution Lengths

```
# Length 20
x = True
y = False
z = True
z = x and y
(...)
if x == y:
      z = true
z = z and x
if y == x:
      x = z
print(z)
```

Figure 1: **Examples of variable assignment problems:** Can transformer language models learn from short instances of the Variable Assignment task (left) to extrapolate to much longer instances (right)? **Length generalization** is the ability to learn from shorter/easier instances of a problem to handle longer/harder instances.

Techniques	In-distribution	Out-of-distribution	Improves with scale
Fine-tune	✓✓	✗	✗
Prompting	✗	✗	✗
Fine-tune + Prompting	✓✓	✗	✗
Fine-tune + Scratchpad	✓✓	✗	✗
Prompting + Scratchpad	✓	✓	✓
Fine-tune + Prompting + Scratchpad	✓✓	✓✓*	✓✓

Table 1: Performance on length generalization tasks of three techniques that language models admit: (1) Finetuning, (2) Prompting (or in-context few-shot learning) and (3) Scratchpad (Chain-of-Thought reasoning). We find that each technique (and the combinations thereof) have different modes of failure and present different trade-offs regarding in and out-of-distribution coverage. ✗ signifies poor ✓ signifies nontrivial, ✓✓ signifies near-perfect performance. (*) Refers to task-dependency.

from short examples to problems of arbitrary length. In particular, we focus on two simple algorithmic tasks, *parity* and *variable assignment*, in which the model needs to keep track of a state in order to extrapolate to longer lengths (see Figure 1). These problems are illuminating because their simplicity allows us to probe the failure modes as well as contrast the learned solutions with the ground truth algorithm. They provide us with a setting to study how/when these large language models start to fail.

We study combinations of three kinds of techniques for LLMs: finetuning, few shot prompting (also referred to as in-context learning), and use of a scratchpad (also referred to as chain-of-thought), to understand the role of each method and the interplay among the three in length generalization. Interestingly, we observe non-trivial interactions among the three techniques; see Table 1.

Contributions Our main contributions are as follows:

- We define and characterize the problem of length generalization using notions such as state tracking, execution depth, and per-step error rate. We study and carefully design two tasks, *parity* and *variable assignment*, that measure length generalization (Section 2).
- We find that in the finetuning regime, scaling data, model sizes, and compute does not improve length generalization (Section 3.1). We also observe that even when the model attains perfect in-distribution accuracy, it performs poorly in out-of-distribution domains. Surprisingly, different hyperparameter choices for finetuning have a large effect on length generalization performance, while having minimal effect on the final in-distribution performance (Section 3.3).
- We establish finetuning with scratchpad also fails to generalize to longer problems, in contrast to what is suggested by previous works [3]. We look into three potential failure cases: positional encoding, the presence of distractors, and end of token prediction, and conclude that distractors are the main culprit of failures for length generalization (Section 4).
- We show that in the in-context learning regime, use of a scratchpad shows a qualitatively different behavior and significantly alleviates the decay of performance on longer problems. This capability is significant, as it implies that for LLMs, there are certain skills, like length generalization, that can be learned through in-context learning rather than through finetuning even in the presence of infinite data. This is in stark contrast to the common norms of machine learning (Section 5).

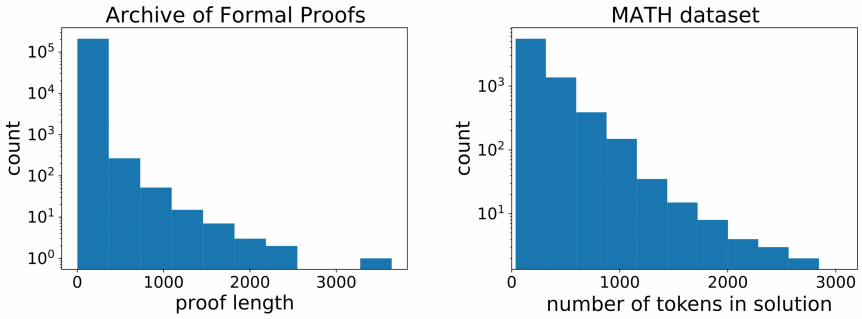

Figure 2: **Real world datasets have heavy tails in length: (left)** Histogram of lengths for proofs presented in the Archive of Formal Proofs **(right)** Histogram of the number of tokens for solutions in the MATH dataset. [6]

2 Length Generalization

Many sequence tasks—especially ones that require reasoning capabilities—have problem instances that differ in terms of their *lengths*. Shorter instances are often easier to state, process, and handle, and require less compute to find the answer. By contrast, longer instances are more challenging to parse and require more compute to solve. Tasks that have a *reasoning component* are especially well represented in this category — multi-hop reasoning [7], program execution [8], deductive reasoning [9] and theorem proving [10], to name a few. Note that having to deal with differing problem lengths poses two significant challenges. First, it is often the case that one encounters longer problem instances than the ones ever encountered during training, and is required to extrapolate. Second, even though longer problem instances have much more variety, real-world datasets often contain few long instances (see Figure 2). Both of these challenges are exacerbated if learning agents are not able to generalize across and beyond the lengths they learn from during training. This paper is about investigating to what extent transformer based language models are able to observe short problem instances and extrapolate to longer ones.

Instance Length as Number of Steps in a Markov Process It is possible to define *problem length* in many different ways to capture different aspects of problem difficulty. Does there exist a notion of length that would expose the same length-generalization-related problem structure observed in qualitatively very different settings? Such a framing would enable researchers to design algorithms and interventions that have the potential to generalize across a broad range of tasks. To this end, we take the approach of characterizing length in the context of a *deterministic Markov process*. From this perspective, length is simply the number of state transitions experienced by an initial world state. In other words, the data-generation process can be described as sampling an (1) *initial state* and a (2) *variable number of state transformations* to be applied sequentially on the initial state. The agent is provided both the initial state and the transformations, and is asked to predict the final state. This framing applies to a wide range of sequence problems, if not all of them—ranging from more mechanical tasks such as code and algorithm execution and theorem proving, to less structured tasks, such as solving math problems and summarizing novels.

In our empirical investigation we focus on two synthetic tasks: *parity* and *variable assignment*. These tasks avoid problem-specific subtleties that could mislead our analyses, while strongly capturing the deterministic Markov process structure.

2.1 Tasks

Parity: The parity task is an age-old learning problem that requires the trained agent to predict whether a bit-string has an even or odd number of ones in it. For example, the parity of the bitstring $[0, 1, 1, 0, 1]$ is "odd" (or 1) as opposed to "even" (or 0), because there is an odd number of 1s in the bit-string. The parity task admits a sequential solution that enables length generalization in a straightforward way: simply process the bits left-to-right and record the parity of the bits processed so far as the state. The default notion of length in the parity task is the number of bits in the input. However, we also experiment with a version where the number of bits is kept constant, and the number of 1s (i.e. the parity flipping bit) is systematically varied. The number of 1s stands for the number of state changes contained in the input bit-string, and actually appears to capture a more relevant notion of length for transformer models (see Section 3.1).

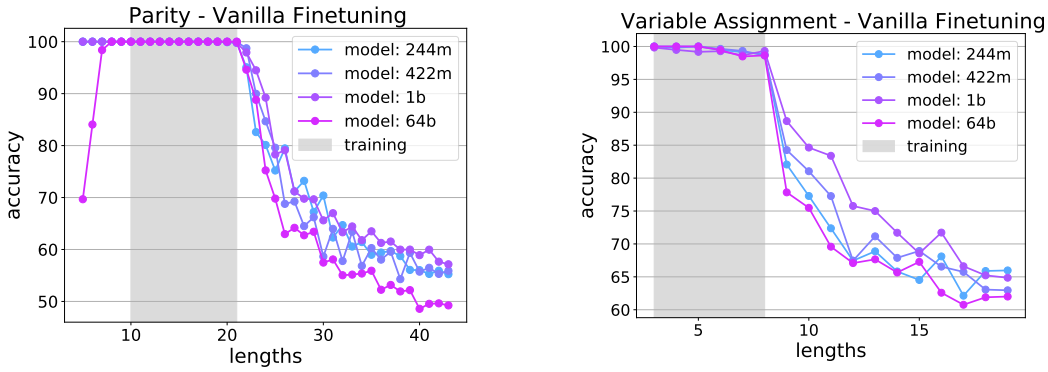

Figure 3: **Finetuned Length Generalization performance doesn't improve with scale:** Models of vastly different scales fail at length generalization on both Parity and Variable Assignment tasks, displaying identical generalization pathologies. The x-axis represents problem length and the y-axis represents the accuracy attained at that problem length. The training lengths are highlighted in grey.

Boolean Variable Assignment Task: The Boolean Variable Assignment task is designed to capture arbitrarily long, potentially branching unidirectional execution flows. An instance of this task can be seen in Figure 1. The inputs consist of semantically correct (i.e. bug-free) Python programs in which each line contains a boolean variable assignment operation. The output is simply the value of the variable presented in the final line of the program. The sequential solution to this task is to simply execute the program line by line while keeping track of the state of all variables.

The data generation procedure involves randomly generating execution flows that involve Boolean operations; see Supplementary Material (SM) for details.

We focus our evaluations on two variants of this dataset. (1) The *diverse variable assignment* split consists of a wide range of boolean operators available and is intended to contain maximally diverse programs. (2) The *chain-like variable assignment split* consists only of operations that compose the values of already defined variables. This results in long chains of dependencies between the initial values of the variables and the queried one, ensuring that there are almost no redundant operations in the program (i.e. operations that can be removed without affecting the output of the program). This split emphasizes the sequential nature of the variable assignment problem.

3 Standard Finetuning Fails at Length Generalization

We begin by demonstrating that finetuning transformer models on length-generalization tasks results in poor out-of-distribution performance. In experiments we use `LaMDA` [2] decoder-only models. These checkpoints were trained using general natural language data. We use the AdaFactor optimizer [11] during finetuning, and tune the learning rate, batch size and dropout. We trained the networks until the in-distribution validation accuracy settles (20000 gradient steps for parity and 18000 gradient steps for variable assignment). The loss was only computed on the target tokens (i.e. the model wasn't trained to model the input questions).

3.1 Scale Doesn't Improve Length Generalization

Parity: We finetuned four pretrained `LaMDA` models with 244m, 422m, 1b and 64b parameters on the parity task, where the training distribution included randomly sampled bitstrings of length 10 to 21. We then evaluated the performance on bitstrings of length 3 to 40; see Figure 3. We find that model scale has a little effect on length generalization.

Variable Assignment: We finetuned the same models on the *chain-like* Variable Assignment Task, described in Section 2. We kept the in-distribution lengths at 3 to 11, and evaluated the test performance on lengths 3 to 19. The results can be seen in Figure 3. Just like in the parity task, while the in-distribution performance is (near) perfect, out-of-distribution performance degrades rapidly as length increases. To get a sense of just how weak the out-of-distribution performance is, we also trained a 422m model on the same dataset, except we shuffled the operations before feeding it to the

[2] We omit the name of the model for the purpose of anonymization.

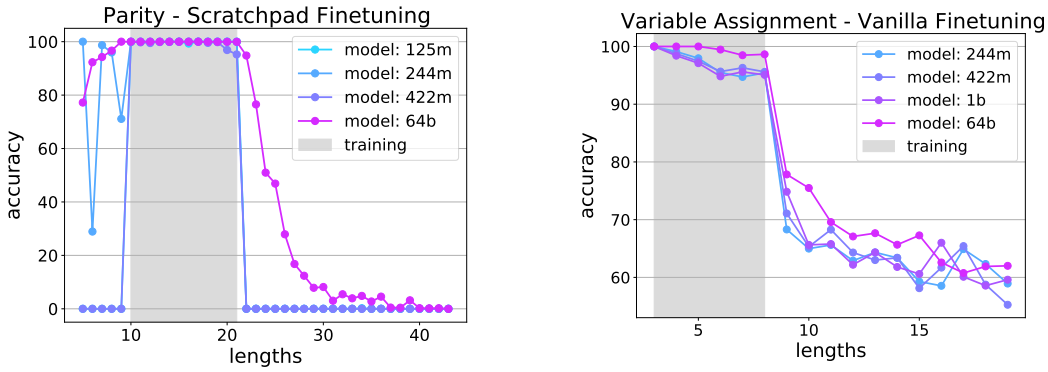

Figure 4: **Scratchpad finetuning displays poor length generalization:** Scratchpad finetuning displays qualitatively similar length generalization pathologies as vanilla finetuning. The x-axis represents problem length and the y-axis represents the accuracy attained at that problem length. The training lengths are highlighted in grey.

model. This removes the sequential dependency between the operations, and helps us establish a strong baseline that only predicts the answers based on non-sequential, spurious correlations. The accuracy-length curves for the baseline can be found in SM.

3.2 Transformers Prefer Parallel Strategies over Sequential Ones

The results presented in Section 3.1 establish that, when presented with sequential length generalization problems, transformers are biased toward learning non-sequential "shortcut" solutions that fail at longer problem instances. We ran additional experiments to gain a better understanding of the nature of this generalization pattern.

On parity, we ran finetuning on a different distribution of bit-strings: Instead of first randomly sampling the number of bits in the input bit-string, then sampling the values of the bits, we fixed the total number of bits in the input, and only varied the *number of ones* in the bit-string uniformly. We trained with 10 to 20 ones in the input distribution and tested on an interval containing 1 to 30 ones. This makes sure that the number of tokens (now fixed at 30) is now disambiguated from number of state changes, which for parity is equal to the number of ones. The difference between in and out-of-distribution performance is even starker for this data distribution (Figure 5): while in-distribution performance was 100%, OOD performance was roughly equivalent to random prediction[3]. This suggests that the transformers are learning a non-sequential solution that involves counting the number of ones in the input, and then thresholding the output. This is not surprising, given that self-attention is an equivariant transformation capable of performing pooling operations like max-pooling [12]. This strategy doesn't allow for knowledge transfer between problems of different lengths. Note that this bottom-up counting behaviour is complementary to the left-to-right counting behaviour displayed by recurrent models Suzgun et al. [13].

On the variable assignment dataset, we finetuned a 255m `LaMDA` model on the *diverse* split of the variable assignment dataset of programs up to 16 lines, and evaluated on the same data generating distribution up to 32 lines. We measured the evolution of the model's accuracy with respect to training iterations on different program lengths (quantified by number of lines). The results are in SM.

We again observed that a different notion of length (which we call *computational graph depth*) captures the difficulty of problem instances better than number of program operations. A variable assignment program can be represented as a computational graph where each node corresponds to a variable, and each edge corresponds to an operation. Computational graph depth is the length of the longest dependency chain that connects to the queried variable node. This notion of length corresponds to the highly parallelizable strategy of executing programs by iteratively resolving computational graph dependencies. We present two results that suggest that computational graph depth is a more relevant notion of length for transformers. (1) Inspecting the order of problem instances in which the trained transformer correctly solves this task, we find that performance is

[3]The periodic 0% and near 100% performance on OOD lengths is due to the models' tendency to output 0 or 1 depending on whether the input has a significantly higher ratio of 0s or 1s. On average, the accuracy on OOD length is not better than random guess.

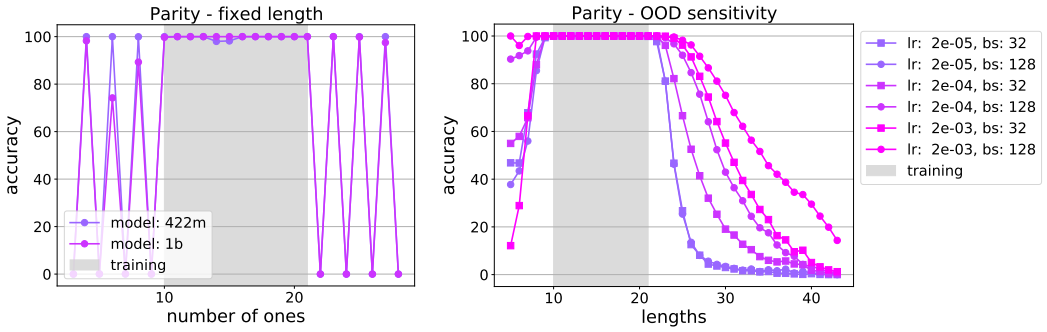

Figure 5: **(left) Complete lack of length generalization:** Transformers trained on the parity task have difficulty generalizing to bit-strings that have a different number of $1s$. **(right) Sensitivity to hyperparameters:** Trained networks sharing architecture, data and in-distribution loss can have very different length generalization performances. *lr* stands "learning rate" and *bs* stands for "batch size".

strongest on examples with small computational graph depth, even if these examples are long in terms of number of operations. (2) The transformer does a good job of handling programs with an out-of-distribution number of operations, but for which computational graph depth is in-distribution.

3.3 In-Distribution Generalization Doesn't Predict OOD Generalization on Length Generalization Tasks

Prior work on out-of-distribution generalization establishes that in many tasks, in-distribution loss is a strong predictor of out-of-distribution generalization [14]. Our experiments on the parity task indicate that the distribution shift induced by changing problem lengths falls outside of the this category. Figure 5 shows how the same model trained on the same data achieving roughly the same in-distribution cross entropy loss behaves on OOD data, where the difference is solely induced by the choice of different hyperparameters.

4 Scratchpad Finetuning Still Fails at Length Generalization

It has been shown in prior work that it's possible to get pretrained LLMs to solve a given task by not only outputting the answer, but also the solution steps behind it. Nye et al. [15] use scratchpad finetuning to achieve strong in-distribution performance on execution based tasks such as code execution and computing polynomials. While they also report modest length generalization results on integer arithmetic, we find that scratchpad finetuning suffers from similar length generalization pathologies than vanilla finetuning does. The results on parity and variable assignment tasks can be seen in Figure 4. The precise scratchpad strategies used for these tasks are described in detail in SM.

Error analysis: To understand the causes of failure in training scratchpad strategies, we focused on two architectural choices that could account for the poor performance: (1) how transformers encode position information, and (2) whether the transformers are trained to predict an end-of-sequence (EOS) token. `LaMDA` models use T5 position biases [16] to handle position information. If the network is only trained with short instances, position biases that handle longer positional distances might not be trained, explaining poor length generalization. Similarly, Newman et al. [17] report that networks trained with EOS token prediction often suffer from generalizing to longer problem instances, because of the models' tendency to emit EOS tokens prematurely, as well as the EOS tokens' effect on the representations that get learned.

We tested the extent to which these effects can explain lack of length generalization as follows. We padded both the input bit-strings and the scratchpad content with dummy padding tokens to make the token count the same. We also augmented the input and scratchpad targets with the same number of padding tokens on the left and right so that the relevant bit to attend to when executing the sequential scratchpad strategy corresponds to the same T5 position bias bin. Examples of the updated input-target pairs can be seen in SM. While this intervention helps, the trained models still display significant length generalization issues.

To gain further insight about the source of the problem, we plotted how the scratchpad target prediction error rates change as a function of (1) how far along one is in constructing the scratchpad, and (2)

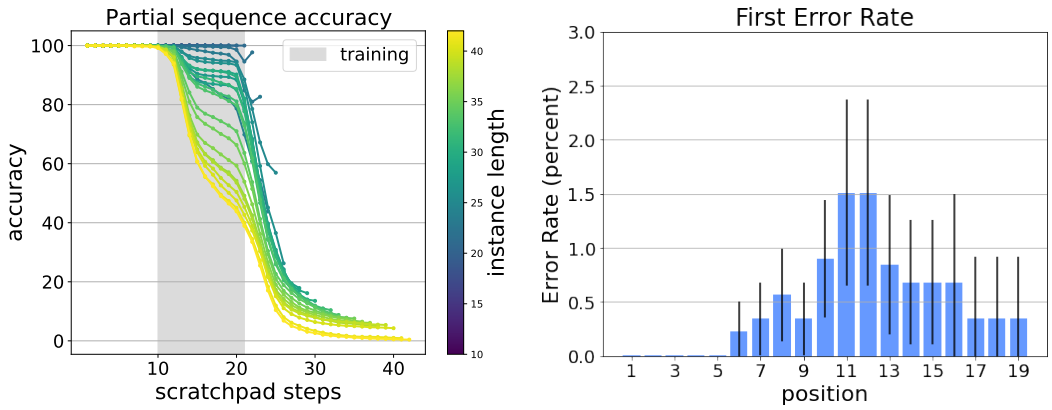

Figure 6: **(left) Effect of input length on per-step scratchpad accuracy:** Points corresponds to the accuracy (y-axis) of the first x scratchpad steps (x-axis) on parity instances of variable length (color). If the input length is out-of-distribution, even in-distribution scratchpad steps are inaccurate, implying the model hasn't learned an attention pattern that generalizes to longer bit-strings. **(right) Roughly constant per-step error rate:** The per-step error rates of the `LaMDA` 128b model, few-shot finetuned on the coin-flip version of the parity task remain roughly constant across the scratchpad steps. This is in stark contrast with zero-shot scratchpad finetuned models, where the per-step error rates increase abruptly when the model is evaluated on OOD lengths.

the length of the input bit-string. The results can be seen in Figure 6. The fact that the model makes mistakes in in-distribution scratchpad steps when the input has an OOD length implies that the attention mechanism isn't capturing the relevant part of the input to form the scratchpad output. See SM for additional analysis.

5 Scratchpad Prompting Significantly Improves Length Generalization

Wei et al. [4], Nye et al. [15] and Lewkowycz et al. [5] showed that combining prompting (i.e. in-context learning) with scratchpad strategies present a powerful combination. They demonstrate that pretrained LLMs, without the help of any finetuning, can solve grade school math word problems and execute pieces of code with nontrivial correctness [15], when prompted with the right scratchpad strategy. We corroborate these findings, and report that scratchpad prompting endows pretrained LLMs with the capability of *variable length template matching* (see Figure 8). That is, in-context learning enables the model to "learn" solution steps from a small number of short instances, and apply the same template on significantly longer instances with a high degree of accuracy.

5.1 Few-shot scratchpad

Contrary to vanilla and scratchpad finetuning, we find that under the right conditions, few-shot scratchpad strategies sometimes significantly improves LLMs' capability to extrapolate to lengths much further than what pretraining weights grant them.

To evaluate the performance of few-shot conditioning with scratchpad inputs without any finetuning, we phrase the parity problem in natural language as a coin flipping task. An example for the few-shot prompts we used can be seen in Figure 8. Wei et al. [4] also report results on the coin-flip task: the scratchpad format we used differs from theirs in that while ours respects the sequential nature of the task (i.e. each coin flip corresponds to a step in the scratchpad solution), Wei et al. [4]'s scratchpad strategy involves summing up the number of coin flips, then deciding on the final output based on the evenness/oddness of the sum. Also, while they only test up to 4 flips, we go up to 20 flips while still attaining highly nontrivial accuracy levels.

For the variable assignment task, our scratchpad strategy involves copying over the program that's being executed, with comments added in between lines specifying the value of the variable that was assigned in the line above. Instances of this scratchpad strategy can be seen in SM.

Figure 7 shows the performance of the pretrained `LaMDA` 128b model on the coin-flip version of the parity task. Figure 8 shows an instance of how a length 3 prompt can induce the model to correctly output a 20 step scratchpad. We find that with the right scratchpad prompt, LLMs are able

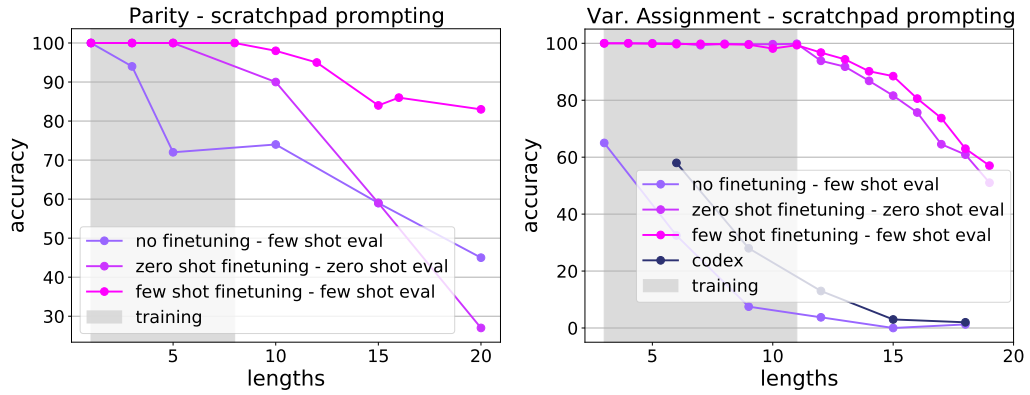

Figure 7: **Few-shot finetuning with scratchpad** displays qualitatively different behaviour on parity and variable assignment tasks. On parity, where the non-finetuned model already performs very well, few-shot-finetuning with scratchpad leads to a significant performance boost over zero-shot finetuning with scratchpad. On variable assignment, where the base model doesn't perform poorly, there's not a significant gap between few-shot finetuning and zero-shot finetuning with scrathpad. The performance of OpenAI's Codex model [18] on the variable assignment task is also provided.

to generate correct scratchpad solutions. This reduces the problem to simply filling in the content of the generation correctly by inferring the right state transitions without having to figure out how to extrapolate the solution template.

Few-Shot Finetuning with Scratchpad Strategies: Does combining finetuning, few-shot prompting, and scratchpad strategies improve length generalization?

We find that the answer is **yes** in the case of parity. As seen in Figure 7, few-shot finetuning performs significantly better than the baseline model, both on in- and out-of-distribution lengths. Note that the vanilla (i.e. no shot) finetuning baseline also outperforms the no-finetuning baseline, it actually does worse on the larger lengths — a pathology that doesn't appear with few-shot finetuning.

The results point to a qualitatively different picture for the variable assignment task. Both few-shot finetuning and vanilla finetuning result in similar length generalization behavior (Figure 7). We hypothesize that this distinction is caused by the different pretrained performances that the model displays on these tasks: while length generalization is already strong with no finetuning on parity, that's not the case for variable assignment. In the latter case, the model is forced to acquire a new skill via finetuning, which displays the same pathologies as zero-shot finetuning with scratchpad. As a sanity check, we evaluated the (few-shot) finetuned performance of the pretrained model on an alternative, synthetic prompt style that yields poor performance without any pretraining: As expected by the aforementioned hypothesis, we observed that the few-shot finetuned model on this task also shows significant length generalization pathologies. The results can be found in SM. We leave a more rigorous evaluation of this hypothesis as future work.

6 Related Works

There have been many attempts to study generalization from shorter/easier to longer/harder examples.

Challenges in length generalization: Several existing works have investigated pathologies that arise when models are asked to generalize to processing and generating longer (measured by number of tokens) sequences. Newman et al. [17] find that sequence models trained with and in the absence of the end-of-sequence token display qualitatively different length extrapolation behaviour and learn different representations. Dubois et al. [19] proposes modifications to the commonly used dot-product attention to improve the models' ability to extrapolate to longer sequences. Murray and Chiang [20] demonstrate that neural machine translation models tend to have a bias towards generating shorter-than-desired translations. Yehudai et al. [21] show that length generalization issues are also present in training graph neural networks, where extrapolating across graph size presents a challenge. Ju et al. [22] propose a new attention mechanism to facilitate recurrent processing in transformer models. Press et al. [23] propose modifying transformer attention biases to facilitate generalization beyond the training context length. Concurrent work [24] propose a synthetic dataset named LEGO (Learning Equality and Group Operations), an instantiation of which resembles our variable assignment task

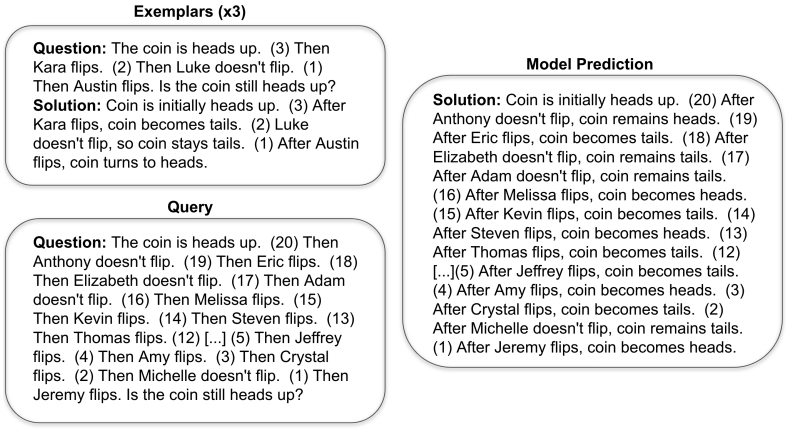

Figure 8: **Few-shot length generalization:** The largest LaMDA model is able to map the scratchpad solution template from a few short exemplars onto much longer queries.

where the only boolean operations allowed are *assign* and *negate and assign*, and overriding the values of variables is not allowed. Their analyses on OOD generalization largely complement ours: while we focus on decoder-only architectures and scratchpad strategies as a way of carrying over state, they focus on encoder-only architectures, and investigate the effect of weight-sharing.

Easy-to-Hard generalization: Schwarzschild et al. [25] and Bansal et al. [26] use weight-tied neural networks to generalize from easy to hard examples. Schwarzschild et al. [25] also provide three tasks to benchmark easy-to-hard generalization. Dehghani et al. [27] and Kaiser and Sutskever [28] assess the capabilities of their proposed architectures on easy-to-hard generalization problems.

Inductive Biases Related to Lenght Generalization: McCoy et al. [29] study the inductive bias of seq-to-seq learners on English question formation and English tense reinflection tasks and find that LSTM and GRU networks often display differing strategies, caused by the use of differing activation functions. Suzgun et al. [13] find that recurrent networks can perform dynamical counting, and encode hierarchical representations, which enables them to solve nontrivial Dyck tasks using k-counters. Kharitonov and Chaabouni [30] also study the inductive bias of different architectures, and conclude that transformer and LSTM architectural have a tendency to learn hierarchical strategies, whereas CNN based strategies display more linear structure. He et al. [31] propose a method to learn natural inference models that are not biased on spurious correlations. McCoy et al. [32] show that transformer models that display strong performance in natural language inference can have superficial biases that fool them in systematic ways and proposes a framework to think about these biases.

7 Conclusion

The ability to learn from shorter/easier problem instances to generalize to longer/harder ones is a key capability in a large number of tasks, especially ones requiring reasoning. We defined the concept of *length generalization* and measured language models' length generalization capabilities. After conducting careful experiments using finetuning, scratchpads, and few-shot prompting, we reached the following conclusions: (1) Generalizing in length is a challenge for language models at least up to the 100B parameter scale. Both vanilla finetuning and finetuning with scratchpads suffer from a lack of length generalization caused by models' tendency to pick up non-sequential pattern that don't apply to longer problem instances. (2) Few-shot scratchpad prompting enables pretrained large language models to pick up scratchpad-templates that extrapolate to arbitrary lengths, leading to dramatic improvements on longer problem instances. Unlike raw finetuning, this approach does scale with model size [4]. (3) Trying to further enhance the performance of few-shot scratchpad prompted LLMs via finetuning yields mixed results, depending on the non-finetuned performance of the base model at the target task. We emphasize that the aforementioned few-shot variable length pattern matching capability — something that doesn't require changing model architecture — offers a qualitatively different approach to handle length generalization in contrast to prior art that introduced architectural modifications to achieve the same goal. This capability is also significant in that it implies that for LLMs, there are certain skills, like length generalization, that can be learned better through in-context learning rather than through finetuning, even in the presence of infinite data.

References

[1] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901, 2020.

[2] Yasaman Razeghi, Robert L Logan IV, Matt Gardner, and Sameer Singh. Impact of pretraining term frequencies on few-shot reasoning. *arXiv preprint arXiv:2202.07206*, 2022.

[3] Maxwell Nye, Anders Johan Andreassen, Guy Gur-Ari, Henryk Michalewski, Jacob Austin, David Bieber, David Dohan, Aitor Lewkowycz, Maarten Bosma, David Luan, et al. Show your work: Scratchpads for intermediate computation with language models. *arXiv preprint arXiv:2112.00114*, 2021.

[4] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Ed Chi, Quoc Le, and Denny Zhou. Chain of thought prompting elicits reasoning in large language models, 2022. URL https://arxiv.org/abs/2201.11903.

[5] Aitor Lewkowycz, Anders Andreassen, David Dohan, Ethan Dyer, Henryk Michalewski, Vinay Ramasesh, Ambrose Slone, Cem Anil, Imanol Schlag, Theo Gutman-Solo, et al. Solving quantitative reasoning problems with language models. *arXiv preprint arXiv:2206.14858*, 2022.

[6] Dan Hendrycks, Collin Burns, Saurav Kadavath, Akul Arora, Steven Basart, Eric Tang, Dawn Song, and Jacob Steinhardt. Measuring mathematical problem solving with the math dataset. *arXiv preprint arXiv:2103.03874*, 2021.

[7] Haoyu Wang, Mo Yu, Xiaoxiao Guo, Rajarshi Das, Wenhan Xiong, and Tian Gao. Do multi-hop readers dream of reasoning chains? *arXiv preprint arXiv:1910.14520*, 2019.

[8] Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, et al. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*, 2021.

[9] Peter Clark, Oyvind Tafjord, and Kyle Richardson. Transformers as soft reasoners over language. *arXiv preprint arXiv:2002.05867*, 2020.

[10] Yuhuai Wu, Albert Qiaochu Jiang, Jimmy Ba, and Roger Grosse. Int: An inequality benchmark for evaluating generalization in theorem proving. *arXiv preprint arXiv:2007.02924*, 2020.

[11] Noam Shazeer and Mitchell Stern. Adafactor: Adaptive learning rates with sublinear memory cost. In *International Conference on Machine Learning*, pages 4596–4604. PMLR, 2018.

[12] Juho Lee, Yoonho Lee, Jungtaek Kim, Adam Kosiorek, Seungjin Choi, and Yee Whye Teh. Set transformer: A framework for attention-based permutation-invariant neural networks. In *International Conference on Machine Learning*, pages 3744–3753. PMLR, 2019.

[13] Mirac Suzgun, Sebastian Gehrmann, Yonatan Belinkov, and Stuart M Shieber. Lstm networks can perform dynamic counting. *arXiv preprint arXiv:1906.03648*, 2019.

[14] Vaishnavh Nagarajan, Anders Andreassen, and Behnam Neyshabur. Understanding the failure modes of out-of-distribution generalization. *arXiv preprint arXiv:2010.15775*, 2020.

[15] Maxwell Nye, Anders Johan Andreassen, Guy Gur-Ari, Henryk Michalewski, Jacob Austin, David Bieber, David Dohan, Aitor Lewkowycz, Maarten Bosma, David Luan, et al. Show your work: Scratchpads for intermediate computation with language models. *arXiv preprint arXiv:2112.00114*, 2021.

[16] 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. *arXiv preprint arXiv:1910.10683*, 2019.

[17] Benjamin Newman, John Hewitt, Percy Liang, and Christopher D Manning. The eos decision and length extrapolation. *arXiv preprint arXiv:2010.07174*, 2020.

[18] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

[19] Yann Dubois, Gautier Dagan, Dieuwke Hupkes, and Elia Bruni. Location attention for extrapolation to longer sequences. *arXiv preprint arXiv:1911.03872*, 2019.

[20] Kenton Murray and David Chiang. Correcting length bias in neural machine translation. *arXiv preprint arXiv:1808.10006*, 2018.

[21] Gilad Yehudai, Ethan Fetaya, Eli Meirom, Gal Chechik, and Haggai Maron. From local structures to size generalization in graph neural networks. In *International Conference on Machine Learning*, pages 11975–11986. PMLR, 2021.

[22] Da Ju, Stephen Roller, Sainbayar Sukhbaatar, and Jason Weston. Staircase attention for recurrent processing of sequences. *arXiv preprint arXiv:2106.04279*, 2021.

[23] 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.

[24] Yi Zhang, Arturs Backurs, Sébastien Bubeck, Ronen Eldan, Suriya Gunasekar, and Tal Wagner. Unveiling transformers with lego: a synthetic reasoning task. *arXiv preprint arXiv:2206.04301*, 2022.

[25] Avi Schwarzschild, Eitan Borgnia, Arjun Gupta, Furong Huang, Uzi Vishkin, Micah Goldblum, and Tom Goldstein. Can you learn an algorithm? generalizing from easy to hard problems with recurrent networks. *Advances in Neural Information Processing Systems*, 34, 2021.

[26] Arpit Bansal, Avi Schwarzschild, Eitan Borgnia, Zeyad Emam, Furong Huang, Micah Goldblum, and Tom Goldstein. End-to-end algorithm synthesis with recurrent networks: Logical extrapolation without overthinking. *arXiv preprint arXiv:2202.05826*, 2022.

[27] Mostafa Dehghani, Stephan Gouws, Oriol Vinyals, Jakob Uszkoreit, and Łukasz Kaiser. Universal transformers. *arXiv preprint arXiv:1807.03819*, 2018.

[28] Łukasz Kaiser and Ilya Sutskever. Neural gpus learn algorithms. *arXiv preprint arXiv:1511.08228*, 2015.

[29] R Thomas McCoy, Robert Frank, and Tal Linzen. Does syntax need to grow on trees? sources of hierarchical inductive bias in sequence-to-sequence networks. *Transactions of the Association for Computational Linguistics*, 8:125–140, 2020.

[30] Eugene Kharitonov and Rahma Chaabouni. What they do when in doubt: a study of inductive biases in seq2seq learners. *arXiv preprint arXiv:2006.14953*, 2020.

[31] He He, Sheng Zha, and Haohan Wang. Unlearn dataset bias in natural language inference by fitting the residual. *arXiv preprint arXiv:1908.10763*, 2019.

[32] R Thomas McCoy, Ellie Pavlick, and Tal Linzen. Right for the wrong reasons: Diagnosing syntactic heuristics in natural language inference. *arXiv preprint arXiv:1902.01007*, 2019.

[33] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *The journal of machine learning research*, 15(1):1929–1958, 2014.

Checklist

1. For all authors...
   (a) Do the main claims made in the abstract and introduction accurately reflect the paper's contributions and scope? [Yes]
   (b) Did you describe the limitations of your work? [Yes]
   (c) Did you discuss any potential negative societal impacts of your work? [Yes]
   (d) Have you read the ethics review guidelines and ensured that your paper conforms to them? [Yes]
2. If you are including theoretical results...
   (a) Did you state the full set of assumptions of all theoretical results? [N/A]
   (b) Did you include complete proofs of all theoretical results? [N/A]
3. If you ran experiments...
   (a) Did you include the code, data, and instructions needed to reproduce the main experimental results (either in the supplemental material or as a URL)? [No] The code and the data are proprietary.
   (b) Did you specify all the training details (e.g., data splits, hyperparameters, how they were chosen)? [Yes]
   (c) Did you report error bars (e.g., with respect to the random seed after running experiments multiple times)? [No]
   (d) Did you include the total amount of compute and the type of resources used (e.g., type of GPUs, internal cluster, or cloud provider)? [No]
4. If you are using existing assets (e.g., code, data, models) or curating/releasing new assets...
   (a) If your work uses existing assets, did you cite the creators? [N/A]
   (b) Did you mention the license of the assets? [N/A]
   (c) Did you include any new assets either in the supplemental material or as a URL? [N/A]
   (d) Did you discuss whether and how consent was obtained from people whose data you're using/curating? [N/A]
   (e) Did you discuss whether the data you are using/curating contains personally identifiable information or offensive content? [N/A]
5. If you used crowdsourcing or conducted research with human subjects...
   (a) Did you include the full text of instructions given to participants and screenshots, if applicable? [N/A]
   (b) Did you describe any potential participant risks, with links to Institutional Review Board (IRB) approvals, if applicable? [N/A]
   (c) Did you include the estimated hourly wage paid to participants and the total amount spent on participant compensation? [N/A]

Input: >>> 1 1 0 1 1 0 1 1 ==

Scratchpad: 1 0 0 1 0 0 1 0

Target: 0

Input: >>> - - - 1 1 0 1 1 0 1 1 - - ==

Scratchpad: - - - 1 0 0 1 0 0 1 0 - -

Target: 0

Figure 9: **Parity problem instances: (left)** A sample 8-bit parity problem instance, along with the scratchpad targets. The scratchpad represent the intermediate parity state as the sequence is processed left to right. **(right)** A parity problem instance with the padded scratchpad strategy. Both the input and the scratchpad targets are padded left and right with the same number of padding tokens, such that the relevant bit to attend to while constructing the scratchpad bits is always equidistant even when there are different number of bits in the input.

A Data Generation Details

We describe the data generation procedures for the parity and variable assignment tasks in detail.

A.1 Parity Datasets:

Synthetic Parity Dataset: A 8-bit example of the synthetic parity example, along with the corresponding scratchpad targets, can be seen in Figure 9. We added the prefix "> > >" to signify the start of the parity sequence, and the suffix "==" to signify the start of the target or scratchpad tokens. There's no special meaning associated with the particular prefixes and suffixed used.

We experimented with two version of the synthetic parity dataset: In one split, we varied the *number of bits* in the input, and in the other one, we varied the *number of ones*.

- **Varied number of bit split:** To generate the samples in this split, we first sampled the number of bits, then sampled each bit individually from a uniform Bernoulli distribution. For training, we used lengths between 3 and 20, and for validation/testing, we used lengths between 3 and 40.
- **Varied number of ones split:** Here, we fixed the number of bits at 30. To sample each instance, we first uniformly sampled the number of ones, then randomly placed each one in the fixed-length bitstring by randomly shuffling the bits. We used 10 to 20 ones in the training split, and 1 to 30 ones in the validation/test splits.

  Padded scratchpad: The padded scratchpad format can be seen in 9. Both the input and the scratchpad targets are padded left and right with the same number of padding tokens respectively, such that the relevant bit to attend to while constructing the scratchpad bits is always equidistant. Moreover the total number of characters/tokens is also kept constant. The number of tokens to pad on the left and right is determined (uniformly) randomly.

The parity datasets contain 1000000 samples.

Natural Language Parity Dataset: In order to tap into the natural language understanding capabilities of pretrained language models, we situated the parity task as a "coin flip problem". In this framing, flipping a coin corresponds to 1 and not flipping a coin corresponds to 0. To make the inputs as close as possible to English without occupying too many tokens, we used the sentence templates "Then <NAME> flips." and "Then <NAME> doesn't flip." to represent whether the coin was flipped or not respectively, where "<NAME>" refers to a randomly sampled given name. We also prepended each step with an integer id that count backwards from the total number of steps there are in the input sequence. We've experimented with versions where the integer ids are incremented. This didn't lead to a significant difference in the overall performance.

Two representative sample input-target pairs (including the exemplars) are provided in Figure 10.

A.2 Boolean Variable Assignment Dataset:

An instance of the variable assignment dataset can be seen in Figure 11. The data generation procedure is aimed at synthesizing large number of qualitatively different programs: (1) A subset of boolean variable assignment operations is uniformly sampled from a large pool of operations. (2) The number

Input:
Question The coin is heads up. (4) Then Williams flips. (3) Then Ward flips. (2) Then Valentine doesn't flip. (1) Then Son doesn't flip. Is the coin still heads up?\nSolution Coin is initially heads up. (4) After Williams flips, coin turns to tails. (3) After Ward flips, coin becomes heads. (2) Valentine doesn't flip, so coin stays heads. (1) Son doesn't flip, so coin remains heads. DONE
#
Question The coin is heads up. (4) Then Shade flips. (3) Then Kong doesn't flip. (2) Then Kodi flips. (1) Then Charleston flips. Is the coin still heads up?\nSolution Coin is initially heads up. (4) After Shade flips, coin turns to tails. (3) Kong doesn't flip, so coin stays tails. (2) After Kodi flips, coin becomes heads. (1) After Charleston flips, coin turns to tails. DONE
#
Question The coin is heads up. (4) Then Ka doesn't flip. (3) Then Justice flips. (2) Then Johan flips. (1) Then Jamaica flips. Is the coin still heads up?\nSolution Coin is initially heads up. (4) Ka doesn't flip, so coin remains heads. (3) After Justice flips, coin becomes tails. (2) After Johan flips, coin turns to heads. (1) After Jamaica flips, coin becomes tails. DONE
#
Question The coin is heads up. (4) Then Cy flips. (3) Then Booker flips. (2) Then Ace doesn't flip. (1) Then Ren flips. Is the coin still heads up?\nSolution Coin is initially heads up. (4) After Cy flips, coin turns to tails. (3) After Booker flips, coin becomes heads. (2) Ace doesn't flip, so coin stays heads. (1) After Ren flips, coin turns to tails. DONE
#
Question The coin is heads up. (6) Then Tristan flips. (5) Then Hillary flips. (4) Then Olivia flips. (3) Then Rosa doesn't flip. (2) Then Kurt flips. (1) Then Glenn doesn't flip. Is the coin still heads up?"

Scratchpad targets:
"Solution Coin is initially heads up. (6) After Tristan flips, coin becomes tails. (5) After Hillary flips, coin turns to heads. (4) After Olivia flips, coin becomes tails. (3) Rosa doesn't flip, so coin remains tails. (2) After Kurt flips, coin turns to heads. (1) Glenn doesn't flip, so coin stays heads. DONE

Input:
Question The coin is heads up. (4) Then Katarina doesn't flip. (3) Then January flips. (2) Then Duke flips. (1) Then Cal doesn't flip. Is the coin still heads up?\nSolution Coin is initially heads up. (4) Katarina doesn't flip, so coin remains heads. (3) After January flips, coin becomes tails. (2) After Duke flips, coin turns to heads. (1) Cal doesn't flip, so coin stays heads. DONE
#
Question The coin is heads up. (4) Then Berry flips. (3) Then Abbi flips. (2) Then Tam flips. (1) Then Ikea doesn't flip. Is the coin still heads up?\nSolution Coin is initially heads up. (4) After Berry flips, coin becomes tails. (3) After Abbi flips, coin turns to heads. (2) After Tam flips, coin becomes tails. (1) Ikea doesn't flip, so coin remains tails. DONE
#
Question The coin is heads up. (4) Then Cain doesn't flip. (3) Then Woody flips. (2) Then Von doesn't flip. (1) Then Thu doesn't flip. Is the coin still heads up?\nSolution Coin is initially heads up. (4) Cain doesn't flip, so coin stays heads. (3) After Woody flips, coin turns to tails. (2) Von doesn't flip, so coin remains tails. (1) Thu doesn't flip, so coin stays tails. DONE
#
Question The coin is heads up. (4) Then Russ flips. (3) Then Williams flips. (2) Then Ward doesn't flip. (1) Then Valentine flips. Is the coin still heads up?\nSolution Coin is initially heads up. (4) After Russ flips, coin becomes tails. (3) After Williams flips, coin turns to heads. (2) Ward doesn't flip, so coin remains heads. (1) After Valentine flips, coin becomes tails. DONE
#
Question The coin is heads up. (8) Then Melissa flips. (7) Then Kevin doesn't flip. (6) Then Steven flips. (5) Then Thomas flips. (4) Then Timothy doesn't flip. (3) Then Kyle doesn't flip. (2) Then Rachel doesn't flip. (1) Then Laura doesn't flip. Is the coin still heads up?

Scratchpad targets:
Solution Coin is initially heads up. (8) After Melissa flips, coin turns to tails. (7) Kevin doesn't flip, so coin stays tails. (6) After Steven flips, coin becomes heads. (5) After Thomas flips, coin turns to tails. (4) Timothy doesn't flip, so coin remains tails. (3) Kyle doesn't flip, so coin stays tails. (2) Rachel doesn't flip, so coin remains tails. (1) Laura doesn't flip, so coin stays tails. DONE

Figure 10: **Natural Language parity problem instances:** The coin flip task - which consists of tracking the state of a coin as it undergoes a number of flip or no-flip operations - shares the same underlying problem structure as the parity task.

of operations and number of variables (along with their names, which are single-letter characters) are uniformly sampled based on pre-set hyperparameters. (3) One by one, operations and the variables included in the operations are sampled, while making sure that each added operations retains the semantic correctness of the program.

We now outline the hyperparameters used to generate the *chain-like* and *diverse* splits.

Chain-like split:

- **Boolean operators:** assign to *and* with another variable, assign to *or* with another variable, assign to *xor* with another variable, negate
- **Minimum/maximum number of operations:** 8, 30
- **Minimum/maximum number of variables in the program:** 4, 8

Diverse split:

Input:

```
e = False
t = True
v = True
y = True
o = False
n = False
if False == n:
        n = True
if False == v:
        e = True
if False == y:
        v = True
v = True and v
if False == t:
        e = True
if e == True:
        y = v
print(y)
```

Scratchpad targets:

```
# With comments:
e = False
# e = False
t = True
# t = True
v = True
# v = True
y = True
# y = True
o = False
# o = False
n = False
# n = False
if False == n:
        n = True
# n = True
if False == v:
        e = True
# e = False
if False == y:
        v = True
# v = True
v = True and v
# v = True
if False == t:
        e = True
# e = False
if e == True:
        y = v
# y = True
print(y)
# Answer: True
```

Input:

```
m = True
z = True
j = False
n = True
v = False
z = m or z
m = False ^ m
v = False ^ v
j = m or z
n = j or z
v = False ^ v
m = True ^ m
print(v)
```

Scratchpad targets:

```
# With comments:
m = True
# m = True
z = True
# z = True
j = False
# j = False
n = True
# n = True
v = False
# v = False
z = m or z
# z = True
m = False ^ m
# m = True
v = False ^ v
# v = False
j = m or z
# j = True
n = j or z
# n = True
v = False ^ v
# v = False
m = True ^ m
# m = False
print(v)
# Answer: False
```

Figure 11: **Variable assignment problem instance:** A problem instance from the Boolean Variable Assignment dataset. The scratchpad strategy consists of outputting the value of the recenly updated variable in form of comments. Note that both the input and the scratchpad are valid Python programs.

- **Boolean operators:** assign to *and* with another variable, assign to *or* with another variable, assign to *xor* with another variable, negate, assign to *and* with boolean, assign to *or* with boolean, assign to *xor* with a boolean, conditional assign to a boolean, assign to another variable, conditional assign to another variable
- **Minimum/maximum number of operations:** 16, 32
- **Minimum/maximum number of variables in the program:** 4, 8

The scratchpad format (seen in Figure 11) consists of copying over the input program with comments after each line specifying the value of the recently updated variable. This ensures that the scratchpad itself is a valid Python program and can be used with models pretrained with Python data.

Both splits have 1500000 samples.

B Baseline for Vanilla Finetuning on Variable Assignment

Just how weak are the vanilla finetuned models on the OOD lengths on the variable assignment task? We trained baseline models with the same parameter count on a modified version of the variable assignment dataset where the order of the operations were randomly shuffled. While this leaves in some of the spurious features that correlate with the right answer, it completely eliminates the possibility of running a sequential algorithm to get to the final answer.

The results can be found in Figure 12. On OOD data, the performance of the shuffled-ops baseline is on par with the models trained with the clean version of the dataset. Note that the length generalization deficiency that the shuffled ops baselines display is even more dramatic than that of the models'

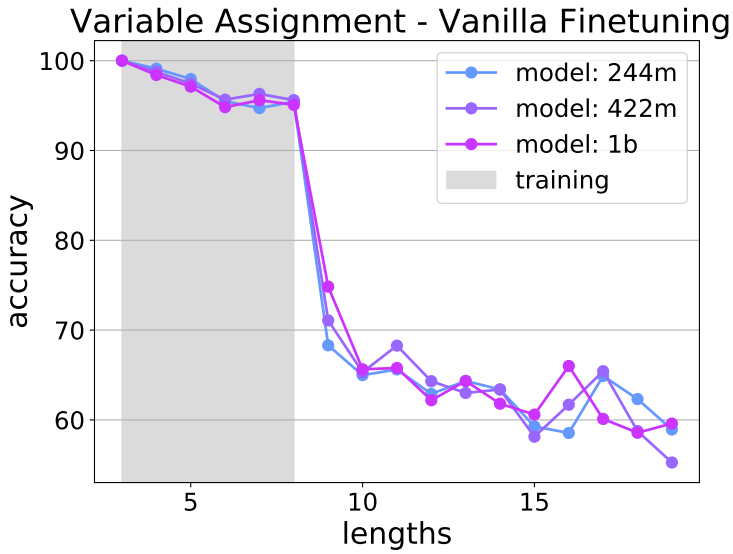

Figure 12: **Results of finetuning on the shuffled-ops variable assignment dataset:** We trained baseline models with the same parameter counts on a modified version of the variable assignment dataset where the order of the operations were randomly shuffled. The generalization gap between in and out-of-distribution data persists here as well, due to transformers' tendency to prefer parallel strategies that don't generalize to larger lengths.

trained with clean data. This is not surprising, as the primary source of lack of length generalization is the transformers' tendency to prefer parallel strategies that don't generalize to larger lengths over picking up sequential algorithm

C Experimental Conditions

We outline the training conditions and hyperparameter used in the main experiments.

In our experiments on different datasets, we first did a learning rate sweep over different model sizes, and preferred the largest learning rates that ensured training stability. This is due to the fact that we observed the best length generalization when we used larger learning rates (see Figure 5). We used the AdaFactor optimizer in all of our finetuning experiments [11]. We didn't use dropout [33] in the parity experiments and used a dropout rate of 0.05 in the variable assignment experiments. Our initial experiments suggest that one can often reach similar in and out-of-distribution performance either by training the weights from scratch, or finetuning from the pretrained weights. To keep the experiments consistent, we always initialized training using the pretrained weights. In variable assignment, we did a learning rate sweep over 0.0033, 0.00033 and 0.000033. For parity, we search over learning rate velus of 0.002, 0.0002 and 0.00002. We used a constant learning rate profile all throughout learning. Due to memory constraints, we used a batch size of 32 when training the $64b$ and $128b$ models.

We used greedy decoding in all of our experiments (including few-shot scratchpad ones). We experimented with temperature sampling with reranking based on sentence likelihoods, but found that doing this doesn't lead to qualitatively different results.

D Computational Graph Depth is the Relevant Notion of Difficulty on Variable Assignment

Computational graph depth captures a more relevant notion of difficulty on the variable assignment task for transformers. In Figure 13, we show how the accuracy of a transformer model evolves on samples of problem instances with different computational graph depths (left), and how the same

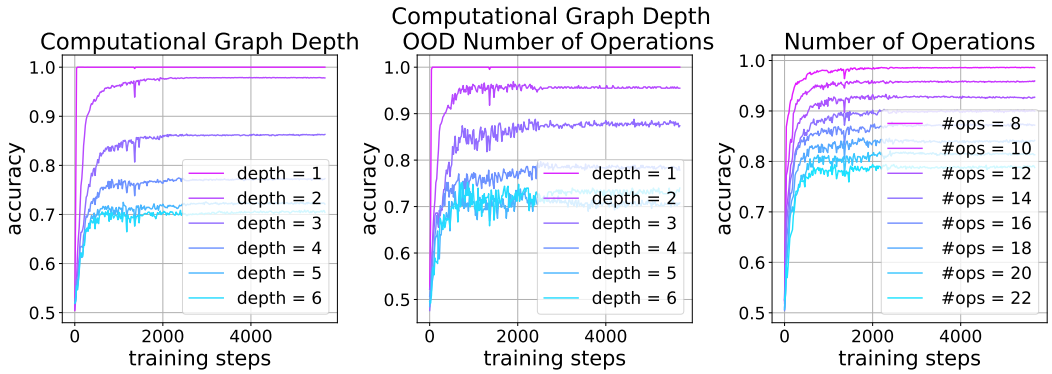

Figure 13: **Evolution of Performance over Training Iterations on Different Computational Graph Depths:** Computational graph depth corresponds to the length of the longest dependency chain linking to the queried variable in the variable assignment task. This quantity captures a more suitable notion of length/difficulty for transformer models. On the **left** plot, we show how the accuracy of a transformer model evolves for problem instances with different computational graph depths. In the *middle*, we show the same, except that we constrain the number of operations in the program to an out-of-distribution number. The accuracy values roughly remain unchanged, indicating that it's not the number of operations, but computational graph depth that determines the difficulty of a problems instance. On the right, we show the evolution of accuracy on instances with different number of operations for reference.

quantity behaves if we fix the number of operations in a program at an out-of-distribution length (middle) [4]. Two takeaways from this analysis are:

- The fact that the accuracy values on samples with different computational graph depths roughly remain unchanged on OOD program lengths indicates that it's not the number of operations, but computational graph depth that captures a more relevant notion of difficulty.
- Transformers initially pick up how to handle programs with a small computational graph depth throughout training and then move to more difficult programs.

E Effect of Prompt Style on Few-Shot Finetuning Performance

In Section 5.1, we hypothesized that few-shot finetuning only leads to significant improvements in length generalization performance if the non-finetuned performance on the same task already at a nontrivial level. To provide a sanity check for this, we ran few-shot finetuning using an alternative prompt style for the coin-flip task that yields poor non-finetuned performance. As can be seen in image , the few-shot finetuned performance shows significant length generalization pathologies. We leave a systematic study of how prompt style affects length generalization as future work.

F Distractor Analysis for Scratchpad Strategies

Our analysis in Section 4 indicates that length generalization pathologies persist even when we use the padded scratchpad strategy that makes sure that it's not untrained position encodings and/or the EOS token prediction that causes the aforementioned pathologies. This points to the fact that the transformer doesn't learn to attend to the "right" section of the input and scratchpad that implements the sequential strategy that generalizes to longer lengths — it's thrown off by distractor tokens in the input and/or the preceding scratchpad targets. The distractor tokens at which section of the transformer context window (input or scratchpad) contribute more to the performance deterioration? If we remove all the distractor tokens, can we achieve perfect length generalization?

To answer these questions, we trained four transformer models under the following conditions: (1) We used the padded scratchpad strategy described in Section 4 with no modification, (2) We manually masked the preceding scratchpad tokens that don't contribute to the correct sequential algorithm (i.e.

[4]The longest in-distribution number of operations is 15, and we fix this quantity at 20 in the middle plot.

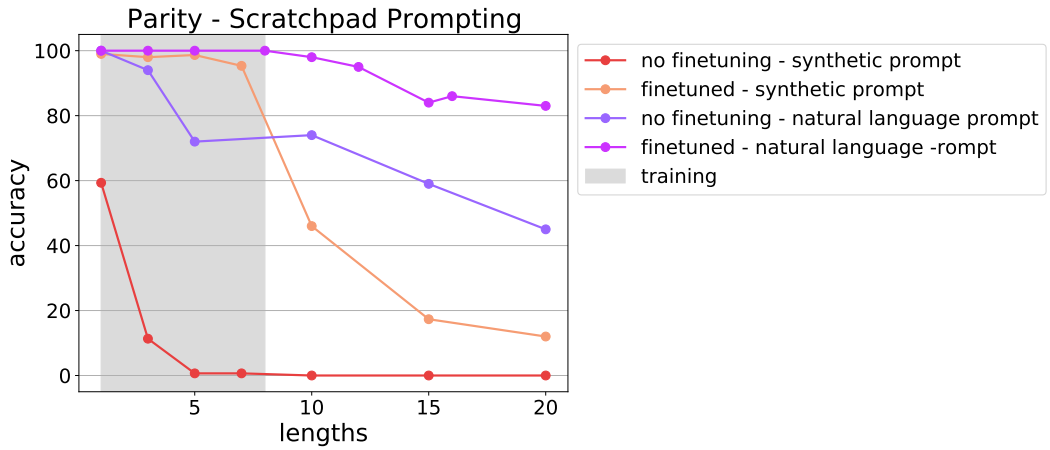

Figure 14: **Effect of prompt style on few-shot finetuning performance:** we evaluated the (few-shot) finetuned performance of the pretrained model on an alternative, synthetic prompt style that yields poor performance without any pretraining. We observed that the few-shot finetuned model on this task also shows significant length generalization pathologies. This is in line with our hypothesis that the non-finetuned performance of the base model should be non-trivial for few-shot finetuning to consistently yield strong length generalization results.

we masked the distractor tokens in the input), (3) We masked the input tokens that don't contribute to the correct sequential algorithm, (4) we masked the distractor tokens in both the input and the preceding distractor tokens. To give an example, let's say the input bitstring is "[1 1 0 1 1]", and the model has emitted the scratchpad tokens "[1 0 0]" so far. Masking the distracting sratchpad tokens simply means replacing all but the last scratchpad token with dummy padding tokens: "[x x 0]". Similarly, removing the distracting input tokens corresponds to masking out the part of the input that the network doesn't need to attend to while predicting the next bit: "[x x x 1 x]". Masking both corresponds to removing the distractor tokens in both the input and the target, so that the input and the preceding scratchpad become "[x x x 1 x]" and "[x x 0]" respectively.

The results can be seen in Figure 15. For all four experimental conditions, we plotted the accuracy of the trained models on predicting the scratchpad tokens at different steps for inputs of varying (in and OOD) lengths. We conclude from this experiment that:

- Removing all distractor tokens does result in perfect length generalization.
- The distracting input tokens are hurt length generalization performance more.

Based on this analysis, we conclude that innovations in transformer architectures and/or training methodology/objective that alleviate the issues caused by distractor tokens have a chance at significantly improving length generalization.

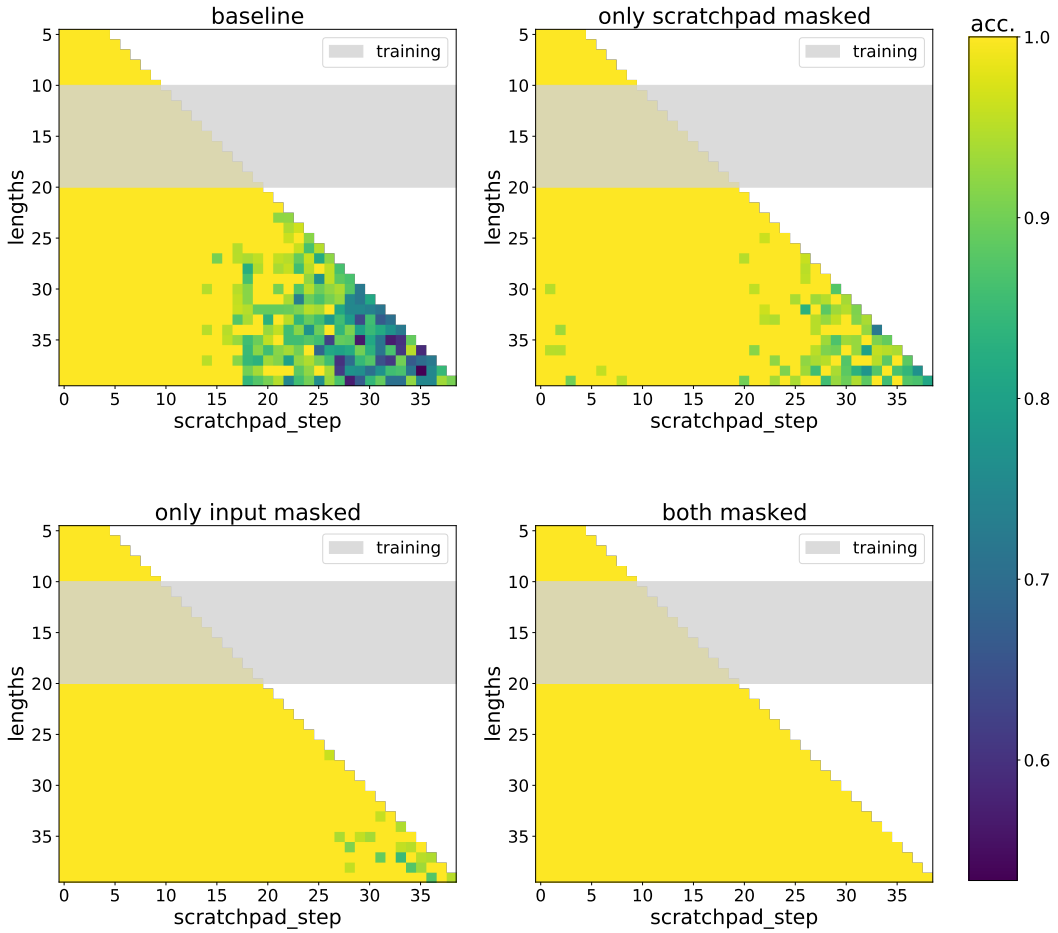

Figure 15: **Distractor analysis:** We trained scratchpad-augmented transformers to solve the parity task, where we systematically masked out the tokens in the input (left bottom) and the scratchpad (right top) that don't need to be attended to while implementing the correct sequential algorithm that solves parity. The plots illustrate the accuracy of the trained models on predicting the scratchpad tokens at different steps for inputs of varying (in and OOD) lengths. This analysis shows that (1) removing all distracting tokens leads to perfect length generalization (right bottom), (2) the distractor tokens in the input contribute more to the length generalization pathologies (right top versus left bottom).