SELF-IMPROVING TRANSFORMERS OVERCOME EASY-TO-HARD & LENGTH GENERALIZATION CHALLENGES

Anonymous authors
Paper under double-blind review

ABSTRACT

Large language models often struggle with length generalization and solving complex problem instances beyond their training distribution. We present a self-improvement approach where models iteratively generate and learn from their own solutions, progressively tackling harder problems while maintaining a standard transformer architecture. Across diverse tasks including arithmetic, string manipulation, and maze solving, self-improving enables models to solve problems far beyond their initial training distribution—for instance, generalizing from 10-digit to 100-digit addition without apparent saturation. We observe that in some cases filtering for correct self-generated examples leads to exponential improvements in out-of-distribution performance across training rounds. Additionally, starting from pretrained models significantly accelerates this self-improvement process for several tasks. Our results demonstrate how controlled weak-to-strong curricula can systematically teach a model logical extrapolation without any changes to the positional embeddings, or the model architecture.

1 INTRODUCTION

Despite the remarkable success of transformer-based language models (Vaswani et al., 2017) across a wide range of tasks, these models exhibit significant limitations in *length generalization*—the ability to extrapolate to longer sequences than those seen during training. Even in simple algorithmic tasks such as arithmetic, standard transformer models trained with autoregressive objectives struggle to generalize to longer problem instances (Dubois et al., 2019; Hupkes et al., 2020; Newman et al., 2020; Anil et al., 2022).

To address this, prior work has explored various approaches, including changes to positional embeddings (Ruoss et al., 2023; Li et al., 2023; McLeish et al., 2024; Kazemnejad et al., 2024; Sabbaghi et al., 2024; Cho et al., 2024; Zhou et al., 2024), architectural modifications (Fan et al., 2024; Duan et al., 2023), and data format changes such as index hinting (Zhou et al., 2023; 2024). While effective in controlled setups, these approaches are often incompatible with how large language models (LLMs) are trained in practice, as they introduce task-specific modifications that are unclear how and to what extent they would transfer to the general purpose settings.

In this work, we attempt to overcome length generalization challenges in the standard transformer setting, by building around an interesting phenomenon that transformers exhibit, i.e., "transcendence" (Zhang et al., 2024). Transcendence is the ability of a student model to generalize slightly beyond the difficulty of the data provided by a teacher during training. Specifically, models trained on simple instances of a task, say n digit arithmetic, can sometimes generate correct outputs for slightly harder instances, e.g., $n + 1$ digit arithmetic, with some accuracy. We leverage this property by applying a **self-improvement** framework, where the model iteratively generates its own training data and progressively learns from harder examples.

Self-improvement has been widely studied in various contexts (Singh et al., 2023; Gulcehre et al., 2023; Liang et al., 2024), typically in settings where external verifiers, weak supervision, or filtering mechanisms are used to ensure data quality. We demonstrate that extreme length generalization is indeed possible under this framework, *without any modification to the base transformer architecture*. For tasks like reverse addition and string copying, self-improvement succeeds with no explicit data filtering. However, for harder problems such as multiplication and shortest-path finding in mazes, self-improvement without data filtering fails due to error accumulation. We show that simple filtering

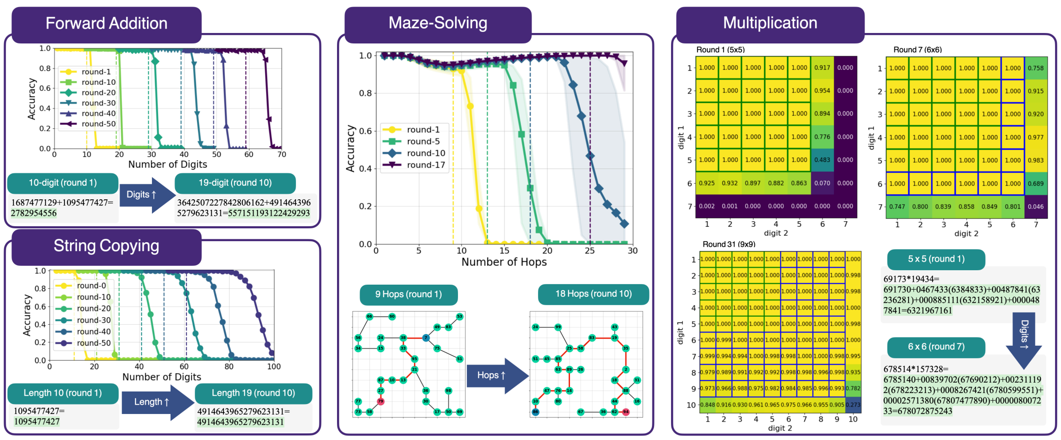


Figure 1: Overview of self-improvement results. Models trained with self-improvement can tackle increasingly complex tasks that extend far beyond their initial training distributions, achieving significant generalization **without any additional supervision**.

techniques—such as length filtering and majority voting—suffice to maintain data quality and enable self-improvement to extend far beyond the initial training distribution.

Our findings suggest that self-improvement is not limited to length generalization but also enables *easy-to-hard generalization*, where a model trained on simpler tasks successfully learns harder tasks without additional supervision. Notably, our approach does not introduce a new self-improvement framework but instead demonstrates its effectiveness across diverse algorithmic tasks.

Furthermore, we investigate the dynamics of self-improvement and show that: (1) controlling the weak-to-strong curriculum is crucial, as models require a structured difficulty schedule to avoid catastrophic failure, (2) self-improvement accelerates over time, as models increasingly benefit from harder examples, leading in some cases to exponential extrapolation, and (3) starting with a pretrained models singificantly accelerates self-improvement, allowing to generalize further and faster than models trained from scratch.

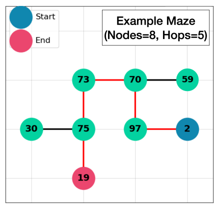
Our findings provide evidence that learn self-improvement is a general purpose and scalable solution for length and easy-to-hard generalization. Our contributions can be summarized as:

1. We apply an iterative self-training framework to train transformers on the arithmetic, maze and string manipulation tasks, and successfully tackle **easy-to-hard generalization** to extreme out-of-distribution test data.
2. We motivate the importance of a carefully crafted self-improvement schedule and **label filtering** based on length and majority voting, which are central to consistent self-improvement.
3. We show that the rate of self-improvement can be exponential and pretrained models can achieve faster acceleration in easy-to-hard generalization.
4. We investigate some key failure modes of self-correction due to label noise leading to an **error avalanche**, and discuss how they can be overcome through weak verification.

2 Related Works

Length and Easy-to-Hard Generalization. Length generalization is concerned with extrapolating to longer sequence lengths than those seen during training (Anil et al., 2022). Previous approaches to improve length generalization includes architectural modifications, including specialized positional embeddings (Li et al., 2023; Ruoss et al., 2023; McLeish et al., 2024; Kazemnejad et al., 2024; Sabbaghi et al., 2024; Cho et al., 2024; Zhou et al., 2024), looping Fan et al. (2024), novel attention mechanisms (Duan et al., 2023), and input format augmentation (Zhou et al., 2023; 2024). In contrast, our approach adheres to the standard transformer architecture without introducing significant modifications to architecture, positional encoding, or input structure. While prior approaches typically rely on fixed-length training dataset, we alternate between training and generating training datasets.

Table 1: Examples of Tasks Considered

Task Type	**Input** (**Q**: Prompt, **A**: label)	**Task Difficulty**
Reverse Addition	Q: 31558+91786= A: 232451	Max digit length of the two operands
Forward Addition	Q: 85513+68719= A: 154232	
Multiplication	Q: 34895*148= A: 348950+0273932(3653542) +00447874=36972305	
Copy	Q: 12345= A:12345	Length of string
Reverse	Q: 12345= A: 54321	
Maze Solving	 Finding shortest path from node 2 to 19 (← example image for illustration) Q: 2>19#73:70,75-97:2,70-70:73,97,59 -75:73,30,19-2:97-30:75-59:70-19:75= A: 2>97>70>73>75>19	(1) Number of hops between start & end (2) Number of nodes

More generally, easy-to-hard generalization is the paradigm where human annotation is provided for easier tasks but aiming to enable generalization to harder tasks with no additional supervision (Schwarzschild et al., 2021; Bansal et al., 2022; Burns et al., 2023; Hase et al., 2024; Sun et al., 2024). For instance, Zhang et al. (2024) study this *transcendence* phenomenon in chess, showing that chess transformers can sometimes outperform all players in the training dataset. Similarly, Sun et al. (2024) finds that a reward model trained on easier math problems can be effectively transferred to harder problems, through reinforcement learning.

Self Improvement. When high-quality training labels are unavailable or costly to obtain, training on self-generated labels provides an efficient way to enhance model capabilities. Typically, this involves generating candidate labels, filtering or verifying them to prune errors, and retraining on the refined self-generated data (Zelikman et al., 2022; Wang et al., 2022b; Huang et al., 2022; Singh et al., 2023; Chen et al., 2023; Gulcehre et al., 2023; Madaan et al., 2024; Yuan et al., 2024; Liang et al., 2024). This approach has been successfully applied across various domains, including reasoning (Zelikman et al., 2022; Huang et al., 2022; Singh et al., 2023), mathematics (Zhang & Parkes, 2023; Charton et al., 2024; Liang et al., 2024), coding (Chen et al., 2023), and general instruction tuning (Wang et al., 2022b; Yuan et al., 2024).

Extensive discussion of related works is in Appendix A.

3 Preliminaries and Experimental Setup

In this section, we describe the experimental setup, including the model architecture, tasks, training methodology, evaluation criteria, and the self-improvement framework.

Models We adopt the LLaMA architecture with six layers, six attention heads, and an embedding dimension of 384 and a total of 14M parameters. Positional embeddings are excluded, using the No Positional Encoding (NoPE) method (Kazemnejad et al., 2024). Character-level tokenization is used across all tasks, except for the maze-solving task, where numbers (0–99) are tokenized individually.

Tasks We evaluate our approach on a diverse set of tasks, categorized into arithmetic operations, string manipulation, and maze solving. All tasks we consider admit a straightforward notion of difficulty. We denote the difficulty level of a problem instance x as an integer Difficulty(x). Table 1 provides examples, difficulty definitions, and relevant sections of each task.

- **Arithmetic operations:**
  1. *Addition* : We consider both reverse and forward addition of two numbers of equal length. In reverse addition, both operands and the answers are reversed, so they are written with the least significant digit first. Forward addition, in contrast, follows the standard format.
  2. *Multiplication* : Multiplication tasks are presented in a chain-of-thought (CoT) data format (Deng et al., 2024), which includes intermediate steps to guide the computation.

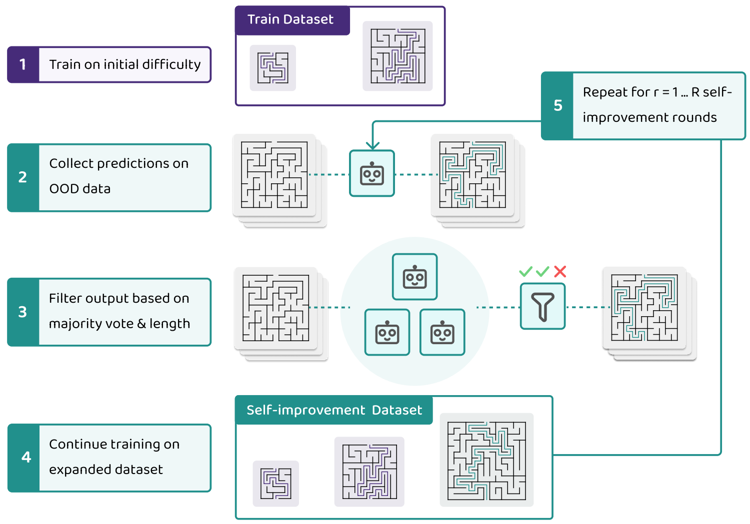


Figure 2: Illustration of our self-improvement procedure. At each round, the training data is updated with the model's predictions on progressively harder problems.

- **String manipulation:**
  1. *Copy* : Copying the input sequence.
  2. *Reverse* : Reversing the input sequence
- **Maze solving:** The task is to solve mazes represented as tree graphs. Given a tree graph and a specified start node and end node, the goal is to find the shortest path.

Data Generation and Sampling We generate an initial supervised training dataset $\mathcal{D}_0$ of up to a fixed difficulty level d_0 by uniformly sampling the difficulty level $d \leq d_0$, followed by independent sampling of the data conditioned on the difficulty. Denoting the input as x_i, labels as y_i,

$$\mathcal{D}_0 = \{(x_i, y_i)\}_{i=1}^{N_0}, \quad \text{where Difficulty}(x_i) \leq d_0.$$

Details on data generation and sampling are provided in Appendix C.2.

Self-Improvement Framework The self-improvement framework begins by training a model using the labeled training dataset $\mathcal{D}_0$, which gives us our base model M_0.

For each subsequent round r ($r = 1, 2, 3, \dots$), we increase the problem difficulty, such as the number of digits or string length for arithmetic and string manipulation tasks, or the number of hops for maze-solving tasks, to d_r. Using the previous model M_{r-1}, we generate N_r new self-improve data samples $\mathcal{D}_r$ defined as:

$$\mathcal{D}_r = \{(x_i, M_{r-1}(x_i))\}_{i=1}^{N_r}, \quad d_{r-1} \leq \text{Difficulty}(x_i) \leq d_r$$

Instead of the true labels y_i, we obtain the predicted labels $M_{r-1}(x_i)$ from the output of the model. At each self-improvement round r, the model is trained on the combined dataset $\mathcal{D}_0 \cup \mathcal{D}_1 \cup \cdots \cup \mathcal{D}_{r-1}$, which includes the initial labeled dataset and all subsequent self-improvement datasets. To ensure sufficient training on the most recently generated data $\mathcal{D}_{r-1}$, we up-sample it with a sampling probability of 50%. The remaining datasets $\mathcal{D}_0, \dots, \mathcal{D}_{r-2}$ are sampled uniformly at random. This iterative process allows the model to gradually tackle harder problems, leveraging its own predictions to expand the training data and improve generalization.

Data Filtering We employ two unsupervised data-filtering methods to refine our self-improvement dataset: 1) length filtering and 2) majority voting. For a given self-improved dataset $\mathcal{D}_r = \{(x_i, M_{r-1}(x_i))\}_{i=1}^{N_r}$ at round r, data is filtered based on specific criteria on the model-generated outputs $M_{r-1}(x_i)$, producing a smaller, refined dataset $\tilde{\mathcal{D}}_r = \{(x_i, M_{r-1}(x_i))\}_{i=1}^{\tilde{N}_r}$.We provide more details on the motivation and implementation in Section 5.

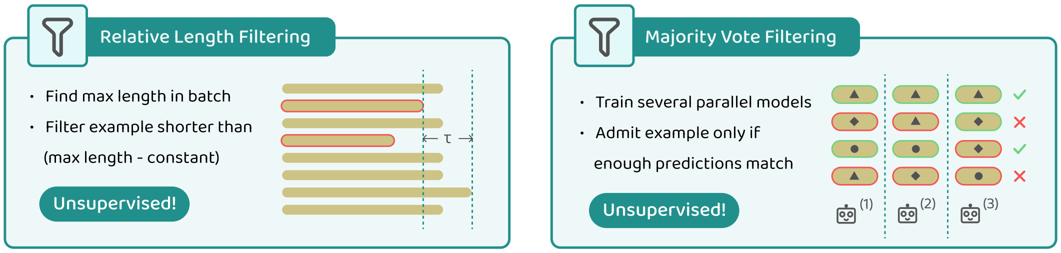


Figure 4: Overview of the two data-filtering methods employed.

Training and Evaluation Except for the experiments on pretrained Llama 3.2 models, all models are trained from scratch using the conventional next-token prediction objective. The loss is computed solely on the completion, meaning that the input prompt is masked, and only the model's predictions are included in the loss computation. Detailed settings, including hyperparameters and training schedules, are provided in the Appendix C.3.

During inference, we use greedy decoding and exact-match accuracy as the primary metric for evaluation. A prediction is deemed correct if all tokens in the output sequence match the ground truth; any discrepancy in the generated tokens is classified as an incorrect prediction.

4 Warm-up: Length Generalization on Reverse Addition

Reversed addition, where the operands and output are written with the least significant digit first, has been shown to enhance sample efficiency and performance (Lee et al., 2023). Reversed addition has become a popular setting for studying length generalization in arithmetic tasks (Lee et al., 2023; Shen et al., 2023; Zhou et al., 2023; 2024; Cho et al., 2024; McLeish et al., 2024).

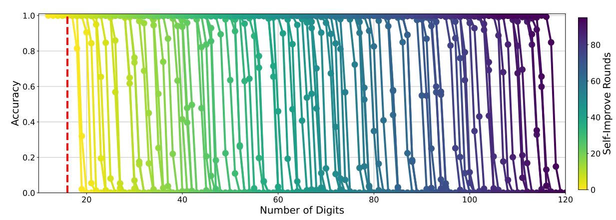


Figure 3: Reverse addition task. The self-improvement framework enables a model initially trained on 1-16 digit examples to generalize perfectly to over 100-digit addition. Each shade of color is a different self-improvement round.

Results Figure 3 demonstrates that, starting with a model trained on 1 to 16-digit reverse addition, the self-improvement framework enables near-perfect length generalization up to 100 digits without any additional supervision or modifications to positional encodings, input formats, or the Transformer architecture.

5 Unsupervised Data Filtering

Our framework leverages models' ability to generalize slightly beyond their training difficulty to sample increasingly hard examples. A critical component for success is the quality of the self-generated data. Low-quality data can negatively impact the model's generalization performance, leading to even lower-quality data in subsequent rounds and ultimately causing a cascading degradation of the self-improvement process as illustrated in Figure 22.

While cascading error effects are analyzed in greater detail in Appendix B.6 and Section 5, this section focuses on two key data-filtering methods used in this work: length filtering and majority voting (Figure 4). And in Section 6 we apply the filtering methods to enable difficulty generalization in forward addition, multiplication and mazes.

Relative Length Filtering. A common error in model-generated data is that the generated labels are often shorter than the correct answers (Figure 13). These observations motivate a filtering method based on the relative lengths of model-generated predictions. Specifically, predictions shorter than a predefined threshold—calculated relative to the maximum prediction length within their batch—are filtered out. For a batch of model-predicted outputs, we identify the maximum length of the output $L = \max |M_{r-1}(x_i)|$ and filter out predictions $M_{r-1}(x_i)$ with lengths shorter than a predefined threshold τ. This method is *unsupervised*, as it relies solely on comparing lengths within model-generated outputs rather than referencing ground-truth labels. While particularly suited to length generalization tasks, where harder problems are expected to yield longer answers, length-based filtering shows broader potential for addressing similar challenges in other tasks.

Majority Voting Generating multiple candidate answers to ensure self-consistency is a widely used approach for enhancing data quality (Huang et al., 2022; Wang et al., 2022a; Qu et al., 2024; Peng et al., 2024). However, unlike the common practice of sampling multiple reasoning paths by generating outputs with a non-zero temperature, our task of interest requires a single correct answer for each instance. To address this, we train k models $(M_{r-1}^{(1)}, \cdots, M_{r-1}^{(k)})$ using different random seeds and self-improvement data, then apply a majority-voting mechanism with a threshold τ.

For each self-improved dataset $\mathcal{D}_r^s = \{(x_i, M_{r-1}^{(s)}(x_i))\}_{i=1}^{N_r}$ where s is the seed index, we filter the data such that only pairs $\{(x_i, M_{r-1}^{(s)}(x_i))\}$ where $M_{r-1}^{(s)}(x_i)$ matches at least $\lceil \tau \times k \rceil$ outputs among the k models are retained. This ensures that only high-consensus data are preserved for training in subsequent rounds, thereby significantly improving overall data quality and model performance. This approach is conceptually similar to an iterative version of the bagging algorithm (Breiman, 1996).

6 Length and Difficulty Generalization on Forward Addition, Multiplication, Maze

We extend our evaluation to a class of harder tasks, including forward addition, multiplication, and maze-solving. Our results demonstrate that the framework is not limited to length generalization but extends to **difficulty generalization**, where the model incrementally learns to solve increasingly difficult problems. By employing controlled sampling of problem difficulty and data filtering techniques for each round, the model successfully adapts to harder tasks, highlighting the versatility and robustness of the self-improvement approach.

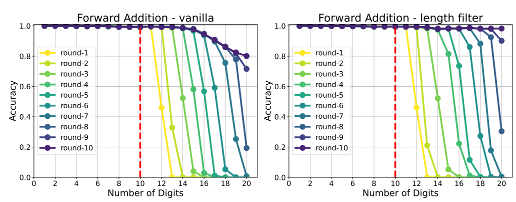


Figure 5: Models trained on forward addition over 10 self-improvement rounds. (Left) Without data filtering. (Right) With length-based filtering using a threshold of 2. Data filtering significantly enhances length generalization performance.

6.1 Forward Addition

Forward addition is a straightforward task, yet very challenging for transformer models to length generalize on. In reverse addition, each step only requires processing a fixed-size subset of the input. However, in the forward addition, the size of the relevant input required to generate correct tokens increases, making the problem more complex (Zhou et al., 2023).

Results. Figure 5 shows the results of forward addition experiments, where the model is initially trained on labeled data of up to 10 digits and then undergoes 10 rounds of self-improvement.

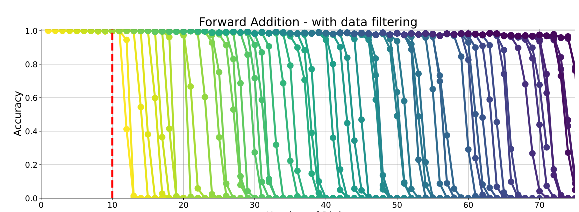


Figure 6: Results on the forward addition task with length filtering. The model is initially trained on labeled forward addition data of lengths 1 to 10. Using the self-improvement framework over 60 rounds, with incremental increases in digit length by 1 per round, the model achieves strong generalization to lengths up to 75.

Without any data filtering (Left), the model's performance begins to deteriorate after a few rounds of training, leading to a collapse in generalization. However, applying the length-based filtering approach with a threshold length of 2 results in significant improvements in length generalization performance (Right). By refining the dataset at each round, the self-improvement framework remains robust across multiple rounds.

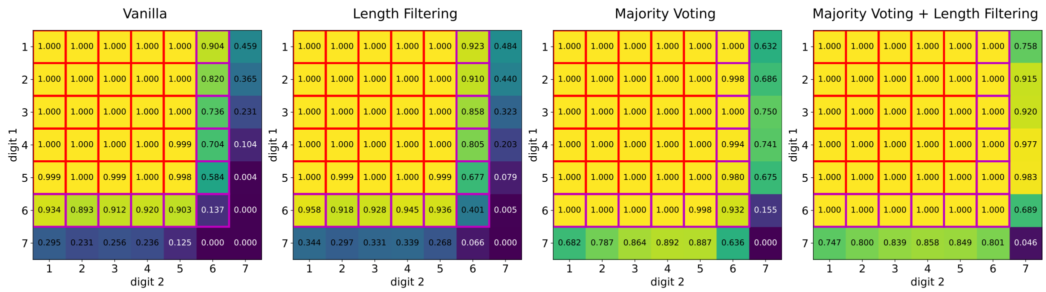


Figure 7: Comparison of filtering methods at round 7. From left to right: no filtering, length filtering, majority voting, and a combination of majority voting and length filtering. Data filtering significantly improves self-improvement performance, with the combined approach achieving the best results.

With continued training over 60 self-improvement rounds, the model maintains performance exceeding 98% accuracy for sequences up to length 70 (Figure 6). This demonstrates the effectiveness of length-based filtering in sustaining the self-improvement process and enabling length generalization.

6.2 Multiplication

We also extend our approach on multiplication, which is a challenging task even in-distribution (Dziri et al., 2024). Fine-tuning large language models on datasets with chain-of-thought(CoT) steps has shown limited success. We adopt a data format similar to Deng et al. (2024), where multiplication is given a problem of multiplying two numbers, the label expands the multiplication into steps that include partial products of multiplying the first operand with each digit of the second operand and the intermediate results.

The model is initially trained on n-by-n multiplication examples with $n = 5$. Directly introducing $n+1$-by-$n+1$ examples results in poor performance, hence, we adopt a more fine-grained difficulty schedule where we sample $n+1$-by-m and m-by-$n+1$ examples with m growing from 1 to $n+1$. This gradual progression allows the model to adapt incrementally to larger operand sizes, making the transition to harder examples more manageable.

Results. To improve the quality of self-generated training data, we apply three data filtering methods: length filtering, majority voting, and a combination of both (Appendix C.3).

Figure 7 compares the effectiveness of these filtering methods at round 7, where models are trained on self-generated data for up to 6-by-6 multiplication. All three filtering methods enhance self-improvement, with majority voting outperforming length filtering. Applying both majority voting and length filtering achieves near-perfect generalization to 6-by-6 multiplication.

Training for additional rounds further extends this generalization. The combined filtering strategy continues to yield near-perfect accuracy up to 9-by-9 multiplication (Figure 36), with the potential for even further generalization in subsequent rounds. We further demonstrate that we can accelerate the process, achieving perfect performance on 10-by-10 multiplication in just 19 rounds (Figure 20).

6.3 Maze

We extend our evaluation from arithmetic to a more complex problem: finding the shortest path in a maze. Pathfinding presents significant challenges for autoregressive models (Bachmann & Nagarajan, 2024). Our mazes can be represented by a tree graph in a 2-dimensional space and they do not have loops. Figure 31 provides a visualization of this task and the corresponding input and output data format. Details on maze generation are provided in Appendix C.2.3.

We evaluate two generalization settings: 1) increasing the number of hops while keeping the number of nodes fixed, and 2) increasing the number of nodes while keeping the number of hops fixed. In the first setting, the input graph description remains constant in size, but the output length grows as the difficulty increases. In the second setting, the input graph expands with more nodes, while the output remains of fixed length.

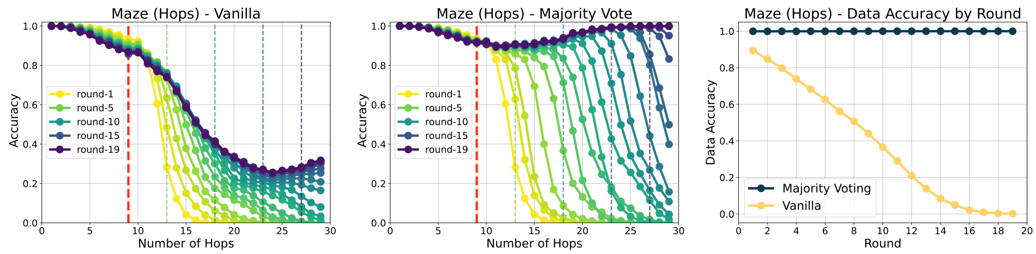


Figure 8: Maze-solving with increasing hops ($N = 30$ nodes). Models are trained on graphs with up to 9 hops and generalized by increasing hops by 1 in each self-improvement round. Results show mean accuracy across 3 seeds. (Left) No filtering. (Middle) Majority voting. (Right) Self-improve data accuracy per round. Filtering significantly enhances data accuracy and improves generalization.

6.3.1 Increasing the Number of Hops

The difficulty of the maze-solving task increases with the number of hops required from the start node to the end node. We begin by training the model on a labeled dataset containing paths of up to $h = 9$ hops. In each self-improvement round, we increase h by one, progressively introducing longer paths, while fixing the number of nodes $N = 30$.

Results. As shown in Figure 8, without data refinement, self-generated training data degrades over successive rounds, leading to an eventual collapse in the self-improvement process. In contrast, majority voting stabilizes data quality, allowing near-perfect data quality and the model continues to successfully generalize to paths up to 30 hops.

Additional results on increasing the number of nodes (while fixing the number of hops), and using external verifiers on the validity of moves or the end nodes are provided in the Appendix B.3. These results show that majority voting based filtering—without any external verification—performs comparably with using oracle verifiers and allows difficulty generalization.

7 Ablations

7.1 Increasing OOD generalization with more self-improvement

Sampling instances that are too difficult for the current model is detrimental to the quality of self-improvement data, which causes downstream performance to break down. However, in tasks like reverse addition and copy, we observe that the out-of-distribution (OOD) extrapolation capabilities improve progressively as the model undergoes more rounds of self-improvement, which means we can sample more and more difficulty levels every round. Figure 9 illustrates how the number of additional OOD lengths achieving over 99% accuracy grow with each round when the model is self-improved using only one additional digit per round. The model's OOD extrapolation capabilities expand as it is trained on longer sequences.

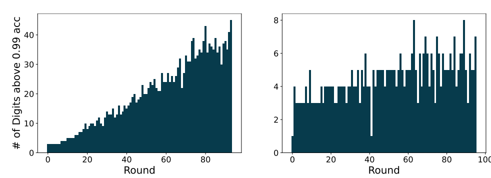


Figure 9: Number of **extra** OOD digit lengths achieving over 99% accuracy when self-improving with one additional digit per round, on (Left) copy and (Right) reverse addition tasks. The growing OOD capability suggests the potential to sample more digits per round as self-improvement progresses.

7.2 Accelerating self-improvement

Since the amount of extra OOD generalization increases roughly linearly with each additional round of self-improvement (Figure 9), sampling as many difficulty levels as possible per round could lead to exponential improvements in performance. Therefore, we propose an accelerated self-improvement schedule: At each round, the self-improvement dataset is uniformly sampled from *all difficulty levels achieving over 99% evaluation accuracy*, instead of incrementally sampling by only one additional length. As shown in Figure 10, this approach allows the model to achieve 100 digit extrapolation with less than half of the rounds. All other hyperparameters remain unchanged. We also provide results in the multiplication setting in Figure 20.

7.3 Pretrained Models

We extend our self-improvement framework to pretrained models, specifically Llama-1B and Llama-3B (AI@Meta, 2024), to explore scaling effects and the impact of finetuning on larger models. For consistency in tokenization, we use character-level tokenization instead of the default tokenizer of the Llama models, and use LoRA (Hu et al., 2021).

Results. Larger models achieve better extrapolation performance, which leads to faster acceleration with larger models. Figure 10 compares self-improvement acceleration between Llama-3B, Llama-1B, and a smaller 14M parameter model trained from scratch. The results demonstrate that larger pretrained models can generalize to longer sequences with fewer rounds of self-improvement.

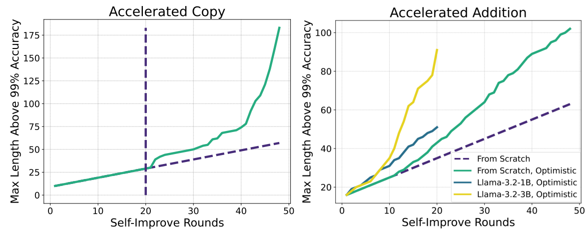


Figure 10: Maximum input length achieving over 99% accuracy at different self-improvement rounds for (Left) Reverse addition and (Right) Copy task. The dashed linear line represents the standard schedule of sampling one additional length per round. Faster self-improvement schedules allow the model to generalize to longer inputs with fewer rounds. Furthermore, finetuning from pretrained models enhances the acceleration.

8 Limitations

In our framework, the model does not generate new input instances during self-improvement; it only generates solutions (labels) for training. When it is unfeasible to generate the problems themselves, modeling the input distribution conditioned on task difficulty becomes an additional challenge.

A key consideration in self-improvement is defining and quantifying task difficulty. In real-world domains such as mathematics and natural language tasks, formalizing "difficulty" remains an open question. Our experiments demonstrate that careful difficulty scheduling is crucial for effective self-improvement. However, we also find that models exhibit some robustness to difficulty slack—especially when trained on harder tasks (Section 7.1) and when leveraging pretrained models (Section 7.3).

Another fundamental assumption in our framework is that models can handle slightly harder tasks than those seen in training. While this holds in many structured tasks, there are cases where such generalization is inherently difficult. For example, training on raw multiplication problems without intermediate steps leads to poor OOD generalization, making self-improvement infeasible. However, we show that breaking down tasks into intermediate steps enables slight OOD generalization, which can be leveraged for self-improvement(Section 6.2). This highlights the importance of designing task representations that align with a model's inherent generalization capabilities.

9 Conclusion

In this work, we have shown self-improvement training enables transformers to gradually generalize from easy to hard problems without access to hard labels. One extension is to incorporate more sophisticated verifiers as well as problem classes that is easy to verify but hard to solve. We expect self-improve to synergize with strong verification to enable transformers to solve harder problems beyond arithmetic or mazes.

References

AI@Meta. Llama 3 model card. 2024. URL https://github.com/meta-llama/llama3/blob/main/MODEL_CARD.md.

Sina Alemohammad, Josue Casco-Rodriguez, Lorenzo Luzi, Ahmed Imtiaz Humayun, Hossein Reza Babaei, Daniel LeJeune, Ali Siahkoohi, and Richard Baraniuk. Self-consuming generative models go mad. *ArXiv*, abs/2307.01850, 2023. URL https://api.semanticscholar.org/CorpusID:259341801.

Cem Anil, Yuhuai Wu, Anders Andreassen, Aitor Lewkowycz, Vedant Misra, Vinay Ramasesh, Ambrose Slone, Guy Gur-Ari, Ethan Dyer, and Behnam Neyshabur. Exploring length generalization in large language models. *Advances in Neural Information Processing Systems*, 35:38546–38556, 2022.

Gregor Bachmann and Vaishnavh Nagarajan. The pitfalls of next-token prediction. *arXiv preprint arXiv:2403.06963*, 2024.

Arpit Bansal, Avi Schwarzschild, Eitan Borgnia, Zeyad Emam, Furong Huang, Micah Goldblum, and Tom Goldstein. End-to-end algorithm synthesis with recurrent networks: Extrapolation without overthinking. *Advances in Neural Information Processing Systems*, 35:20232–20242, 2022.

Hritik Bansal, Arian Hosseini, Rishabh Agarwal, Vinh Q Tran, and Mehran Kazemi. Smaller, weaker, yet better: Training llm reasoners via compute-optimal sampling. *arXiv preprint arXiv:2408.16737*, 2024.

Reza Bayat, Mohammad Pezeshki, Elvis Dohmatob, David Lopez-Paz, and Pascal Vincent. The pitfalls of memorization: When memorization hurts generalization. 2024. URL https://api.semanticscholar.org/CorpusID:274610625.

Quentin Bertrand, Avishek Joey Bose, Alexandre Duplessis, Marco Jiralerspong, and Gauthier Gidel. On the stability of iterative retraining of generative models on their own data. *ArXiv*, abs/2310.00429, 2023. URL https://api.semanticscholar.org/CorpusID:263334017.

Leo Breiman. Bagging predictors. *Machine learning*, 24:123–140, 1996.

Martin Briesch, Dominik Sobania, and Franz Rothlauf. Large language models suffer from their own output: An analysis of the self-consuming training loop. *ArXiv*, abs/2311.16822, 2023. URL https://api.semanticscholar.org/CorpusID:265466007.

Collin Burns, Pavel Izmailov, Jan Hendrik Kirchner, Bowen Baker, Leo Gao, Leopold Aschenbrenner, Yining Chen, Adrien Ecoffet, Manas Joglekar, Jan Leike, et al. Weak-to-strong generalization: Eliciting strong capabilities with weak supervision. *arXiv preprint arXiv:2312.09390*, 2023.

François Charton, Jordan S Ellenberg, Adam Zsolt Wagner, and Geordie Williamson. Patternboost: Constructions in mathematics with a little help from ai. *arXiv preprint arXiv:2411.00566*, 2024.

Xinyun Chen, Maxwell Lin, Nathanael Schärli, and Denny Zhou. Teaching large language models to self-debug. *arXiv preprint arXiv:2304.05128*, 2023.

Hanseul Cho, Jaeyoung Cha, Pranjal Awasthi, Srinadh Bhojanapalli, Anupam Gupta, and Chulhee Yun. Position coupling: Improving length generalization of arithmetic transformers using task structure. 2024. URL https://api.semanticscholar.org/CorpusID:273695226.

Gonzalo Martínez Ruiz de Arcaute, Lauren Watson, Pedro Reviriego, José Alberto Hernández, Marc Juárez, and Rik Sarkar. Combining generative artificial intelligence (ai) and the internet: Heading towards evolution or degradation? *ArXiv*, abs/2303.01255, 2023. URL https://api.semanticscholar.org/CorpusID:257280389.

Yuntian Deng, Yejin Choi, and Stuart Shieber. From explicit cot to implicit cot: Learning to internalize cot step by step. *arXiv preprint arXiv:2405.14838*, 2024.

Elvis Dohmatob, Yunzhen Feng, Pu Yang, Francois Charton, and Julia Kempe. A tale of tails: Model collapse as a change of scaling laws. *ArXiv*, abs/2402.07043, 2024. URL https://api.semanticscholar.org/CorpusID:267628004.

Shaoxiong Duan, Yining Shi, and Wei Xu. From interpolation to extrapolation: Complete length generalization for arithmetic transformers. *arXiv preprint arXiv:2310.11984*, 2023.

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

Nouha Dziri, Ximing Lu, Melanie Sclar, Xiang Lorraine Li, Liwei Jiang, Bill Yuchen Lin, Sean Welleck, Peter West, Chandra Bhagavatula, Ronan Le Bras, et al. Faith and fate: Limits of transformers on compositionality. *Advances in Neural Information Processing Systems*, 36, 2024.

Ying Fan, Yilun Du, Kannan Ramchandran, and Kangwook Lee. Looped transformers for length generalization. *arXiv preprint arXiv:2409.15647*, 2024.

Yunzhen Feng, Elvis Dohmatob, Pu Yang, Francois Charton, and Julia Kempe. Beyond model collapse: Scaling up with synthesized data requires reinforcement. *arXiv preprint arXiv:2406.07515*, 2024.

Matthias Gerstgrasser, Rylan Schaeffer, Apratim Dey, Rafael Rafailov, Henry Sleight, John Hughes, Tomasz Korbak, Rajashree Agrawal, Dhruv Pai, Andrey Gromov, et al. Is model collapse inevitable? breaking the curse of recursion by accumulating real and synthetic data. *arXiv preprint arXiv:2404.01413*, 2024.

Nate Gillman, Michael Freeman, Daksh Aggarwal, Chia-Hong Hsu, Calvin Luo, Yonglong Tian, and Chen Sun. Self-correcting self-consuming loops for generative model training. *arXiv preprint arXiv:2402.07087*, 2024.

Caglar Gulcehre, Tom Le Paine, Srivatsan Srinivasan, Ksenia Konyushkova, Lotte Weerts, Abhishek Sharma, Aditya Siddhant, Alex Ahern, Miaosen Wang, Chenjie Gu, et al. Reinforced self-training (rest) for language modeling. *arXiv preprint arXiv:2308.08998*, 2023.

Peter Hase, Mohit Bansal, Peter Clark, and Sarah Wiegreffe. The unreasonable effectiveness of easy training data for hard tasks. *arXiv preprint arXiv:2401.06751*, 2024.

Ryuichiro Hataya, Han Bao, and Hiromi Arai. Will large-scale generative models corrupt future datasets? In *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV)*, pp. 20555–20565, October 2023.

Arian Hosseini, Xingdi Yuan, Nikolay Malkin, Aaron Courville, Alessandro Sordoni, and Rishabh Agarwal. V-star: Training verifiers for self-taught reasoners. *arXiv preprint arXiv:2402.06457*, 2024.

J. Edward Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, and Weizhu Chen. Lora: Low-rank adaptation of large language models. *ArXiv*, abs/2106.09685, 2021. URL `https://api.semanticscholar.org/CorpusID:235458009`.

Audrey Huang, Adam Block, Dylan J. Foster, Dhruv Rohatgi, Cyril Zhang, Max Simchowitz, Jordan T. Ash, and Akshay Krishnamurthy. Self-improvement in language models: The sharpening mechanism, 2024. URL `https://arxiv.org/abs/2412.01951`.

Jiaxin Huang, Shixiang Shane Gu, Le Hou, Yuexin Wu, Xuezhi Wang, Hongkun Yu, and Jiawei Han. Large language models can self-improve. *arXiv preprint arXiv:2210.11610*, 2022.

Dieuwke Hupkes, Verna Dankers, Mathijs Mul, and Elia Bruni. Compositionality decomposed: How do neural networks generalise? *Journal of Artificial Intelligence Research*, 67:757–795, 2020.

Samy Jelassi, Stéphane d'Ascoli, Carles Domingo-Enrich, Yuhuai Wu, Yuanzhi Li, and François Charton. Length generalization in arithmetic transformers. *arXiv preprint arXiv:2306.15400*, 2023.

Amirhossein Kazemnejad, Inkit Padhi, Karthikeyan Natesan Ramamurthy, Payel Das, and Siva Reddy. The impact of positional encoding on length generalization in transformers. *Advances in Neural Information Processing Systems*, 36, 2024.

Nayoung Lee, Kartik Sreenivasan, Jason D Lee, Kangwook Lee, and Dimitris Papailiopoulos. Teaching arithmetic to small transformers. *arXiv preprint arXiv:2307.03381*, 2023.

Shanda Li, Chong You, Guru Guruganesh, Joshua Ainslie, Santiago Ontanon, Manzil Zaheer, Sumit Sanghai, Yiming Yang, Sanjiv Kumar, and Srinadh Bhojanapalli. Functional interpolation for relative positions improves long context transformers. *arXiv preprint arXiv:2310.04418*, 2023.

Yiming Liang, Ge Zhang, Xingwei Qu, Tianyu Zheng, Jiawei Guo, Xinrun Du, Zhenzhu Yang, Jiaheng Liu, Chenghua Lin, Lei Ma, et al. I-sheep: Self-alignment of llm from scratch through an iterative self-enhancement paradigm. *arXiv preprint arXiv:2408.08072*, 2024.

Hunter Lightman, Vineet Kosaraju, Yura Burda, Harri Edwards, Bowen Baker, Teddy Lee, Jan Leike, John Schulman, Ilya Sutskever, and Karl Cobbe. Let's verify step by step. *arXiv preprint arXiv:2305.20050*, 2023.

Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, et al. Self-refine: Iterative refinement with self-feedback. *Advances in Neural Information Processing Systems*, 36, 2024.

Sean McLeish, Arpit Bansal, Alex Stein, Neel Jain, John Kirchenbauer, Brian R Bartoldson, Bhavya Kailkhura, Abhinav Bhatele, Jonas Geiping, Avi Schwarzschild, et al. Transformers can do arithmetic with the right embeddings. *arXiv preprint arXiv:2405.17399*, 2024.

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

Xiangyu Peng, Congying Xia, Xinyi Yang, Caiming Xiong, Chien-Sheng Wu, and Chen Xing. Regenesis: Llms can grow into reasoning generalists via self-improvement. *arXiv preprint arXiv:2410.02108*, 2024.

Yuxiao Qu, Tianjun Zhang, Naman Garg, and Aviral Kumar. Recursive introspection: Teaching language model agents how to self-improve. *arXiv preprint arXiv:2407.18219*, 2024.

Philip Quirke and Fazl Barez. Understanding addition in transformers. *arXiv preprint arXiv:2310.13121*, 2023.

D Rolnick. Deep learning is robust to massive label noise. *arXiv preprint arXiv:1705.10694*, 2017.

Anian Ruoss, Grégoire Delétang, Tim Genewein, Jordi Grau-Moya, Róbert Csordás, Mehdi Bennani, Shane Legg, and Joel Veness. Randomized positional encodings boost length generalization of transformers. *arXiv preprint arXiv:2305.16843*, 2023.

Mahdi Sabbaghi, George Pappas, Hamed Hassani, and Surbhi Goel. Explicitly encoding structural symmetry is key to length generalization in arithmetic tasks. *arXiv preprint arXiv:2406.01895*, 2024.

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*, 34:6695–6706, 2021.

Ruoqi Shen, Sébastien Bubeck, Ronen Eldan, Yin Tat Lee, Yuanzhi Li, and Yi Zhang. Positional description matters for transformers arithmetic. *arXiv preprint arXiv:2311.14737*, 2023.

Changho Shin, John Cooper, and Frederic Sala. Weak-to-strong generalization through the data-centric lens. *arXiv preprint arXiv:2412.03881*, 2024.

Ilia Shumailov, Zakhar Shumaylov, Yiren Zhao, Yarin Gal, Nicolas Papernot, and Ross Anderson. The curse of recursion: Training on generated data makes models forget. *arXiv preprint arXiv:2305.17493*, 2023.

Ilia Shumailov, Zakhar Shumaylov, Yiren Zhao, Nicolas Papernot, Ross Anderson, and Yarin Gal. Ai models collapse when trained on recursively generated data. *Nature*, 631(8022):755–759, 2024.

Avi Singh, John D Co-Reyes, Rishabh Agarwal, Ankesh Anand, Piyush Patil, Xavier Garcia, Peter J Liu, James Harrison, Jaehoon Lee, Kelvin Xu, et al. Beyond human data: Scaling self-training for problem-solving with language models. *arXiv preprint arXiv:2312.06585*, 2023.

Yuda Song, Hanlin Zhang, Carson Eisenach, Sham Kakade, Dean Foster, and Udaya Ghai. Mind the gap: Examining the self-improvement capabilities of large language models. *arXiv preprint arXiv:2412.02674*, 2024.

Zhiqing Sun, Longhui Yu, Yikang Shen, Weiyang Liu, Yiming Yang, Sean Welleck, and Chuang Gan. Easy-to-hard generalization: Scalable alignment beyond human supervision. *arXiv preprint arXiv:2403.09472*, 2024.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc Le, Ed Chi, and Denny Zhou. Self-consistency improves chain of thought reasoning in language models. *arXiv preprint arXiv:2203.11171*, 2022a.

Yizhong Wang, Yeganeh Kordi, Swaroop Mishra, Alisa Liu, Noah A Smith, Daniel Khashabi, and Hannaneh Hajishirzi. Self-instruct: Aligning language models with self-generated instructions. *arXiv preprint arXiv:2212.10560*, 2022b.

Kaiyue Wen, Zhiyuan Li, Jason Wang, David Hall, Percy Liang, and Tengyu Ma. Understanding warmup-stable-decay learning rates: A river valley loss landscape perspective. *arXiv preprint arXiv:2410.05192*, 2024.

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*, pp. 11975–11986. PMLR, 2021.

Weizhe Yuan, Richard Yuanzhe Pang, Kyunghyun Cho, Sainbayar Sukhbaatar, Jing Xu, and Jason Weston. Self-rewarding language models. *arXiv preprint arXiv:2401.10020*, 2024.

E. Zelikman, Yuhuai Wu, and Noah D. Goodman. Star: Bootstrapping reasoning with reasoning. 2022. URL `https://api.semanticscholar.org/CorpusID:247762790`.

Edwin Zhang, Vincent Zhu, Naomi Saphra, Anat Kleiman, Benjamin L Edelman, Milind Tambe, Sham M Kakade, and Eran Malach. Transcendence: Generative models can outperform the experts that train them. *arXiv preprint arXiv:2406.11741*, 2024.

Hugh Zhang and David C Parkes. Chain-of-thought reasoning is a policy improvement operator. *arXiv preprint arXiv:2309.08589*, 2023.

Linfeng Zhang, Jiebo Song, Anni Gao, Jingwei Chen, Chenglong Bao, and Kaisheng Ma. Be your own teacher: Improve the performance of convolutional neural networks via self distillation. In *Proceedings of the IEEE/CVF international conference on computer vision*, pp. 3713–3722, 2019.

Hattie Zhou, Arwen Bradley, Etai Littwin, Noam Razin, Omid Saremi, Josh Susskind, Samy Bengio, and Preetum Nakkiran. What algorithms can transformers learn? a study in length generalization. *arXiv preprint arXiv:2310.16028*, 2023.

Yongchao Zhou, Uri Alon, Xinyun Chen, Xuezhi Wang, Rishabh Agarwal, and Denny Zhou. Transformers can achieve length generalization but not robustly. *arXiv preprint arXiv:2402.09371*, 2024.

A Detailed Discussion of Related Work

Length Generalization. While Transformers (Vaswani et al., 2017) have achieved remarkable success, they often struggle with length generalization—where a model trained on problems of fixed length fails to extrapolate to longer sequences (Anil et al., 2022). Addressing this limitation is crucial, as poor length generalization indicates that language models may not fully understand the underlying task. Zhou et al. (2023) hypothesize that Transformers are more likely to length generalize on tasks with small RASP-L complexity. They demonstrate that tasks such as reverse addition and copying have low RASP-L complexity, making them easier to length generalize, whereas forward addition poses a greater challenge.

Several approaches have been proposed to improve length generalization, particularly in arithmetic tasks. These include modifications to positional embeddings, such as Abacus embeddings (McLeish et al., 2024), NoPE (Kazemnejad et al., 2024), FIRE (Li et al., 2023), and pairwise positional encodings (Sabbaghi et al., 2024; Cho et al., 2024), randomized positional encodings (Ruoss et al., 2023; Zhou et al., 2024). Other methods focus on architectural changes, such as introducing looping mechanisms (Fan et al., 2024) or incorporating hand-crafted bias corrections in attention score matrices (Duan et al., 2023). Additionally, input modifications, such as index hinting, have been explored to enhance generalization (Zhou et al., 2023; 2024). Beyond arithmetic, length generalization has also been studied in the context of size generalization in graph-based tasks (Yehudai et al., 2021).

In contrast, our approach adheres to the standard transformer architecture without introducing significant modifications to architecture, positional encodings, or input structure. A key distinction lies in the training methodology. While prior approaches typically rely on fixed-length training datasets without further updates to model weights, we iteratively update model weights on self-generated datasets, enabling the model to progressively improve and extend its generalization capabilities.

Our multiplication results have relevance with findings by Jelassi et al. (2023), who showed that dataset priming (adding a small number of labeled long-sequence examples) can enable length generalization[1] for multiplication (although this is not strictly out-of-distribution). Our approach of incorporating accurate, self-generated out-of-distribution data via filtering can be seen as an automated form of dataset priming. Furthermore, while our approach uses chain-of-thought (CoT) data for multiplication, we believe it is possible to length generalize on non-COT multiplication as well, by incorporating methods like Deng et al. (2024) to help the model iteratively internalize the CoT steps.

Easy-to-hard Generalization. Our self-improvement framework operates in a setting where human annotation is provided for easier tasks, enabling generalization to harder tasks with no additional supervision. This paradigm, often referred to as easy-to-hard generalization (Schwarzschild et al., 2021; Bansal et al., 2022; Burns et al., 2023; Hase et al., 2024; Sun et al., 2024), leverages the transfer of learned policies or reward models from simpler problems to more challenging ones. For instance, Zhang et al. (2024) study this phenomenon in chess, showing that chess transformers can sometimes outperform all players in the training dataset. Similarly, Sun et al. (2024) finds that a reward model trained on easier mathematical problems can be effectively transferred to harder problems, facilitating generalization through reinforcement learning. Shin et al. (2024) identifies overlap data points—instances containing both easy and hard patterns—as a key mechanism for weak-to-strong generalization, allowing weak models to pseudolabel easier patterns while stronger models use these labels to learn harder patterns. Our work shows that a similar mechanism emerges naturally within self-improvement, where progressively increasing difficulty enables models to generate useful supervision signals for harder tasks without explicit human intervention.

Self Improvement. When high quality training labels are not available, training on self-generated labels is an efficient way to extract more capabilities from the model. Usually, this involves generating candidate labels, pruning wrong labels through verification or filtering, and retraining with self-generated data. ReST (Gulcehre et al., 2023) and I-SHEEP (Liang et al., 2024) propose self-improvement as an general purpose alternative to reinforcement learning from human feedback (RLHF), while Yuan et al. (2024) propose "self-rewarding" model that generates its own instruction tuning set. The self-improvement framework has been applied to a wide range of tasks. For example,

[1]they consider encoder-decoder architecture which differs for our decoder-only model

Zhang et al. (2019) replaces an expensive teacher distillation with self-distillation for image recognition tasks. In LLM reasoning domains, Singh et al. (2023), Huang et al. (2022) and Zelikman et al. (2022) bootstrap complex reasoning capabilities by asking models to generate rationales for unlabeled questions and training on self-generated rationals that yielded correct answers. Similarly, Zhang & Parkes (2023) shows self-improving using chain-of-thought (COT) data sampled from the model allows generalization of the integer addition task to more digits. For coding tasks, Chen et al. (2023) teaches LLMs to self-debug with feedback using self-generated code explanation and unit test execution results. In mathematics, PatternBoost (Charton et al., 2024) shows that transformers can discover unsolved mathematical constructions of various problems using an algorithm that alternates between sampling constructions from the model (local search) and training on self-generated data (global learning). Finally, aiming at understanding the self-improvement process, (Bansal et al., 2024) emphasizes the effectiveness of smaller models; Song et al. (2024) studies the generation-verification gap as a key quantity governing the self-improvement process, while Huang et al. (2024) introduces the "sharpening mechanism", where training on best-of-N responses from the model amortizes maximum likelihood inference and leads to higher quality outputs.

Model Collapse Recent research has extensively investigated the phenomenon of model collapse, where repeated training on a model's own outputs leads to performance degeneration and a loss of the true underlying data distribution (Shumailov et al., 2024; Hataya et al., 2023; de Arcaute et al., 2023; Shumailov et al., 2023; Alemohammad et al., 2023; Briesch et al., 2023).

Shumailov et al. (2024) provide evidence that iterative training on model-generated data, without filtering, results in rapid degeneration and forgetting of the true data distribution. They emphasize the importance of preserving original data sources over time. Similarly, Shumailov et al. (2023) show that the tails of the original content distribution diminish after repeated self-training, while Zhang & Parkes (2023) highlight the error avalanching effect, where errors compound as models are trained on their own generated data.

Despite its apparent inevitability, several strategies have been proposed to mitigate model collapse. Research shows that the risk of collapse diminishes when the initial model closely approximates the true data distribution (Bertrand et al., 2023), or when real data is retained throughout training rather than being fully replaced (Gerstgrasser et al., 2024; Dohmatob et al., 2024; Briesch et al., 2023). Additionally, Gillman et al. (2024); Feng et al. (2024) suggest using reliable verifiers during self-training to ensure high-quality self-generated data, further reducing the likelihood of collapse.

Our approach addresses these challenges by maintaining a core labeled dataset throughout training, consisting of examples of limited length or difficulty. Synthetic data, generated incrementally by the model, is added in a controlled manner. By incorporating unsupervised filtering techniques such as length filtering and majority voting, we ensure the quality of self-generated data. Our framework builds upon prior findings by preserving clean data and selectively incorporating synthetic data.

Additionally, our results in Section B.6 align with findings from Rolnick (2017), which demonstrate that deep neural networks are robust to significant label noise in image classification tasks. Additionally, Bayat et al. (2024) recently emphasized that memorization alone does not harm generalization; rather, the combination of memorization with spurious correlations is what undermines learning. Our results suggest that despite memorizing past mistakes, the self-improvement framework remains effective, provided that incorrect samples do not dominate the training distribution.

B Additional Results

B.1 String Copy & String Reverse

Copying and reversing a given input string is another task that is considered hard for vanilla transformers (Anil et al., 2022; Zhou et al., 2023). The input string consists of digits from 0 to 9.

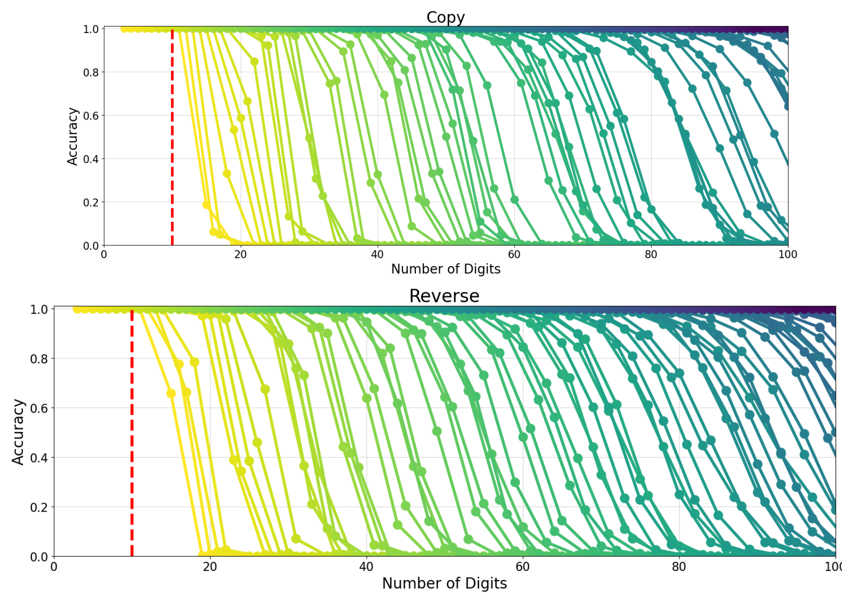


Figure 11: Results on string manipulation tasks. (Top) Copying task. (Bottom) Reversing task. The model, initially trained on strings of length 1 to 10, generalizes to strings of over 120 digits through self-improvement.

Results. Similar to reverse addition task, Figure 11 demonstrates that starting with strings of length 1 to 10, the self-improvement framework enables the model to perfectly generalize to string lengths of over 120 after approximately 100 self-improvement rounds.

B.2 Motivation for Data Filtering

B.2.1 Importance of Data Filtering

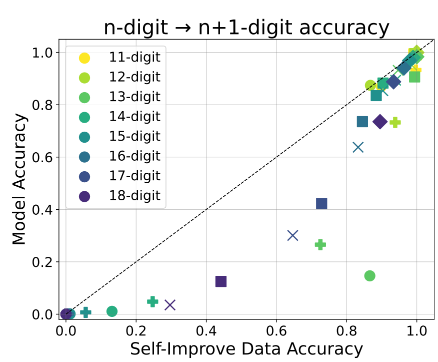


Figure 12: Effect of self-generated data accuracy on length generalization performance in the reverse addition task. Each data point represents the accuracy of the self-improve data $\mathcal{D}_r$ (on n digit addition) generated by model M_{r-1}, and the resulting $n+1$-digit performance of the trained model M_r at round r. The prevalence of points below the $y = x$ line highlights the critical importance of high-quality data for successful self-improvement.

Figure 12 demonstrates this effect in the reverse addition task. The x-axis represents the accuracy of the self-improve dataset $\mathcal{D}_r$, generated by model M_{r-1} at round r, while the y-axis shows the resulting $n+1$-digit performance of model M_r. The prevalence of data points below the $y = x$ line indicates that low-quality data diminishes performance, underscoring the need for maintaining high-quality data throughout the self-improvement process.

B.2.2 OOD RESULTS ARE OFTEN SHORT

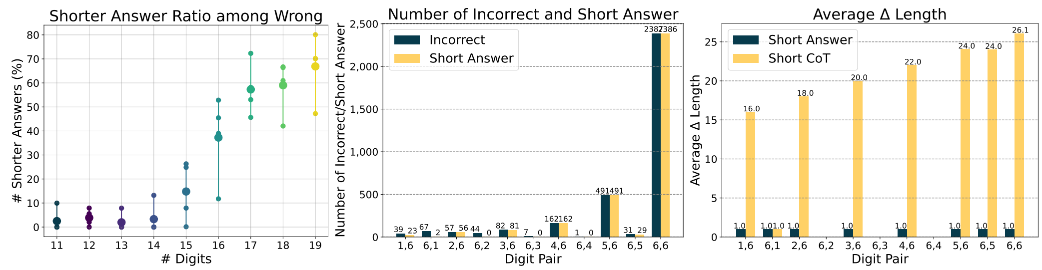


Figure 13: (Left) Reverse addition task: the proportion of shorter answers among incorrect predictions increases with each round. (Mid & Right) CoT-multiplication task with majority voting: (Mid) The majority of incorrect answers are short. (Right) The average length discrepancy of short answers compared to the correct answer or the CoT reasoning part.

Figure 13 illustrates this phenomenon for both the reverse addition and CoT-multiplication tasks. In reverse addition (Left), as the number of digits in the training data increases (or as self-improvement rounds progress), the proportion of incorrect self-generated data where the answer is shorter than the correct label length also increases. Similarly, for CoT-multiplication (Mid and Right), most incorrect answers are shorter than the correct ones. Furthermore, in cases where the answers are shorter, the outputs often miss one or more reasoning steps in the chain-of-thought (CoT) reasoning process.

B.2.3 MAJORITY VOTING LEVERAGES LABEL DIVERSITY

Self-improvement relies on the model's ability to generalize to slightly harder problems. However, this generalization is not always robust and can vary significantly across different training instances (Zhou et al., 2024). Majority voting mitigates this variability by aggregating predictions across multiple independently trained models, thereby improving the reliability of self-generated labels.

To illustrate this variability, Figure 14 shows test accuracy across five models trained with different random seeds on the initial training dataset containing up to 5-by-5 multiplication. Even when trained on identical training data, models exhibit substantial performance differences in extrapolation. Similarly, Figure 15 demonstrates that this variability persists even when models are trained from the same seed data.

Figure 16 demonstrates the effectiveness of majority voting in the multiplication task across five models trained with different seeds during the initial training phase on data $\mathcal{D}_0$, which consists of up to 5-by-5 multiplication problems. The mean accuracy (Left) is relatively low, with a high standard deviation (Mid), indicating substantial variability among the models. By applying majority voting with a consensus on at least 4 out of 5 model outputs, the generated dataset quality improves significantly (Right). For example, while the 5-by-6 multiplication task achieves an average accuracy of 31% across models, the majority-voting strategy generates a dataset with 93.3% accuracy.

In practice, datasets for larger multiplications, such as 5-by-6 digits, are created after multiple rounds of self-improvement training, gradually incorporating m-by-6 and 6-by-m data with incrementally increasing m at each round.

B.2.4 ABLATIONS FOR MAJORITY VOTING

Our majority voting method requires training multiple models in parallel. In our primary setting, we train k models with different random seeds, allowing each to generate and train on its own independent self-improved dataset at every round.

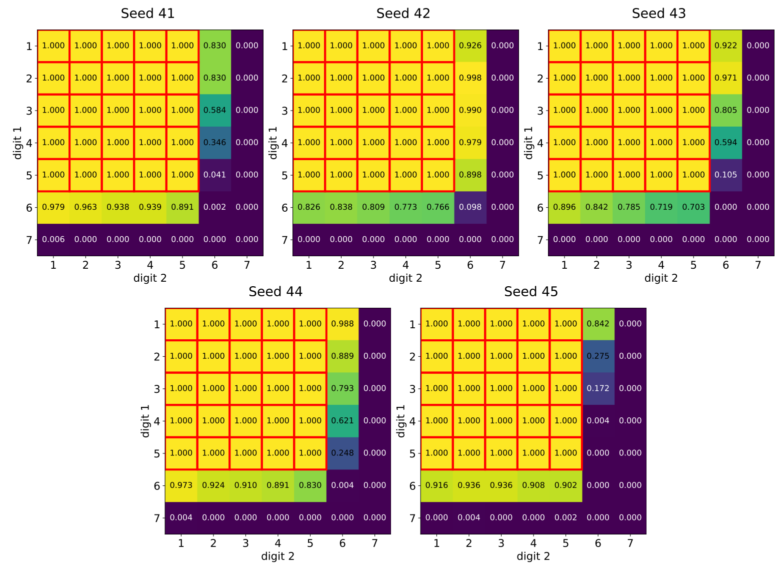


Figure 14: Test accuracy on 5 different seeds during the initial training phase. Models exhibit high variance in performance.

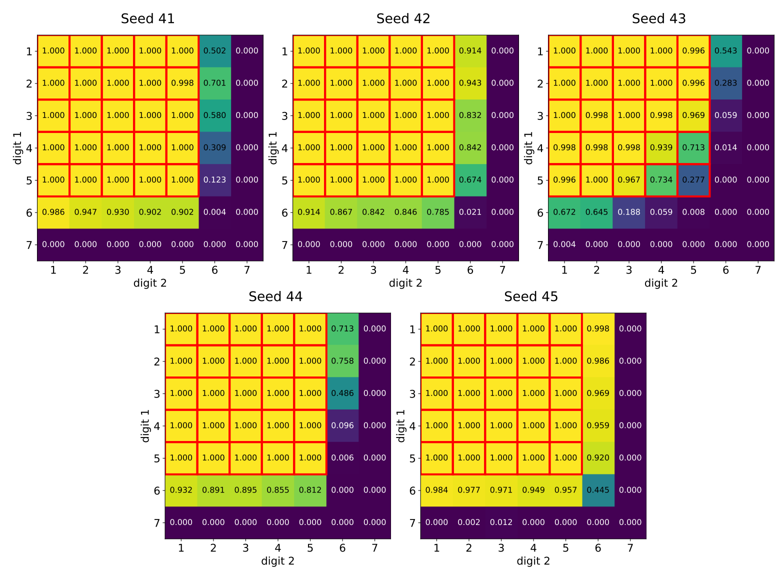


Figure 15: Test accuracy on models trained with the same seed data but different training seeds. Despite identical training data, models exhibit large variability.

To evaluate the necessity of training multiple independent models and generating separate self-improvement datasets, we compare our approach against the following baselines:

1. No majority voting, but larger self-improve data: Instead of using multiple models, we train a single model while sampling k times more self-improve data per round, ensuring that the total amount of generated data matches our main setting.
2. Shared self-improve data: We train k models with different initial seeds but subsequently train all models on the same self-improved dataset.
3. Shared initial training seed: All models are initialized from the same seed but then trained on separate self-improved datasets.
4. Our main setting: Each model is initialized with a different seed and trained on its own independently generated self-improve dataset.

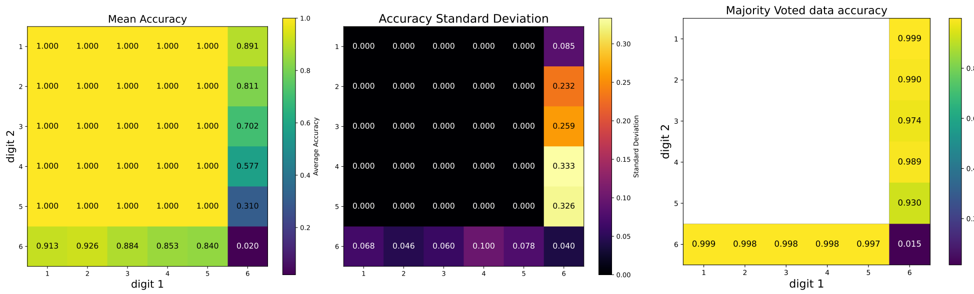


Figure 16: (Left & Mid): Mean and standard deviation of accuracy among five models trained with different seeds on the initial training round. (Right): Accuracy of majority-voted data points. Majority voting significantly boosts data quality, with 5-by-6 multiplication data accuracy increasing from an average of 31% to 93.3%

Figure 17 presents the performance of these variations, highlighting the importance of training on independently generated self-improve datasets rather than simply increasing dataset size or sharing training trajectories across models.

Table 2: Comparison of Data Cost Across Majority Voting Variants

Method	Initial Training Data Cost	Self-Improve Data Cost (Per Round)
No Majority Voting, Larger Data	1	k
Shared Self-Improve Data	k	1
Shared Initial Training Seed	1	k
Full Majority Voting (Ours)	k	k

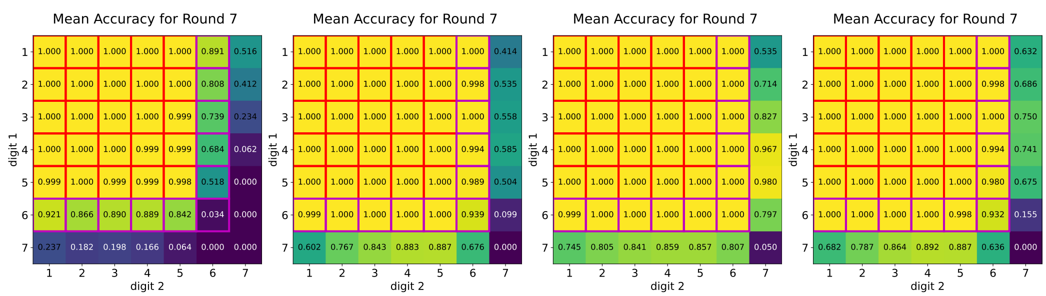


Figure 17: Ablations on majority voting. (Left) No majority voting, but larger self-improve data. (Left-Center) Majority voting with shared self-improve data. (Right-Center) Majority voting with shared initial training seed. (Right) Our primary setting with fully independent training and self-improve datasets.

We set $k = 5$ and report the average performance across five models. Figure 17 shows that simply increasing the amount of self-improvement data without filtering leads to poor performance. Surprisingly, using $5\times$ more self-improvement data per round performs even worse than using less data (Figure 32), consistent with our findings in Section B.6.3.

Additionally, majority voting with shared self-improve data (second panel from the left) underperforms in OOD compared to models trained on separate self-improve datasets. This suggests that model diversity—enabled by training on different self-improve data—may be important for majority voting to be effective.

On the other hand, comparing the right two panels in Figure 17, where the difference lies in whether the base models were trained on different labeled data $\mathcal{D}_0$, we find minimal differences in OOD performance. This may be due to the large size of the initial training set (5M examples), which provides sufficient diversity. Furthermore, as Figure 15 shows, models trained on the same initial dataset but with different training seeds still exhibit substantial variability, suggesting that model diversity can emerge from different training trajectories alone.

B.3 Additional Results on Mazes

B.3.1 Increasing the Number of Nodes

Another approach to increasing task difficulty is to expand the number of nodes in the graph while keeping the number of hops fixed at $h = 9$. We begin by training the model on a labeled dataset containing paths of fixed number of hops $h = 9$, and nodes $N = 10$ to 30. In each self-improvement round, the number of nodes is increased by 3.

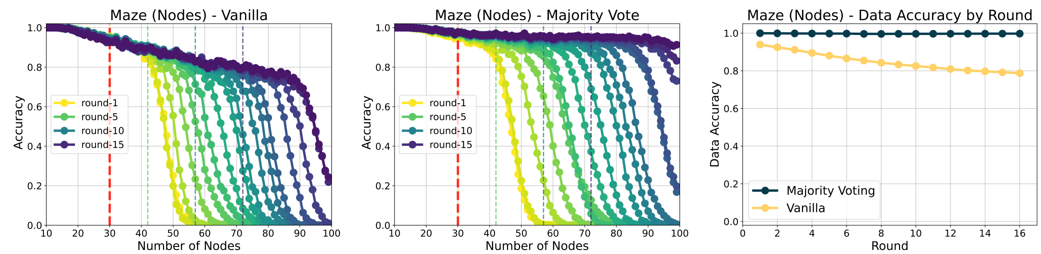


Figure 18: Maze-solving with increasing nodes ($h = 9$ hops). Models are trained on graphs with up to 30 nodes and generalized by incrementally increasing the number of nodes by 3 per round. Majority voting improves generalization to larger graphs.

Results. As shown in Figure 18, training without filtering leads to gradual performance degradation, whereas majority voting preserves high-quality data, maintaining a self-improvement accuracy above 99.7% and enabling generalization to larger graphs with 9 hops.

While these experiments focus on fixing one dimension (number of hops or number of nodes) and increasing the other, alternating between increasing the difficulty in both dimensions is expected to generalize the maze-solving task to handle larger graphs and longer paths simultaneously.

B.3.2 Verification Filters on Mazes

Solving the shortest path problem can be computationally expensive, but verifying the correctness of a given solution is significantly simpler. A valid path can be verified by traversing the sequence and ensuring three conditions: 1) each move is valid, meaning the path follows adjacency constraints; 2) the final destination matches the intended goal; and 3) no nodes are repeated, confirming that the solution is indeed the shortest path.

Self-improvement frameworks commonly incorporate verifiers to filter self-generated data, often leveraging trained models or reward models (Zelikman et al., 2022; Singh et al., 2023; Hosseini et al., 2024; Lightman et al., 2023). While our primary focus is not on training or designing an additional verification mechanism, we investigate the effectiveness of using an external verifier as a data-filtering method.

To this end, we evaluate an oracle verifier that enforces two essential constraints: 1) move validity, ensuring that every transition in the generated solution adheres to the adjacency constraints of the maze, and 2) end validity, confirming that the final node in the solution corresponds to the correct destination. We compare the effectiveness of this oracle-based filtering against self-improvement without data filtering and majority-voting-based filtering to assess its impact on performance and stability.

Results. Figure 19 shows results for mazes with increasing hops, increasing nodes, and three different verification strategies: checking moves, checking end validity, and checking both. As expected, verification improves data quality and serves as an effective filtering technique in self-improvement. Notably, verifying move validity proves to be significantly more effective than verifying only the correctness of the end node. Interestingly, however, majority voting—a strategy that does not rely on an external verifier—performs comparably to verification-based filtering. This suggests that self-consistency mechanisms alone can be sufficient for maintaining high-quality training data.

Additional results, including finer-grained analysis of move validity and end validity beyond exact match accuracy, are provided in Appendix D.0.2.

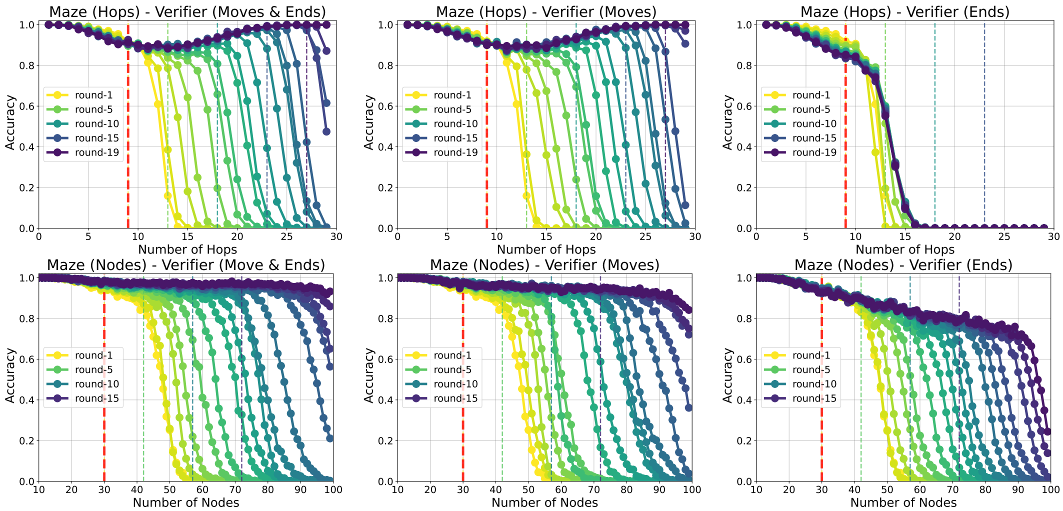


Figure 19: (Top) Increasing hops. (Bottom) Increasing nodes. (Left) Verifier on both moves and ends. (Middle) Verifier on moves only. (Right) Verifier on ends only. Verifier-based filtering improves self-improvement performance, with move validation proving more effective than end validation alone. Interestingly, majority voting performs on par with oracle verification, suggesting that self-consistency mechanisms can serve as effective alternatives to explicit verification.

B.4 Accelerated Self-Improvement for Multiplication

We validate the accelerated self-improvement (Section 7.2) setting to the task of multiplication. For the multiplication task, we observe similar enhancement using an accelerated schedule, as depicted in Figure 20. Under the standard schedule, reaching 10-by-10 multiplication from 5-by-5 requires 41 self-improvement rounds, incrementally increasing one operand by 1 at a time. With the accelerated schedule, we progressively sample more operand pairs as self-improvement proceeds, reducing the required rounds to 19 while achieving perfect test performance (see Figure 37 for full results). The settings for multiplication follow the setting in Section 6.

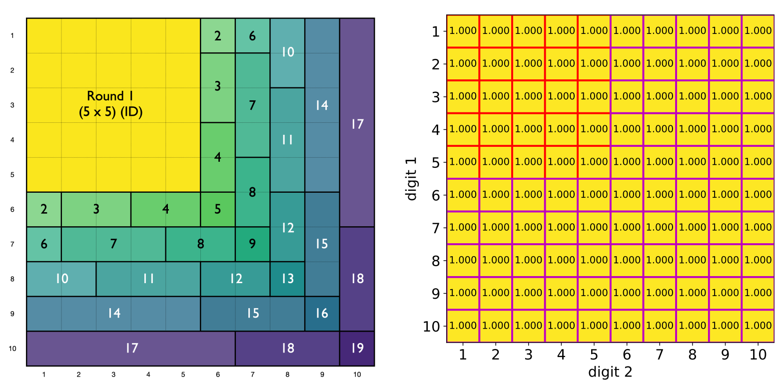


Figure 20: Accelerated self-improvement in multiplication. (Left) Accelerated schedule for multiplication. The rows and columns represent the number of digits in the two operands of the multiplication task. The number within each cell indicates the self-improvement round in which the corresponding digit pair is included for training. (Right) Results at round 19. Controlled scheduling progressively incorporates more digit pairs in each round, accelerating the self-improvement process.

B.5 Results on Pretrained Models

Figure 21 shows the self-improvement results for LoRA finetuning Llama-1B and Llama-3B on the reverse addition task. Pretrained models show more extrapolation than from-scratch models.

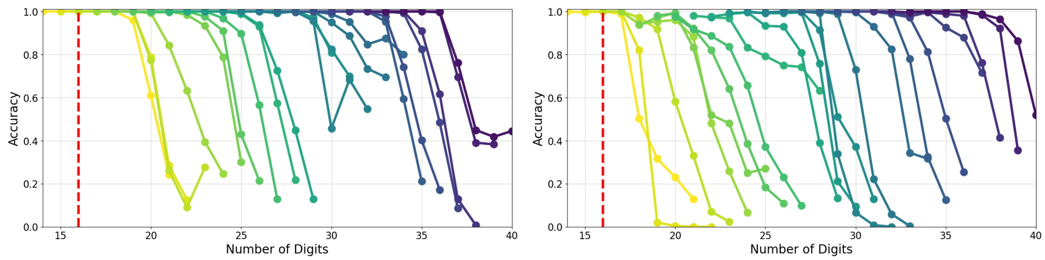


Figure 21: Reverse addition results for pretrained models. (Left) Llama-1B model. (Right) Llama-3B model. Larger models exhibit better extrapolation performance across rounds of self-improvement.

B.6 Analysis on Errors

B.6.1 Error Avalanches in Self-Improvement

Out-of-distribution (OOD) generalization is highly sensitive to inaccuracies in self-generated data. Figure 22 highlights a key challenge in this setting: errors in n-digit training data propagate to $n+1$-digit examples, degrading performance in later rounds. This is evident from data points falling below the $y = x$ line, indicating that self-improvement data is becoming progressively less reliable.

This cascading effect, known as an *error avalanche*, compounds over successive self-improvement rounds, leading to a gradual collapse of the training process. As inaccuracies accumulate, the model's self-generated labels become increasingly erroneous, reducing the effectiveness of future training. Without effective data filtering or correction mechanisms, this process eventually causes the model to fail entirely.

B.6.2 Simulating the Error Avalanche

A natural question to ask at this point is, *how much error the model must accumulate to trigger an avalanche?* We investigate this question by first characterizing the model mistakes, and then injecting synthetic wrong examples in the self-improvement data.

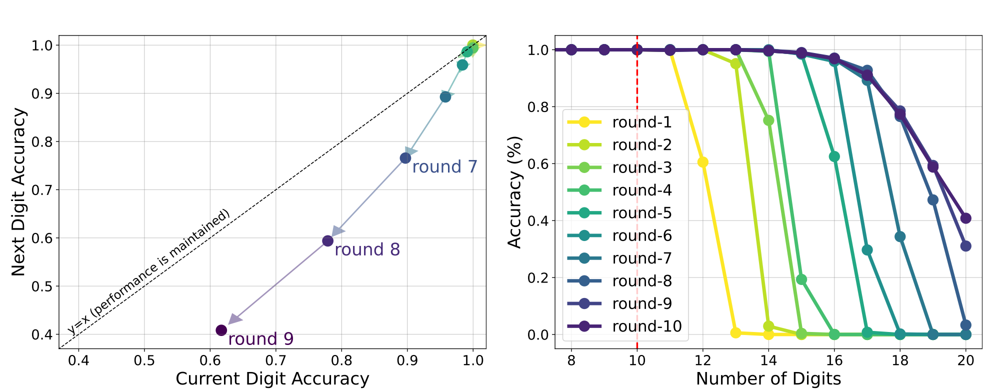


Figure 22: *Error avalanche* is a common failure case for self-improvement. As inaccuracies in self-generated data accumulate, they degrade future rounds of training, leading to eventual failure. (Left) The impact of inaccuracies in n-digit data on $n+1$-digit generalization. (Right) Gradual performance degradation over successive self-improvement rounds

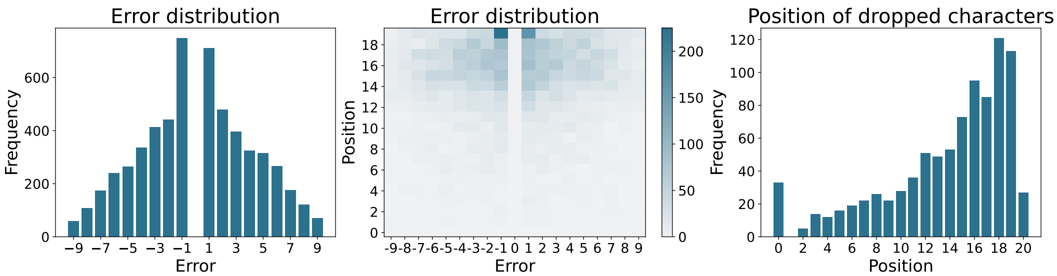


Figure 23: Patterns in model errors. (Left) Most incorrect digits are off by 1. (Middle) Errors cluster near the end of the sequence. (Right) Digit drop errors are strongly location-dependent.

Patterns in Model Mistakes. We can categorize all mistakes into two bins. At each digit position, either the model drop the digit, or outputs a wrong digit. Since these two kinds of mistakes are entangled in practice, we use a string matching algorithm to compare the model output and predictions and obtain the best guess. In figure 23, we find that digit drops by the model are concentrated near the end of the sequence, and wrong digits are most often off by 1.

Additionally, Figure 24 shows that when models generate incorrect answers, the first mismatch with the ground truth typically occurs near the final digits of the sequence (i.e., near the most significant digit in reverse addition). These observations inform our systematic error simulations, which are used to analyze the error avalanche phenomenon in Section B.6.

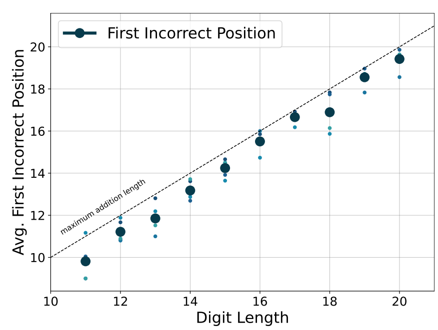


Figure 24: he first incorrect digit in model outputs tends to occur near the most significant digit in reverse addition.

Injecting Synthetic Errors. Having characterized the model mistakes, we simulate them by constructing four kinds of noises:

- Uniform: Replaces the label with a random number of the same length.
- Perturb: Randomly modifies the last three digits by ± 1.
- Drop-Digits: Randomly removes 1, 2, or 3 digits from the last three positions.
- Drop-Perturb: Combines "perturb" and "drop-digits" by first modifying digits and then randomly deleting some.

We inject these errors of varying noise levels in rounds 5 and 20 of the reverse addition task and track their effects after five subsequent self-improvement rounds. As shown in Figure 25, injecting sufficient noise into the training data causes performance on the next difficulty to crash. In particular, we find that 1) structured noises (digit drops and perturbations) are more harmful than uniform noise and 2) more rounds of self-improvement improve robustness against label noise. Additional results on uniform errors are provided in Appendix B.7.

Models can Generalize Despite Memorizing Past Mistakes Since self-improvement involves recycling model predictions into training data, an important question is whether the model continues

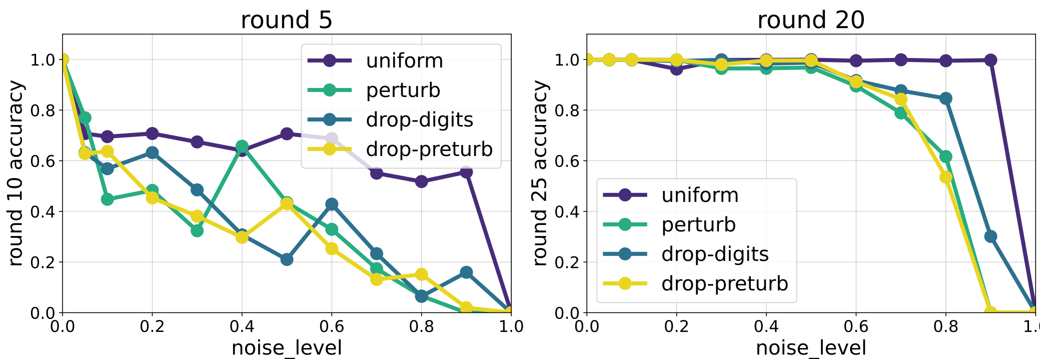


Figure 25: Simulating error avalanche. Synthetic mistakes of varying noise levels are injected at the end of rounds 5 and 20. The self-improvement process continues for 5 more rounds, and the resulting accuracy is recorded. The model tolerates errors up to a certain threshold, with greater tolerance observed in later self-improvement rounds.

making mistakes on previously incorrect examples. To investigate this, we isolate incorrect self-generated samples and evaluate the model's performance on them. As shown in Figure 26, the model struggles to rectify these errors. Accuracy on incorrect training examples decreases over successive rounds, suggesting that repeated exposure to errors reinforces them rather than correcting them.

However, memorizing past mistakes does not necessarily cause an error avalanche. The model under self-improvement often generalize to higher difficulties while treating the incorrect samples as outliers. For example, Figure 25 shows that after 20 rounds of self improvement, the model can tolerate a surprisingly large amount of label noise, from both uniform noise and structured noise. This suggests that while individual mistakes persist, they do not necessarily hinder overall generalization.

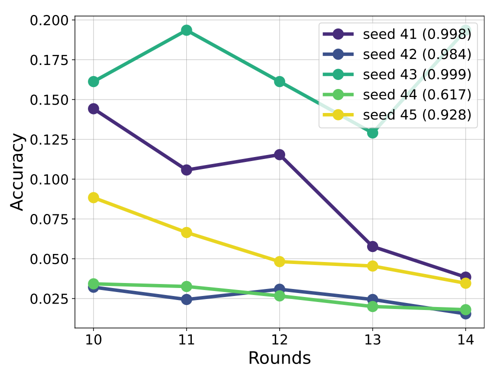


Figure 26: Models memorize their mistakes. Accuracy on incorrect training examples (of $\mathcal{D}_9$) decreases with additional self-improvement rounds, indicating that repeated exposure reinforces memorization of errors instead of correcting them.

B.6.3 Other analysis

Effect of Self-Improvement Dataset Size. We investigate how the quantity of self-generated training data impacts model performance. We first train 10 base models $M_0^{(s)}$ ($s = 1, \dots, 10$) on a supervised 1-10 digit reverse addition dataset $\mathcal{D}_0^s$, each using a different random seed. These models are categorized based on their accuracy on 11-digit addition: low-performing models (less than 98% accuracy) are represented with yellow colors, while high-performing models (more than 98% accuracy) are depicted with blue colors.

To study the effect of dataset size, we generate self-improvement datasets $\mathcal{D}_1^s = \{(x_i, M_0^{(s)}(x_i))\}_{i=1}^{N_1}$ of varying sizes ($N_1 = 10,000, 50,000, 100,000, 500,000, 1,000,000$). Each model is then trained on the combined dataset $\mathcal{D}_0^s \cup \mathcal{D}_1^s$. The number of incorrect examples in each self-generated dataset is approximately $N_1 \times (1 - (\text{11-digit accuracy of } M_0))$.

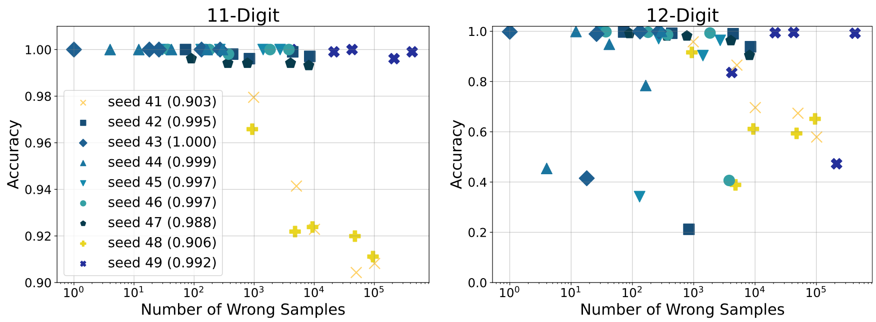


Figure 27: Effect of self-generated training data quantity and quality on model performance. Each model is trained on $\mathcal{D}_0$ (1-10 digit addition) and self-generated $\mathcal{D}_1$ (11-digit addition), then evaluated on 11-digit (in-distribution) and 12-digit (out-of-distribution) test performance. For low-performing models, increasing the quantity of self-generated data leads to degraded performance. For high-performing models, the impact of dataset size is less clear.

Results in Figure 27 show that for low-performing models, increasing the quantity of self-generated data (which is of lower quality) degrades performance on both in-distribution (11-digit) and out-of-distribution (12-digit) addition. In contrast, for high-performing models, the relationship between the number of self-generated examples and performance is less clear. The total number of 11-digit examples seen during training remains constant across experiments, with smaller datasets being repeated more often. This suggests that exposure to a greater diversity of incorrect examples can bias the model more negatively.

B.7 Additional Experiments on Label Noise and Robustness

Robustness against Random Labels To further examine the model's resilience to errors in data, we introduce randomization into the labels during training. Correct labels are replaced with random numbers of the *same length* with probabilities 1, 0.8, 0.5, 0.2, 0.1, and 0. A probability of 1 corresponds to entirely incorrect labels, while 0 indicates fully correct data.

The model is initially trained on 1-10 digit reverse addition and further trained across 8 self-improvement rounds, using self-generated data of lengths 11-18 digits. We then construct a dataset of 19-digit data with randomized labels, denoted as $\mathcal{D}_9^{\text{rand}}$. The model is fine-tuned on a combined dataset consisting of the original dataset $\mathcal{D}_0$, self-improved datasets $\mathcal{D}_1, \ldots, \mathcal{D}_8$, and $\mathcal{D}_9^{\text{rand}}$.

Results in Figure 28 show that the models can tolerate substantial random label noise, maintaining robust performance even when up to 80% of the training data is corrupted. This demonstrates the model's resilience to random errors in the training data and its ability to self-correct such mistakes during learning.

Model Bias vs. Random Labels. Interestingly, biases in self-generated data are more detrimental than uniformly random label noise. As shown in Figure 28, models trained with self-improved data perform worse than random-labeled data of comparable accuracy, given the same dataset size and fine-tuning steps. This suggests that the inherent biases in self-generated data hinder generalization more than randomly introduced noise.

These observations align with findings from Bayat et al. (2024), which highlight that memorization alone does not harm generalization; instead, the combination of spurious correlations undermines learning. Despite memorizing mistakes in self-generated data, the model's overall performance at the same difficulty level often exceeds the quality of the training data.

B.8 Does the Model Truly Learn Addition?

When the two operands are sampled randomly, the probability of encountering an instance with a carry chain length of N decays exponentially with N. Under this sampling strategy, the model may

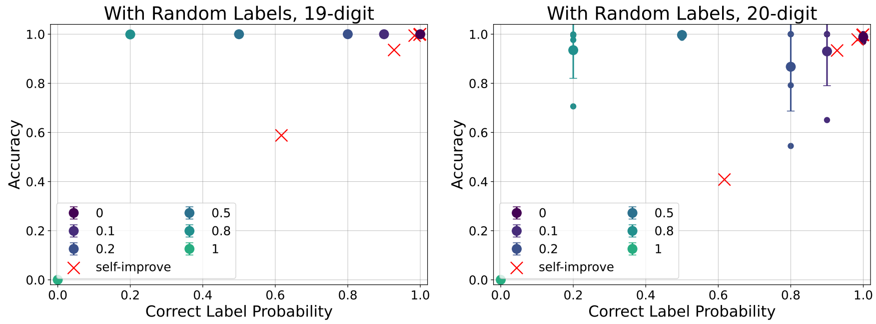


Figure 28: Effect of training on randomized labels. The model is trained on 1-10 digit data, further fine-tuned on 11-18 digit self-generated data over 8 self-improvement rounds, and additionally fine-tuned on 19-digit data with varying probabilities of random label replacement. (Left) Accuracy on 19-digit data. (Right) Accuracy on 20-digit data. The results demonstrate that while the model can self-correct random errors, biases from self-improved data can result in worse performance compared to models trained on random-labeled data of similar accuracy.

rarely, if ever, see "hard[2]" instances of addition, as illustrated in Figure 29. To address this, we manually construct a test dataset to include at least 500 examples for each maximum cascading carry length. This ensures that the evaluation captures the model's ability to handle harder instances of addition.

The results in Figure 30 show that the model is capable of performing additions with up to 20 cascading carries, even though it has never encountered such cases during training. This demonstrates that the model can generalize to harder instances of addition despite being trained predominantly on easier examples.

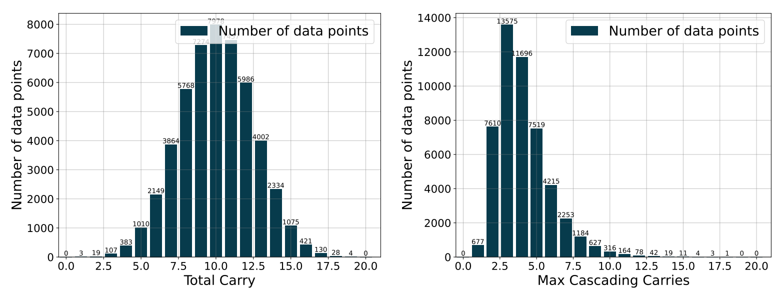


Figure 29: Number of carries in the self-improve dataset of 20-digits. The models does not see examples of high numbers of carry during training.

[2]we define hard instance of addition to be cases with multiple numbers of cascading carries (Quirke & Barez, 2023)

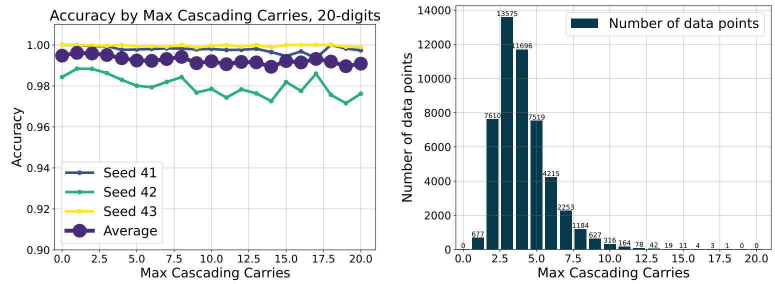


Figure 30: Performance of the model at round 10 (trained with self-generated data up to 20 digits). (Left) Accuracy as a function of the maximum cascading carries. (Right) Number of examples with each maximum cascading carry length in the self-improve training dataset. Models can successfully perform hard - with a high number of cascading carries - addition tasks even without encountering such examples in the training dataset.

C Experimental Setup

C.1 Model

For all experiments, we use a Decoder-only Transformer architecture. Specifically, for all experiments except for pretrained models settings, we use the Llama architecture (AI@Meta, 2024), except we remove the rotary positional encoding. For the inputs format, we have one example per line, and stack all example on the batch dimension. Since the examples can have variable length, we pad each line on the right to the maximum length in the batch. We exclusively use a character level tokenizer. For pretrained models, we replace the default tokenizer with our character tokenizer, while keeping the embedding component of the pretrained model unchanged.

Table 3: Model Parameters

Model	Self-Attn Layers	Num Heads	Embedding Dim
From-Scratch	6	6	384
Llama 3 1B	24	16	1024
Llama 3 3B	32	32	2048

C.2 Data Formats and Data Sampling

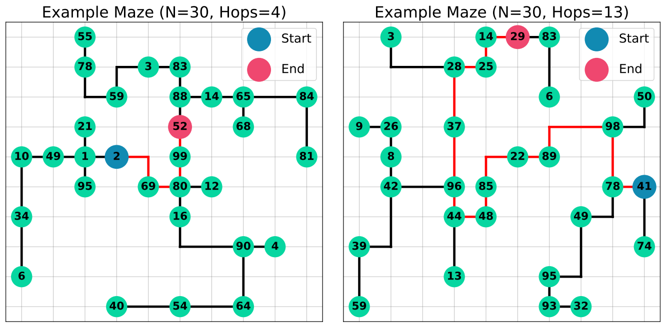
C.2.1 Data Generation and Sampling

We generate an initial supervised training dataset $\mathcal{D}_0$ of up to a fixed difficulty level d_0 by uniformly sampling the difficulty level $d \leq d_0$, followed by independent sampling of the data conditioned on the difficulty. Denoting the input as x_i, labels as y_i,

$$\mathcal{D}_0 = \{(x_i, y_i)\}_{i=1}^{N_0}, \quad \text{where Difficulty}(x_i) \leq d_0.$$

For arithmetic tasks such as addition or multiplication, each problem instance is represented as a tuple $x_i = (a_i, b_i)$, with $\mathcal{D}_0$ containing problems of up to d_0-digit numbers. The digit lengths (d_{a_i}, d_{b_i}) are uniformly sampled from $\{1, \ldots, d_0\}^2$, and the numbers a_i and b_i are uniformly sampled from the ranges $[10^{d_{a_i}-1}, 10^{d_{a_i}} - 1]$ and $[10^{d_{b_i}-1}, 10^{d_{b_i}} - 1]$, respectively.

For string manipulation tasks (e.g., copying or reversing), we uniformly sample string lengths up to d_0 and generate random sequences. Similarly, for maze-solving tasks, we uniformly sample the number of hops or total nodes in the maze and generate random graphs that satisfy these constraints. This strategy ensures balanced coverage across all difficulty levels up to d_0.



Example 1 - ID (N=30, Hops=4)
Input:
2>52#55:78-49:1,10-1:2,49,21,95-3:83,59-84:65,81-68:65-21:1-4:90-34:6,10-2:1,69-54:64,40-64:54,90-83:3,88-59:3,78-95:1-99:80,52-90:4,64,16-40:54-80:99,16,12,69-6:34-88:83,14,52-52:99,88-65:68,14,84-16:80,90-12:80-14:88,65-81:84-69:80,2-10:49,34-78:55,59=
Label:
2>69>80>99>52

Example 2 - OOD (N=30, Hops=13)
Input:
41>29#32:93-48:44,85-44:48,96,13-98:50,78,89-8:42,26-39:59,42-50:98-74:41-29:14,83-25:14,28-14:25,29-93:32,95-96:44,42,37-28:25,37,3-3:28-95:93,49-49:78,95-42:8,96,39-59:39-22:89,85-37:96,28-83:29,6-41:78,74-78:41,49,98-89:22,98-26:8,9-13:44-9:26-6:83-85:48,22=
Label:
41>78>98>89>22>85>48>44>96>37>28>25>14>29

Figure 31: Maze-solving task with $N = 30$ nodes. (Left & Middle) Visualization of the maze task with 4 hops (ID) and 13 hops (OOD). (Right) Example of the data format: the input specifies the start and end nodes along with the graph structure, and the output lists the shortest path as hops. The labeled training dataset includes paths of up to 9 hops, with difficulty increased by adding one hop in each subsequent round.

C.2.2 MULTIPLICATION

We adopt a data format similar to Deng et al. (2024), where the input prompt is `9172*9431=`, and the label expands the multiplication into steps, such as: `17442+067801(132331)+0075180(1398490)+00091720=13976630`. Each step includes the intermediate results (in parentheses) representing partial products formed by multiplying the first operand with each digit of the second operand.

The data format is inherently asymmetrical. For example, an m-by-n multiplication requires n intermediate steps, where each step corresponds to multiplying the m-digit number by one digit of the n-digit number. Conversely, an n-by-m multiplication involves m intermediate steps of multiplying the n-digit number by each digit of the m-digit number.

C.2.3 MAZE

```python
def create_tree_with_hops_wilson(total_nodes, num_hops):
    import networkx as nx

    # Step 1: Create the main path with num_hops
    graph = nx.path_graph(num_hops + 1)

    # Step 2: Add extra nodes to the tree with random walk
    current_nodes = list(graph.nodes())
    new_nodes = list(range(num_hops + 1, total_nodes))

    while new_nodes:
        new_node = new_nodes.pop()
        # random walk to reach graph
        walk = [new_node]
        while walk[-1] not in current_nodes:
            # choose random node from current & new nodes
            random_node = random.choice(current_nodes + new_nodes)
            walk.append(random_node)
            if random_node in new_nodes:
                new_nodes.remove(random_node)
        # add edges
        for i in range(len(walk) - 1):
            graph.add_edge(walk[i], walk[i + 1])
        current_nodes.append(new_node)

    # Step 3: Set the start and end nodes for the main path
    start_node = 0
    end_node = num_hops

    return graph, start_node, end_node

def format_graph(graph, start_node, end_node):
    # Assign random labels to nodes
    node_labels = assign_labels(graph.nodes(), label_range=(1, 99))

    # Get the shortest path (in terms of edge count) from start_node to end_node
    shortest_path = nx.shortest_path(graph, source=start_node, target=end_node)

    # Format the path as a string
    path_labels = [node_labels[node] for node in shortest_path]
    path_string = ">".join(map(str, path_labels))

    # Format start and end nodes
    start_label = node_labels[start_node]
    end_label = node_labels[end_node]
    start_end_str = f"{start_label}>{end_label}#"

    # Build graph_str with end_node connections at the end
    graph_str = ""
    start_node_str = ""  # Temporary storage for the start_node part
    end_node_str = ""  # Temporary storage for the end_node part

    # randomize the order of nodes
    random_nodes = list(graph.nodes())
    random.shuffle(random_nodes)
    for node in random_nodes:
        node_label = node_labels[node]
        # randomize the order of neighbors
        random_neighbors = list(graph.adj[node])
        random.shuffle(random_neighbors)
        neighbor_labels = [node_labels[neighbor] for neighbor in random_neighbors]
        graph_str += f"{node_label}:" + ",".join(map(str, neighbor_labels)) + "-"

    # Combine everything, placing the end_node last
    graph_str = start_node_str + graph_str + end_node_str

    return start_end_str + graph_str[:-1] + "=", path_string, node_labels
```

Listing 1: Code for the maze format generation used

C.3 Experimental Settings

C.3.1 Hyperparameter Configurations

In this section, we provide a detailed overview of the hyperparameter configuration used in our experiments in Table 4 and 5. To enhance memory efficiency and training speed, we employ flash attention and tf32, bfloat16. Our experiments are run using PyTorch 2.4 and CUDA 12.1. Detailed dependencies are provided in our github repository[3]. We use Warmup stable decay (Wen et al., 2024) as the learning rate schedule. In table 4 and 5, the number of constant LR steps is equal to the total training steps minus the sum of warmup and decay steps. We use AdamW optimizer with betas (0.9, 0.99) and epsilon $1e-12$. Weight decay is fixed to 0.1 and we do not use dropout.

Table 4 shows the training hyperparameters for the initial training phase on labeled data D_0. Table 5 shows the hyperparameters for each the self-improve training rounds on $D_{1,\ldots,R}$.

Table 4: Hyperparameters for initial training on labeled data

Task	Batch Size	LR	Iterations	Warmup Iter	Decay Iter
Reverse Addition	1024	5e-4	10000	1000	2000
Reverse Addition (Llama 3 3B)	128	1e-4	1200	120	600
Reverse Addition (Llama 3 1B)	128	1e-4	1200	120	600
Copy/Reverse	1024	5e-4	5000	500	1000
Forward Addition	1024	5e-4	10000	1000	1000
Multiplication	1024	5e-5	10000	1000	2000
Maze (hops)	1024	5e-4	25000	2500	3500
Maze (nodes)	512	5e-4	12000	1200	2800

Table 5: Hyperparameters for self-improvement rounds

Input Format	Batch Size	LR	Iterations	Warmup Iter	Decay Iter
Reverse Addition	1024	5e-4	1500	0	1500
Reverse Addition (Llama 3 3B)	128	1e-4	600	0	600
Reverse Addition (Llama 3 1B)	128	1e-4	600	0	600
Copy/Reverse	1024	5e-4	500	0	500
Forward Addition	1024	5e-4	3000	0	1000
Multiplication	1024	5e-5	3000	0	1000
Maze (hops)	1024	2e-4	5000	500	1000
Maze (nodes)	512	2e-4	4000	400	1000

C.3.2 Self-Improvement Setting for each Task

Reverse Addition. The initial supervised dataset $\mathcal{D}_0$ contains 2 million examples of reverse addition, with operand lengths ranging from 1 to 16 digits. This dataset is used to train the model for 10,000 steps. In subsequent self-improvement rounds, we sample 50,000 additional training examples at each round, extending the operand length by one digit. Specifically, at self-improvement round r, the self-generated data $\mathcal{D}_r$ consists of length-$16+r$ examples produced by the model M_r. The model is fine-tuned on the combined dataset $\mathcal{D}_0 \cup \mathcal{D}_1 \cup \cdots \cup \mathcal{D}_r$ for 1,500 steps, resulting in an improved model M_{r+1}.

String Copy & String Reverse. The initial training set $\mathcal{D}_\prime$ consists of 2 million examples of strings of length 1 to 10. The vocabulary of the string is the digits 0 to 9. For each subsequent round r, we sample D_r consisting of $50,000$ examples of length $10+r$ from the model M_r. Then we continue training M_r on the combined dataset $D_1 \cup \cdots \cup D_r$ for 500 steps to obtain M_{r+1}.

Forward Addition The models are initially trained on a dataset $\mathcal{D}_0$ containing 2 million labeled examples of forward addition, with operand lengths ranging from 1 to 10 digits. This initial training phase spans 10,000 steps. In each subsequent self-improvement round, we generate 50,000 additional training examples, incrementally extending the operand length by one digit. Specifically, at self-improvement round r, the self-generated dataset $\mathcal{D}_r$ contains length-$10+r$ examples produced by the

[3] `https://github.com/JackCai1206/arithmetic-self-improve/`

model M_r. The model is then fine-tuned for 3,000 steps on the combined dataset $\mathcal{D}_0 \cup \mathcal{D}_1 \cup \cdots \cup \mathcal{D}_r$, resulting in an updated model M_{r+1}.

Multiplication The model is initially trained on 5 million n-by-n multiplication examples with $n = 5$. Directly introducing $n+1$-by-$n+1$ examples results in poor performance, hence, we adopt a more fine-grained difficulty schedule. In each self-improvement round, we incrementally increase one operand by one digit, sampling $n+1$-by-m and m-by-$n+1$ examples, where m grows from 1 to $n+1$. This gradual progression allows the model to adapt incrementally to larger operand sizes, making the transition to harder examples more manageable.

For data filtering, we use the following setting: for length filtering, we remove self-generated samples where the output length is shorter than the longest output in the batch by more than 10 tokens. This helps eliminate incorrect solutions that omit intermediate steps. For majority voting, we train five models in parallel using different random seeds and retain only those data points where at least 4 out of the 5 models produce the same output. This strategy ensures that only high-consensus, reliable data points are used for training.

Maze Solving - Increasing Hops. The model is first trained on a dataset $\mathcal{D}_0$ containing 5 million labeled maze-solving examples, where the number of nodes is fixed at $N = 30$ and paths range from $h = 1$ to $h = 9$ hops. This initial training phase spans 25,000 steps. In subsequent self-improvement rounds, we generate 50,000 additional training examples, increasing h by 1, and fine-tune the model for 5,000 steps per round. We experiment with both unfiltered training data and majority voting, where only outputs agreed upon by all 3 models are retained.

Maze Solving - Increasing Nodes. The model is first trained on a dataset $\mathcal{D}_0$ containing 5 million labeled maze-solving examples, with a fixed hop count $h = 9$ and node counts ranging from $N = 10$ to $N = 30$. This initial training lasts 12,000 steps. In self-improvement rounds, the number of nodes N is increased by 3 per round, generating 50,000 additional training examples at each step and fine-tuning for 4,000 steps. We compare training without filtering against majority voting, where only outputs agreed upon by all 3 models are kept.

Ablation Task - Pretrained Models To maintain consistency in tokenization, we use character-level tokenization instead of the default tokenizer of the Llama models. We use LoRA (Hu et al., 2021) with $r = 64$ and $\alpha = 128$ for Llama-1B, and $r = 32$ and $\alpha = 128$ for Llama-3B. In the initial round, we train for 1200 steps with a learning rate schedule that includes 10% warm-up steps to a constant learning rate of 1e-4, followed by 20% cosine decay steps to a final learning rate of 1e-6. For subsequent rounds, we train for 600 steps per round using a cosine decay learning rate schedule without warm-up, starting at 1e-4 and decaying to 1e-6.

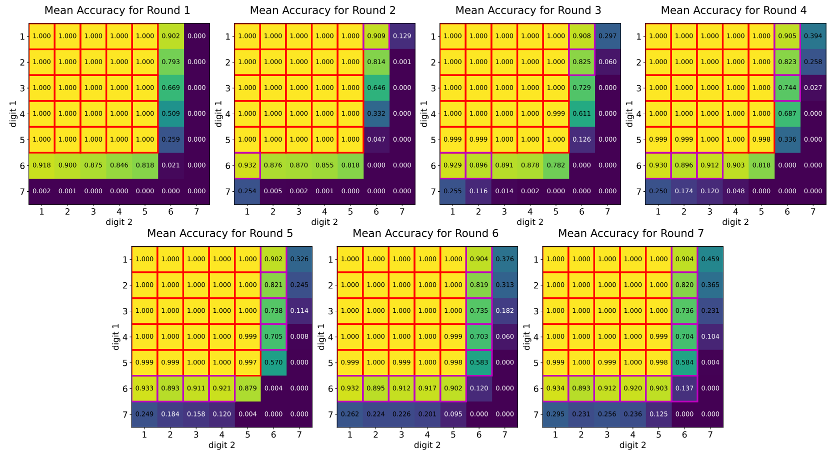


Figure 32: Results for multiplication without filtering. Each cell represents the accuracy on n-digit by m-digit multiplication. Red boxes indicate labeled in-distribution examples, while magenta boxes indicate evaluations after training on self-improved data. The model is initially trained on up to 5-by-5 multiplication. Generalizing to larger multiplications is challenging without data filtering.

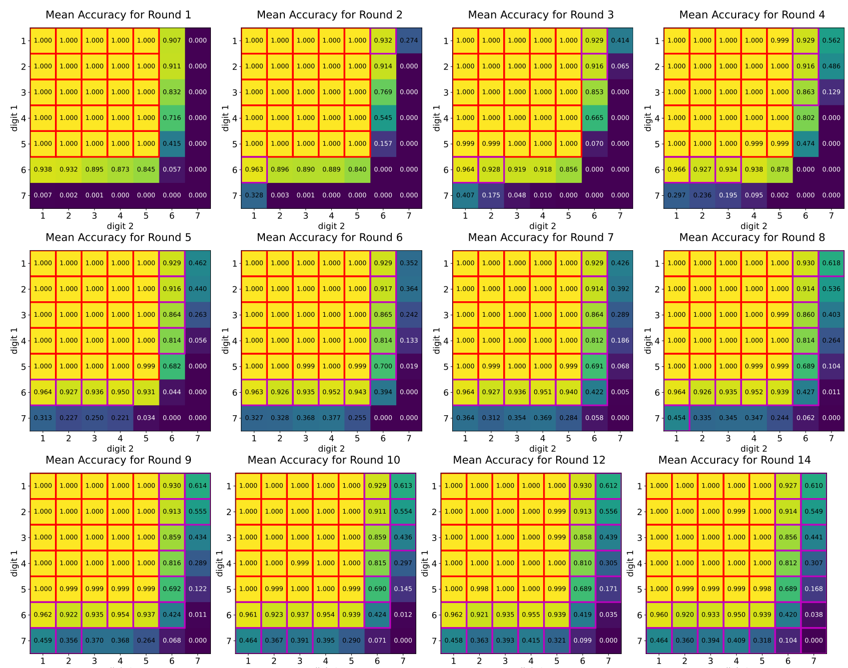


Figure 33: Results for multiplication with length filtering with length threshold of 10.

D Full Results

D.0.1 Results on Multiplication

Each figure represents the average over 5 different models.

D.0.2 Results on Mazes

We provide additional evaluation on mazes, based on the validity of moves and correctness of end nodes.

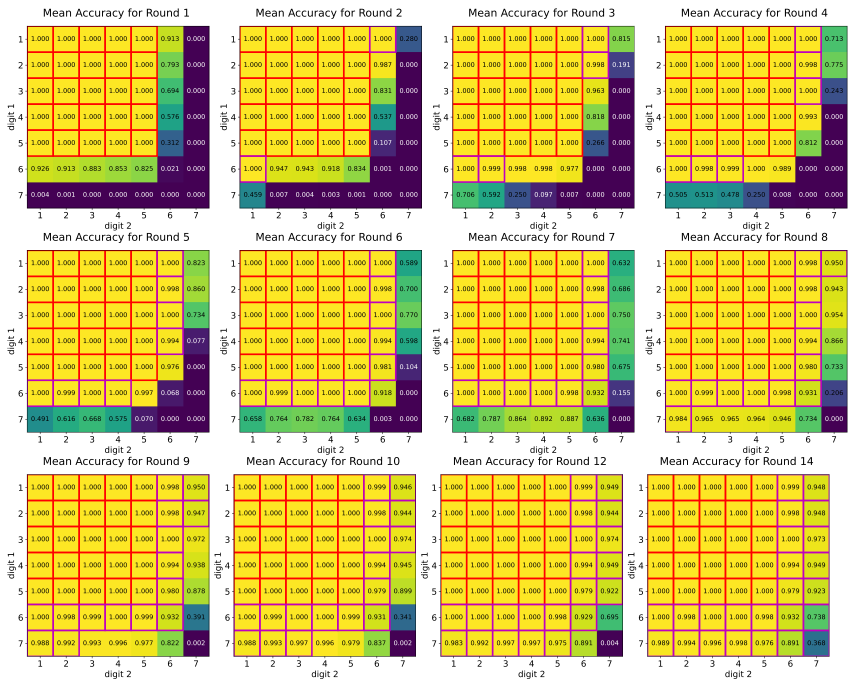


Figure 34: Multiplication with majority voting where filtering is based on agreement of at least 4 out of 5 models. Applying majority voting enables effective generalization from n-by-n to $(n+1)$-by-$(n+1)$ multiplication tasks.

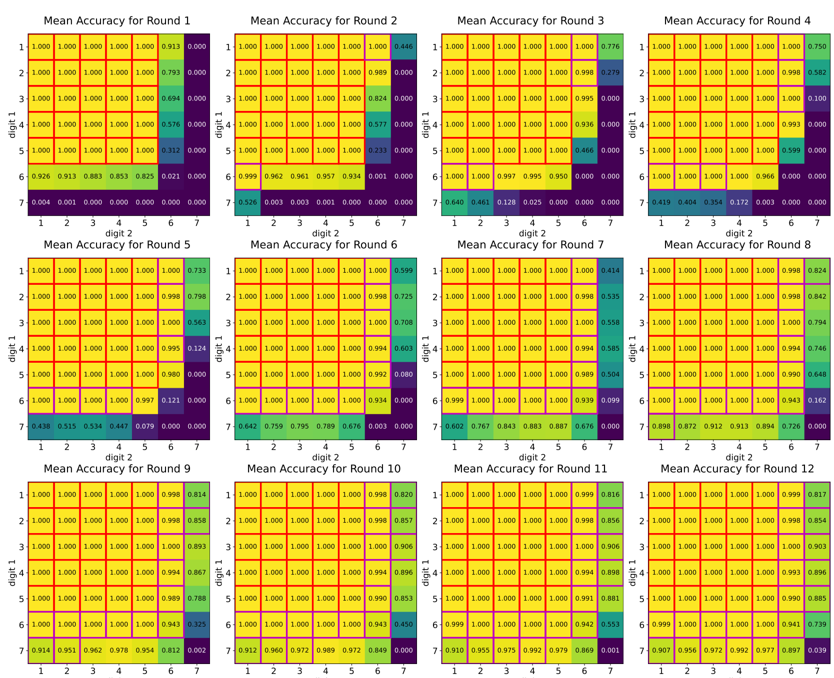


Figure 35: Multiplication task with majority voting with shared self-improve data (See Section B.2.4).

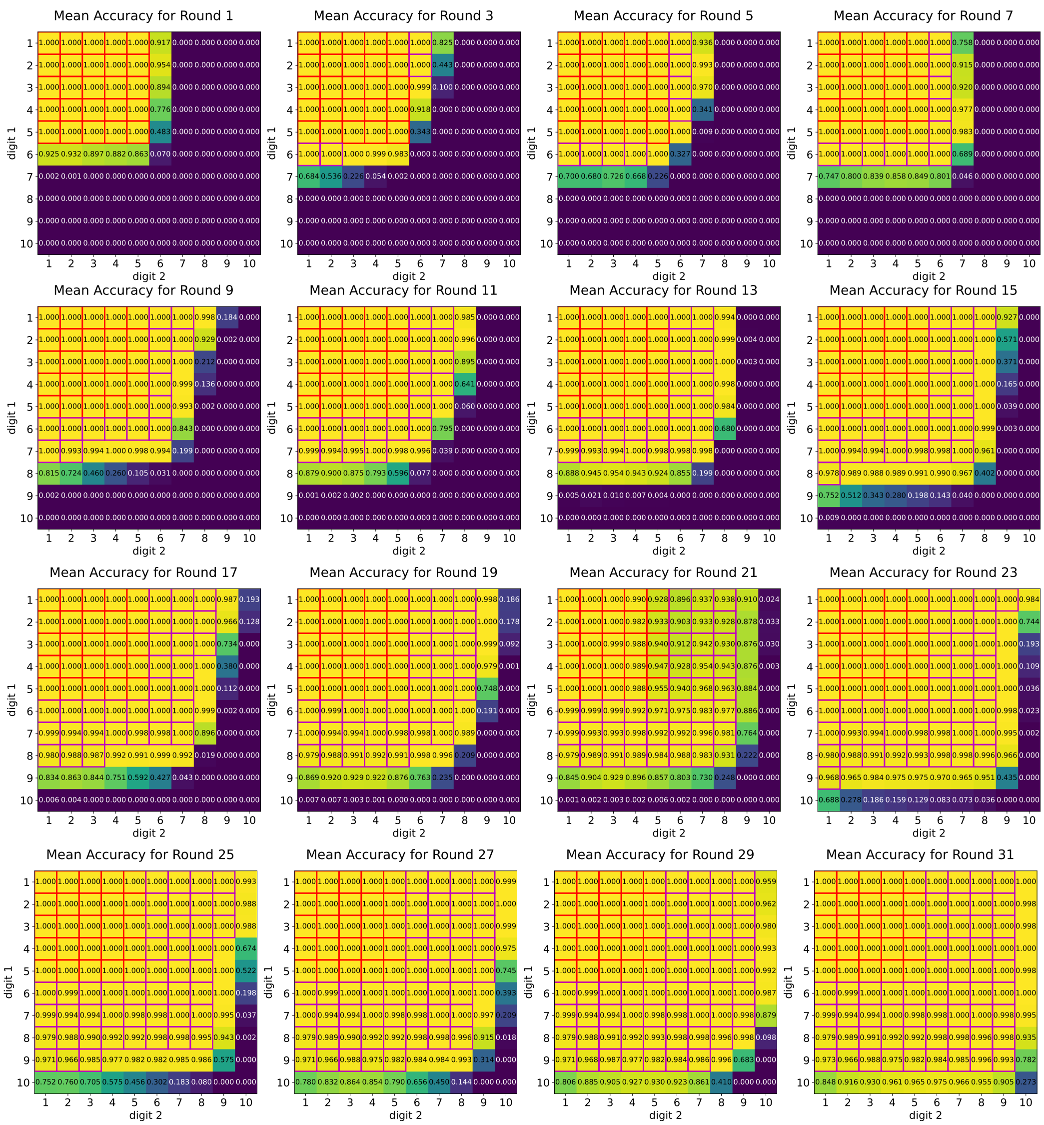


Figure 36: Combining majority voting with length filtering. This approach achieves near-perfect length generalization up to 9×9, and potentially achieving further generalization.

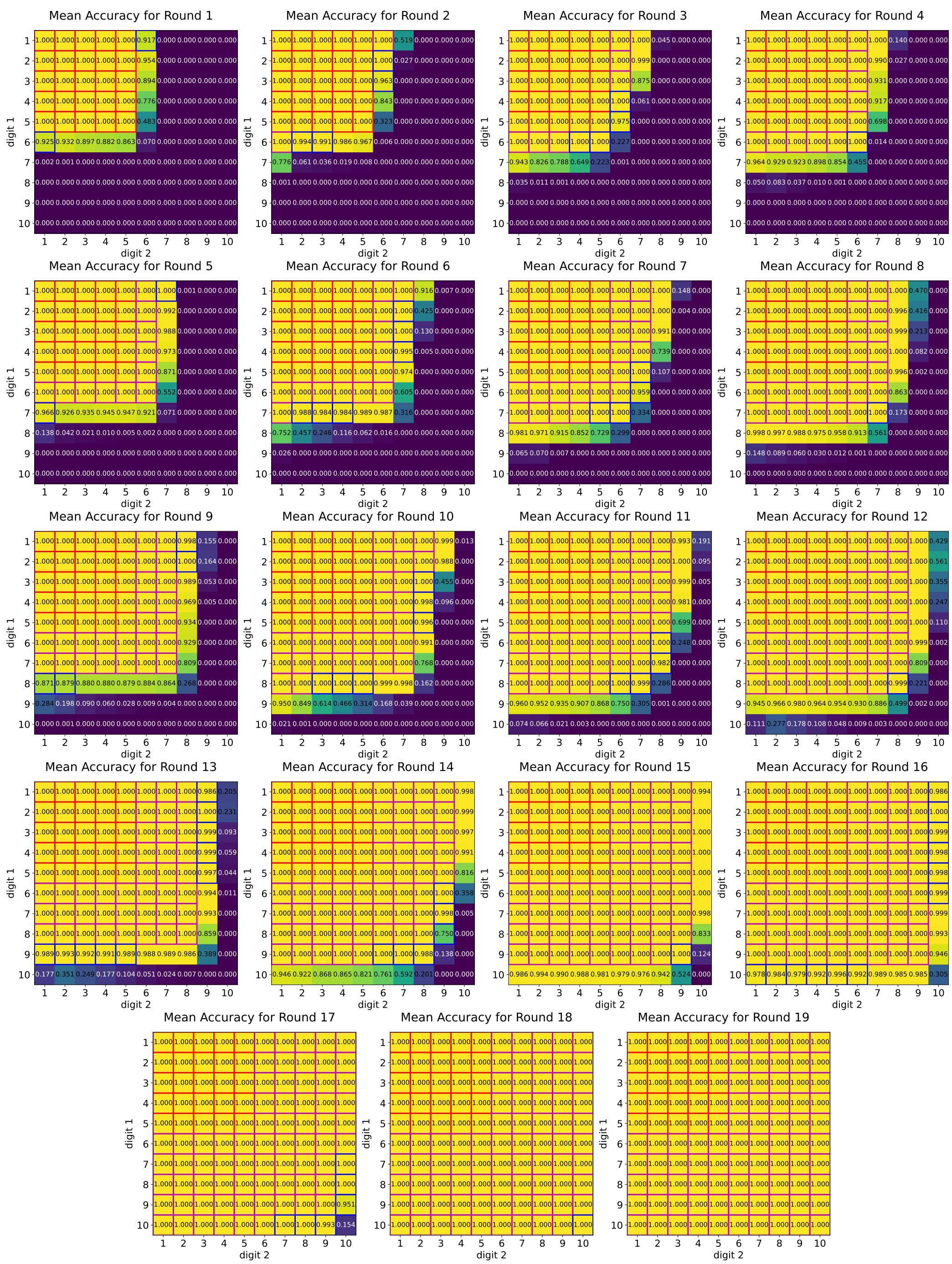


Figure 37: Accelerated multiplication. We can significantly reduce the self-improvement rounds by carefully sampling a wider range of difficulties at every round. Perfect length generalization is achieved up to 10-by-10 multiplication with 19 self-improvement rounds.

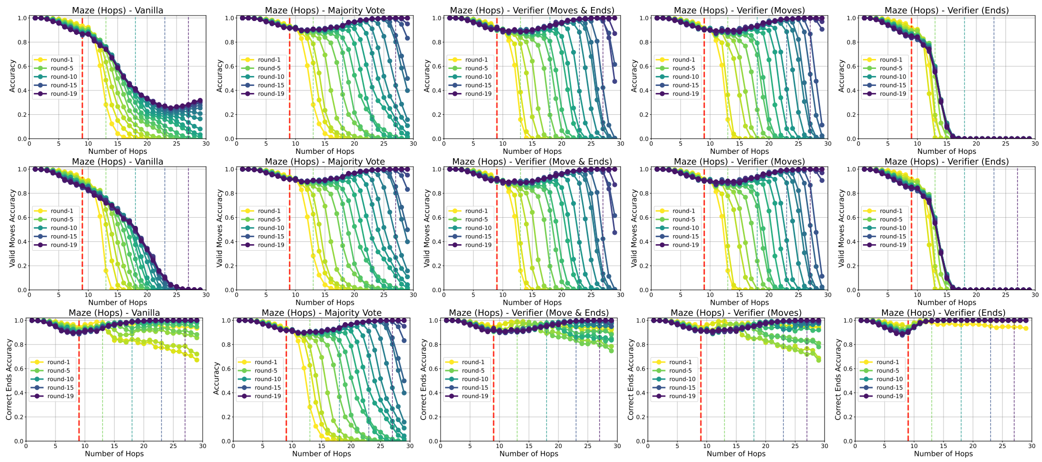


Figure 38: Maze solving task with increasing hops. (Top to bottom) Exact match accuracy, move validation accuracy, and end validation accuracy. (Left to right) No data filtering, majority voting based filtering, verifier on both moves and ends, verifier on moves only, verifier on ends only.

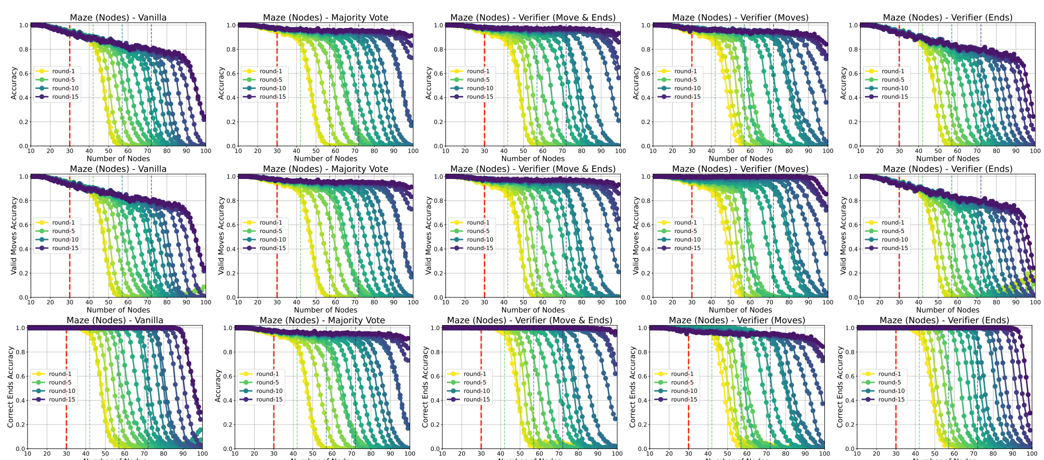


Figure 39: Maze solving task with increasing nodes. (Top to bottom) Exact match accuracy, move validation accuracy, and end validation accuracy. (Left to right) No data filtering, majority voting based filtering, verifier on both moves and ends, verifier on moves only, verifier on ends only.