Re-Tuning: Overcoming the Compositionality Limits of Large Language Models with Recursive Tuning

Eric Pasewark[1], **Kyle Montgomery**[1], **Kefei Duan**[1], **Dawn Song**[2], **Chenguang Wang**[1]*
[1]Washington University in St. Louis, [2]UC Berkeley
{eric.pasewark, kylemontgomery, d.kefei, chenguangwang}@wustl.edu
dawnsong@berkeley.edu

Abstract

We present a new method for large language models to solve compositional tasks. Although they have shown strong performance on traditional language understanding tasks, large language models struggle to solve compositional tasks, where the solution depends on solutions to smaller instances of the same problem. We propose a natural approach to solve the task recursively. Our method, Re-Tuning, tunes models to divide a problem into subproblems, solve those subproblems, and combine the results. We show that our method significantly improves the model performance on three representative compositional tasks: integer addition, dynamic programming, and parity. Compared to state-of-the-art methods that keep intermediate steps towards solving the problems, Re-Tuning achieves significantly higher accuracy and is more GPU memory efficient. Our code is available at `https://anonymous.4open.science/r/Re_Tuning_ARR_FEB_2024`.

1 Introduction

Large language models (LLM) have obtained the state-of-the-art performance on a wide set of tasks (Brown et al., 2020; Taylor et al., 2022; Chowdhery et al., 2022; Anil et al., 2023; OpenAI, 2023; Touvron et al., 2023a,b). However, recent studies (Anil et al., 2022; Dziri et al., 2023; Zhou et al., 2023b) show these models struggle to generalize to compositional tasks, where the solution depends on solutions to smaller instances of the same problem. An example task, integer addition, is shown in Figure 1. When calculating "1,234+4,567", we first break the problem into a smaller problem "234+567". After obtaining the answer to this problem, the original problem is partially solved. Similarly, to get the result of "234+567", we further divide it into solving "34+67". This recursion is the fundamental operation to solve compositional tasks. However, none of the existing approaches have explicitly captured the recursive nature of compositional tasks yet.

In this paper, we propose a recursion-based method for LLMs to improve their ability to solve compositional tasks. Intuitively, we tune LLMs to divide problems into subproblems of the same type as the original problem, solve those subproblems, and combine the results (Figure 1). More specifically, we adopt a top-down approach to solve the original problems recursively. During training, we finetune LLMs on smaller and smaller instances of the original problems, during which LLMs learn to call themselves recursively on a smaller version of the problem until reaching the base case. We empirically set the base case for different tasks. LLMs are taught to devise the solution to the problem by combining the solutions obtained from the simpler versions of the problem. We then deploy the recursively tuned LLMs to solve problems at inference time. The above procedure is referred as recursive tuning (or Re-Tuning in short).

The basic idea behind Re-Tuning is motivated by two lines of work. First, recent work (Nye et al., 2021; Anil et al., 2022; Dziri et al., 2023) show that training LLMs on high-quality scratchpad data, which includes intermediate steps towards solving a problem, can improve performance on certain compositional tasks such as integer addition and parity. Instead of using the intermediate steps to finetune models, which adds additional computation cost to the original problem, Re-Tuning divides the problems into smaller and smaller instances. Each of the instances runs independently with its own context in the associated call stack. The problem size in each context is actually reduced. The solution to each subproblem is then propagated up in the call stack to produce the final solution. A side product is that this allows models to more easily attend to the context, improving the accuracy of

*Corresponding author.

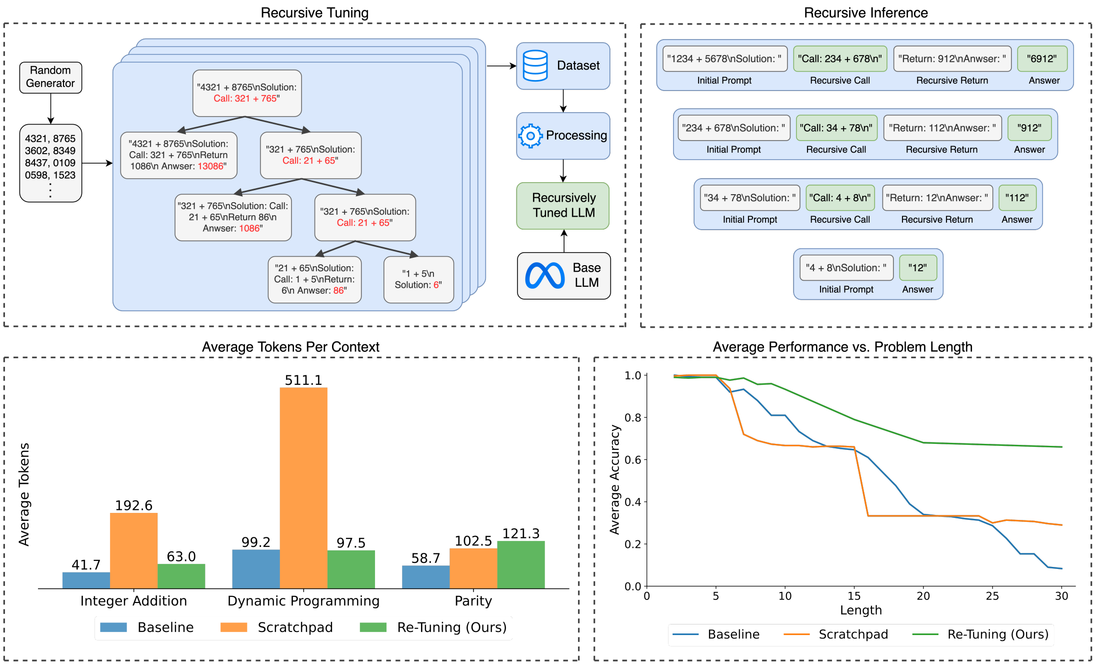


Figure 1: Summary of our approach and results. Upper Left: Our recursive tuning pipeline generates and processes all the recursive subproblems for each randomly generated problem instance in order to train the base LLM. Upper Right: Our recursive inference pipeline allows the model to call itself on a subproblem of reduced size, which enables the subproblem to be solved in a new context, and return the answer back to the initial context. Lower Left: On most problems, Re-Tuning trains on significantly less tokens than the scratchpad method, saving considerable GPU memory. Bottom Right: On average, Re-Tuning outperforms the baseline and scratchpad methods across all tasks, especially as the problems grow in size and complexity.

solving each subproblem. Second, our tuning process is reminiscent of recent works that incorporate tool use in LLMs (Schick et al., 2023; Paranjape et al., 2023). Similar to how these models call a tool and resume generating final output based on the output of the tool, in Re-Tuning, the models call themselves on a subproblem and resume generating after receiving the solution to the subproblem.

We empirically evaluate the performance of Re-Tuning on three representative compositional tasks: integer addition (Zhou et al., 2023b), a dynamic programming problem (Dziri et al., 2023), and the parity problem (Anil et al., 2022; Zhou et al., 2023b). Our results show Re-Tuning improves the average performance of LLaMA 7B and LLaMA 13B on all tasks by 37.4% and 31.9% over baseline finetuning. Compared to scratchpad finetuning, our improvement is striking, with averaging 34.5% and 36.7% points improvements on LLaMA 7B and LLaMA 13B respectively. Importantly, we show Re-Tuning saves significant GPU memory compared to the scratchpad method when training. We hope our results foster future research on recursive learning of large foundation models.

2 Approach

We present Re-Tuning in this section. Re-Tuning recursively tunes LLMs to solve compositional tasks. Specifically, the method (1) recursively decreases the size of the problem, (2) solves the base case, and (3) passes the solutions up the recursion stack, solving subproblems of increasing complexity along the way.

First, Re-Tuning recursively generates prompts for subproblems of decreasing length or complexity. For example, when adding the two numbers 1,234 and 5,678, the model generates a prompt to add 234 + 678. Then this prompt is sent to a separate context where the model generates a new prompt to add 34 + 78, and finally this new prompt is sent to a separate context where the model again generates a new prompt to add 4 + 8.

Next, the base case is solved. Certainly, the base cases are trivial enough to be solved directly in the same context. For the integer addition problem, the base case is to add the two least-significant digits together, for example, 4 + 8.

Finally, the solutions are passed up a level in the recursive stack. They are appended after the prompt in the previous context in the recursive stack. Again, sticking with integer addition, it's helps to know the solution to 4 + 8 when tasked with solving 34 + 78. So the answer the model generates to 4 + 8 is appended to the context tasked with solving 34 + 78. This process of appending solutions continues until eventually, the solution to the first recursive call is passed to the initial context, which helps to solve the initial prompt.

In order to accomplish this, we finetune the model to (1) generate recursive subproblems, (2) solve base cases, and (3) to use the answers propagated up from these recursive calls in its computation. During generation, the model can designate some of its generated text to be a prompt by enclosing the text between '`Call: `' and '`\n`'. Once this happens, we stop generating in the current context and prompt the model with the enclosed text in a new context. In each new context we follow the exact same generation procedure, except for the base case where the model learns to directly output the answer to the base case rather than making another recursive call. When generation in the new context is complete, we take the final answer, which is separated by the text '`\nAnswer: `', and append that to the initial context which generated the prompt. Then we continue generation in the initial context. Basic pseudo-code for the generation procedure is in Figure 9.

In the addition example, the model only needs to generate one recursive call in each context. However, our method works more generally than this. Many recursive calls may be generated in a given context. For example, the dynamic programming problem we describe below requires multiple recursive calls in the initial context to solve the problem.

3 Experiments

We consider three tasks: integer addition, a dynamic programming problem, and the parity problem. For each task, we train 3 types of models: baseline, scratchpad, and Re-Tuning. The baseline models were trained to simply output the solution to the problem. The scratchpad models were trained to generate a scratchpad (Nye et al., 2021) containing intermediate reasoning steps before generating the final solution to the problem. The Re-Tuning models are as described above.

In our evaluation, we look at in-distribution and out-of-distribution (OOD) data. The in-distribution data are those with lengths that were seen in training and the OOD data are those with lengths longer than seen in training. For example, on addition the training data consists of numbers with lengths up to 15 digits. So evaluation examples with 1-15 digits are considered in-distribution and examples with 16 or more digits are considered OOD.

We finetune LLaMA 7B and 13B (Touvron et al., 2023a) using Low-Rank Adapters (Hu et al., 2022). See Appendix A.1 for additional details on the finetuning setup.

3.1 Experimental Setup

3.1.1 Tasks and Datasets

We consider 3 representative compositional problems: integer addition, a dynamic programming problem, and the parity problem. Here, we describe each problem in detail, as well as how the data was constructed. For more details on dataset generation see Appendix A.2

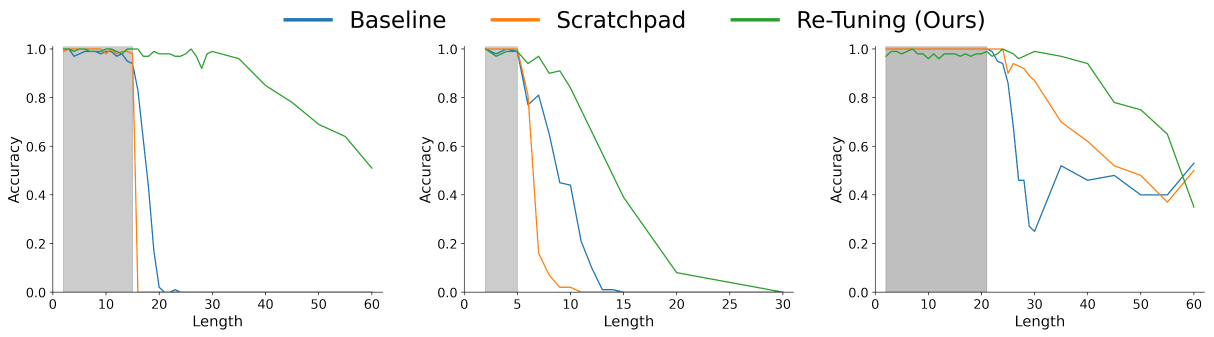
Integer Addition With this problem, we investigate the extent to which LLMs can add two integers. The input to the model is simply the prompt to add 2 numbers. For example, "45 + 97". Language models have some capability to perform addition without any finetuning, but it seemingly disappears as the numbers grow in size. Nye et al. (2021) used a scratchpad to teach language models addition, and more recently Liu and Low (2023) taught LLaMA 7B to add numbers up to 15 digits. In both cases, there is remarkable performance degradation when adding integers larger than those seen during finetuning. Zhou et al. (2023b) suggest that addition is particularly hard for LLMs since it requires precise indexing operations and the non-causal propagation of the carry term.

Following Liu and Low (2023), we generate training data summing randomly-generated integers up to 15 digits long. During evaluation, we sample 100 problems of lengths up to 60 digits.

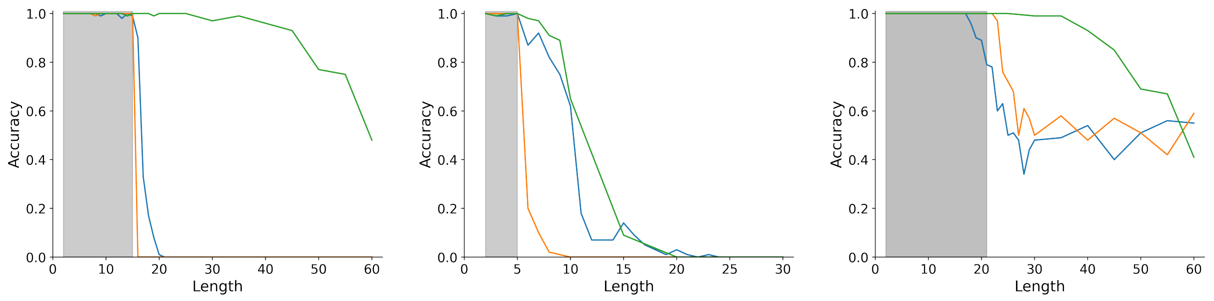
Dynamic Programming Problem We leverage the dynamic programming problem recently studied by Dziri et al. (2023):

> "Given a sequence of integers, find a subsequence with the highest sum, such that no two numbers in the subsequence are adjacent in the original sequence."

This problem can be broken down into two steps: first, recursively generate an array of sub-array sums, and then recursively identify which indices



(a) LLaMA 7B Integer Addition (b) LLaMA 7B Dynamic Programming (c) LLaMA 7B Parity



(d) LLaMA 13B Integer Addition (e) LLaMA 13B Dynamic Programming (f) LLaMA 13B Parity

Figure 2: Performance of LLaMA 7B (top) and LLaMA 13B (bottom) on Addition (left), Dynamic Programming (middle), and Parity (right). The in-distribution range is shaded in gray.

correspond to the highest sum. Re-Tuning generates prompts for each of these steps, which are then solved in recursive contexts.

For example, consider the sequence $[3, 2, -2, 5, 3]$. The subsequence with the highest sum with no adjacent numbers would be the one that has the first 3 and the 5. For this, the model outputs a list of 1s and 2s with 1s corresponding to numbers chosen and 2s corresponding to numbers not chosen. In the case, the model should output $[1, 2, 2, 1, 2]$.

Following Dziri et al. (2023), we exhaustively generate all permutations of arrays up to length 5 for training, where each element is restricted to $[-5, 5]$. Evaluation is done on arrays up to length 30, still with each integer element restricted to $[-5, 5]$.

Parity Problem The parity problem is to determine if there is an even or odd number of 1's in an input array consisting of 0's and 1's. An example input is $[0, 1, 0, 0]$, for which the output should be 1 since the array contains an odd number of 1's. In cases where there is an even number of 1's the output should be 0. This problem was previously studied by Anil et al. (2022) and Zhou et al. (2023b), who explored length generalization on this problem. This problem can be solved by traversing the input array and adding all the numbers mod 2, which is the method we finetune our model to use.

We generate binary arrays up to length 21 for training. For evaluation, we sample 100 arrays from various lengths up to 80.

3.2 Main Results

Here we share our main results on LLaMA 7B and LLaMA 13B, across all three tasks. Results are shown in Figure 2, and discussed in detail in the proceeding sections. Across all problems and model sizes, the Re-Tuning method outperforms the baseline and scratchpad methods, with the clearest difference being on the addition. The baseline method tends to exhibit better OOD generalization compared to scratchpad among the different problems.

3.2.1 Integer Addition

In Figure 2, we can see the Re-Tuning method outperforms baseline and scratchpad methods. The scratchpad method performs the worst, achieving 0% accuracy on every problem longer than those seen during training. The baseline method has nonzero OOD accuracy for problems upto length 20, but still quickly falls to 0% accuracy on longer problems. In contrast, the Re-Tuning method maintains near-perfect accuracy in regimes where the baseline and scratchpad models have 0% accuracy, and maintains around 50% accuracy on adding up to 60 digit numbers. The model from (Liu and Low, 2023), which is also trained on addition up to 15 digits, has similar OOD performance to our baseline model, and falls to 0% accuracy when adding 21 digit numbers.

3.2.2 Dynamic Programming Problem

Again, the Re-Tuning outperforms both baseline and scratchpad approaches, though the gap between Re-Tuning and baseline is narrower for LLaMA 13B than it is for LLaMA 7B. Dziri et al. (2023) also evaluate a finetuned GPT3 with and without scratchpad, which are also trained on examples up to length 5. Both of their models reach 0% accuracy when evaluating on length 10, which is similar to our baseline and scratchpad results.

3.2.3 Parity Problem

Re-Tuning performs as well or better on inputs up to (but not including) size 60. Since the baseline and scratchpad methods only every learn to output either 0 or 1, it can maintain an accuracy of around 50%, or random chance. The Re-Tuning models learn to output text other than just 0 or 1, such as recursive calls, and so they can fall below 50% accuracy on OOD data. Up to length 60, Re-Tuning is still the dominant performer on the problem.

4 Analysis and Further Discussion

In this section, we conduct additional experiments in order to better understand the different mechanisms behind Re-Tuning.

4.1 Ablation Study

For all tasks, as the problem size grows, so does the number of unique possible problems. For example, there are more combinations of 10-digit addition problems then there are 2-digit addition problems. If we randomly sample problems from the space of all possible problems up to some length, then the distribution of problems will be skewed towards longer problem instances. Due to Re-Tuning's recursive design, it's important that an appropriate number of small problems are included in the training data. Our resampling methodology is described in Appendix A.2.1

To better understand the impact of resampling short problem instances for finetuning, we finetune LLaMA-7b using integer addition data with and without resampling. Both models saw the same number of training examples. We collect results on 100 problems of length 5, 20, 35, and 50. Results are shown in Figure 1.

	5	20	35	50
w/ Resampling	1.0	0.98	0.96	0.69
w/o Resampling	1.0	0.97	0.73	0.0

Table 1: Ablation over resampling approach during Re-Tuning finetuning.

While both trained models perform well on problems up to length 20, the superiority of the resampling approach becomes clear on longer problems. At a problem length of 35, the resampling model achieved 96% accuracy, while the accuracy of the model trained without resampling is only 73%. At a problem length of 50, the model trained without resampling fails to correctly solve a single problem instance. However, the model trained with resampling maintains an accuracy of 69%, which suggests that resampling shorter problem instances is important for the success of Re-Tuning.

4.2 Case Study

The scratchpad models often make errors with indexing operations. For example, on the dynamic programming problem, the largest training array size is 5 elements. In Figure 3 we see that the model generated scratchpad starts the index of the array of sub-array sums (dp array) at 5 given an input array of length 7. The dp array should start at the last element of a 7 element array, with index 6 instead of 5. Once the model makes this indexing error on the scratchpad it is unable to recover. In other examples not displayed here, the scratchpad method correctly generates the text for "dp[6]" but fails to populate the subsequent expressions with the correct values from the input array. It is unable to reliably extract the elements with the correct index from the input array. In contrast, the Re-Tuning method is shown in Figure 3. Only the initial context is shown to save space. The Re-Tuning model is able to generate prompts with no indexing difficulty in this case.

4.3 Error Analysis

In order to better understand the types of errors made by Re-Tuning models, we perform extensive error analysis on each task. For each problem, we sample 20 problem instances from a selection of problem lengths, and categorize them into the following types:

Prompt

```
Question: Let's solve input = [1, 0, -2, 1, 0, 4, 2].
```

Scratchpad Model Response

```
Scratchpad: dp[5] = max(input[5], 0) = max(4, 0) = 4
dp[4] = max(input[4], input[5], 0) = max(1, 4, 0) = 4
dp[3] = max(dp[4], input[3] + dp[5], 0) = max(4, -2 + 4, 0) = 4 ... [rest of output truncated]
```

Re-Tuning Response

```
Call: Create dp array [1, 0, -2, 1, 0, 4, 2]\nReturn: [6, 5, 5, 5, 4, 4, 2]
Answer: Call: Create chosen indices array: sum array [6, 5, 5, 5, 4, 4, 2], item array [1, 0...
Return: [1, 2, 2, 1, 2, 1, 2]\nAnswer: [1, 2, 2, 1, 2, 1, 2]
```

Figure 3: Dynamic Programming Example: Prompt and Responses (Truncated)

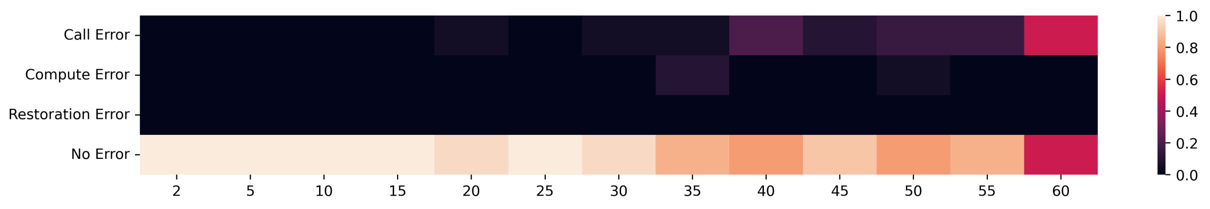
- Call Error: At some point in the call stack, an incorrect recursive call is made. As a result, the input prompt to the new context is incorrect.
- Compute Error: This error can manifest either because the base case is incorrectly solved, or at some point in the call stack the models returns the wrong solution to a subproblem even though the correct answer to its recursive call was received. As a result, the answer returned by the current context to the earlier context will be incorrect.
- Restoration Error: A restoration error occurs if at some point in the call stack, a call error or compute error is made, yet later recovered such that the final answer to the prompt in the initial context is correct. Importantly, since the model is able to recover, instances of restoration errors are classified as correct.
- No Error: In order for a problem instance to be free of errors, each recursive call must be correct, the base case must be solved correctly, and the correct answers are propagated up the call stack, leading to the correct final answer.

Figure 4 displays the error classifications for each problem. Importantly, across all tasks, the prevalence of errors increases with the size of the problem.

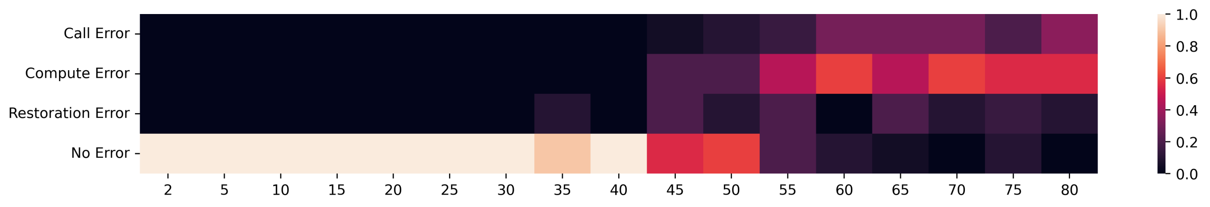
On the integer addition task, very few call errors and compute errors occur before a problem size of 30. The vast majority of those errors are call errors, suggesting that the model has a difficult time constructing the subproblem. Importantly, this aligns with Zhou et al. (2023b), which suggest that simply copying long strings of text with repeating characters is a difficult task for transformer-based models to perform. Furthermore, the lack of restoration errors suggest that once a call error is made, the model has a very hard time recovering.

For the parity problem, we again see very few errors of any type before a problem size of 40. In contrast with the addition problem, the majority of errors made on the parity problem are compute errors, not call errors. This is rather unintuitive, as the addition operation is seemingly much harder than the possible polarity flip in the polarity problem. Interestingly, we see more cases of restoration errors in the parity problem, which suggests that it may be easier for the model to recover from call or compute errors on this task compared to integer addition.

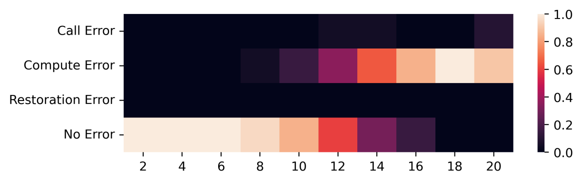
For the dynamic programming problem, we perform error analysis on each subproblem separately. The first subproblem deals with constructing the an array of sub-array sums, while the section subproblem identifies which indices correspond to the maximum sum. While compute errors occur most frequently on the first subproblem, call and restoration errors occur more frequently on the second subproblem. This checks out, as the first subproblem requires a rather simple call, but involves a more complex step to compute the answer, whereas



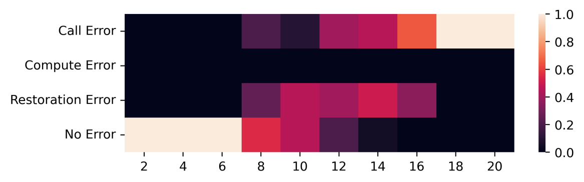
(a) Errors on Integer Addition



(b) Errors on Parity Problem



(c) Errors on Dynamic Programming (subproblem 1)



(d) Errors on Dynamic Programming (subproblem 2)

Figure 4: Error classifications for each problem across samples of size 20 for a selection of problem lengths.

the second subproblem contains a more involved recursive call, but an easier computation to produce the index array. Furthermore, the prevalence of restoration errors on subproblem 2 suggest that these call errors are easier to recover from than the computer errors made in subproblem 1.

4.4 Sample Efficiency

We also run experiments to see the performance of Re-Tuning in the low-data regime on addition. The results are in Figure 5. For each experiment we construct training data consisting of n examples for each problem length, where n is 10, 25, and 50 and the numbers contain between 1 and 15 digits. For example, when n is 10, the model will see 10 examples of adding 2 1-digit numbers, 10 examples of adding 2 2-digit numbers, etc. So it sees 150 examples in total when n is 10. We train baseline, scratchpad, and Re-Tuning models on these examples for 5 epochs each. We do not use any resampling as in the other experiments.

With seeing only 50 examples per problem length, the Re-Tuning model has performance close to the Re-Tuning model in the full-data regime above. In contrast, the baseline model has much worse performance than the baseline model in the full-data regime. With only seeing 10 examples per problem length in training, the Re-Tuning model is comparable to the baseline model that sees 50 examples per problem length in training, a 5x efficiency increase.

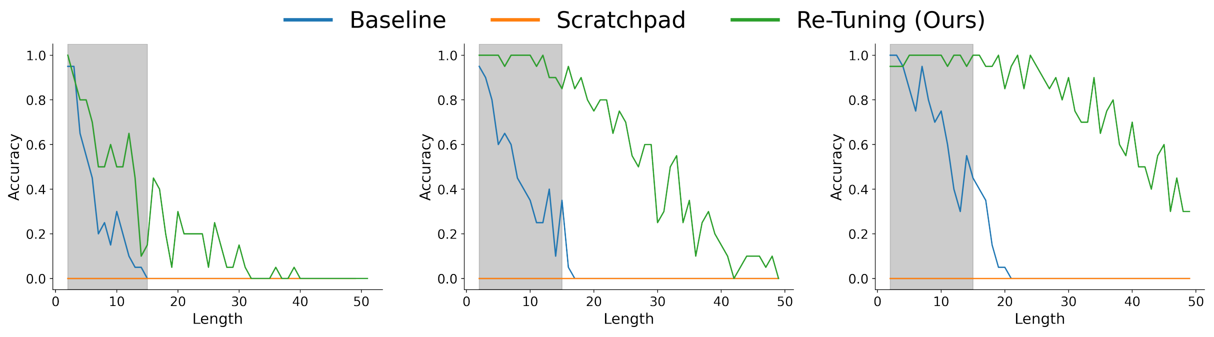
4.5 Prompt Sensitivity

Here, we explore the sensitivity of Re-Tuning models to various prompts during inference. Specifically, we take our best LLaMA-7b trained on the integer addition problem with Re-Tuning using the prompt "`{num_1} + {num_2}\nSolution: `", and we evaluate the model using several alternative prompt formats for inference. Results are shown in Table 2 on 100 problems of length 5, 20, 35, and 50.

	5	20	35	50
"`{num_1} + {num_2}\nSolution: `"	1.0	0.98	0.96	0.69
"`{num_1} + {num_2}\nAnswer: `"	1.0	1.0	0.86	0.65
"`{num_1} + {num_2}\n `"	1.0	0.98	0.88	0.67
"`{num_1} - {num_2}\nSolution: `"	0.28	0.26	0.18	0.07

Table 2: Prompt sensitivity analysis on LLaMA 7B with Re-Tuning on the integer addition problem.

The results suggest that during inference, Re-Tuning is not very sensitive to minor deviations in the prompt. The first prompt in Table 2 is the prompt used during training. The second and third prompts include small prompt deviations, and result in slightly worse performance on longer problems. Specifically, the 2nd prompt uses "`Answer: `"



(a) 10 Examples per Length (b) 25 Examples per Length (c) 50 Examples per Length

Figure 5: Results on Addition with limited training data of 10 examples per length (left), 25 examples per length (middle), and 50 examples per length (right). Scratchpad results are omitted since they were constant at 0% accuracy.

in an attempt to have the model skip the recursive call, but it appears that Re-Tuning is robust against such attacks. Re-Tuning however, is not robust against the fourth prompt, which adversarially prompts for subtraction rather than addition, resulting in significantly worse performance.

5 Related Work

Several works have explored the ability of LLMs for length generalization on compositional problems. Dziri et al. (2023) suggests that LLMs solve compositional tasks via "linearized subgraph matching" and thus fail to learn the underlying algorithm necessary to solve more complex problem instances. Anil et al. (2022) showed that finetuning on a combination of in-context learning and scratchpad prompting could enable better performance. Similarly Re-Tuning involves finetuning pretrained LLMs to make recursive calls in order to improve performance on composition tasks. Other works have studied length generalization on small, purpose-built transformer models. Lee et al. (2023) and Zhou et al. (2023b) showed that training small transformers models from scratch on scratchpad data could enable better length generalization.

Various papers have explored the idea of LLMs prompting themselves or other LLMs, although no papers to our knowledge that explicitly finetune a language model to do so. Zhou et al. (2023a) prompts a language model to break a problem down into simpler steps and then prompts itself to solve each step individually in a sequential, non-recursive manner. Similar methods have been proposed as way to improve the logical consistency of the generated repsonses (Jung et al., 2022; Imani et al., 2023). Weston and Sukhbaatar (2023) use a language model to generate prompts by extracting relevant information from the context. Similarly, Re-Tuning generate a prompt that includes the relevant information for solving a simpler subproblem.

A recent topic of interest has been teaching language models to use tools (Hsieh et al., 2023; Parisi et al., 2022; Schick et al., 2023; Qin et al., 2023; Paranjape et al., 2023; Mialon et al., 2023), which often involve stopping the generation and waiting for the tool output before containing with generation. Re-Tuning can be interpreted as teaching LLMs to use themselves as a tool.

Several papers have investigated the ability of language models on arithmetic tasks (Lee et al., 2023; Liu and Low, 2023; Nye et al., 2021; Nogueira et al., 2021; Kim et al., 2021; Duan and Shi, 2023). In many cases, it was noticed that performance was significantly worse on problems longer than those saw during finetuning.

6 Conclusion

We study the problem of solving compositional tasks with large language models. We propose a new tuning paradigm that decomposes the original compositional problem into smaller and smaller instances of the same problem, solves each, and combines the results to produce the final answer. To the best of our knowledge, our method is the first to utilize the recursive property of the compositional tasks. Experimental results on three standard compositional tasks have demonstrated the effectiveness of our method. Our method not only significantly outperforms standard finetuning and state of the art methods, but also is more memory efficient during training. We hope our method can be used in more tasks where recursive computation is inherent and computation resources are limited.

Limitations

Re-Tuning has shown better accuracy and better sample efficiency than standard methods. However, it does have some disadvantages. Re-Tuning takes longer to generate responses than standard prompting because it generates prompts in addition to generating the final answer. The inference procedure for Re-Tuning is also more complex than standard inference since we need to extract text from contexts, check if there is a generated prompt in the text, and recursively generate using the generated prompts.

Additionally, in the experiments we present, we needed to craft training data with added reasoning, similar to that used in scratchpad training data. This last point is not a disadvantage of the Re-Tuning method, but only of our current experiments. In principle, we can apply Re-Tuning to more general problems by training it to make recursive calls in similar ways to teaching language models to use tools. For example we could use the method of Schick et al. (2023) to teach the model to make recursive calls for general problems.

References

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

Rohan Anil, Andrew M. Dai, Orhan Firat, Melvin Johnson, Dmitry Lepikhin, Alexandre Passos, Siamak Shakeri, Emanuel Taropa, Paige Bailey, Zhifeng Chen, Eric Chu, Jonathan H. Clark, Laurent El Shafey, Yanping Huang, Kathy Meier-Hellstern, Gaurav Mishra, Erica Moreira, Mark Omernick, Kevin Robinson, Sebastian Ruder, Yi Tay, Kefan Xiao, Yuanzhong Xu, Yujing Zhang, Gustavo Hernandez Abrego, Junwhan Ahn, Jacob Austin, Paul Barham, Jan Botha, James Bradbury, Siddhartha Brahma, Kevin Brooks, Michele Catasta, Yong Cheng, Colin Cherry, Christopher A. Choquette-Choo, Aakanksha Chowdhery, Clément Crepy, Shachi Dave, Mostafa Dehghani, Sunipa Dev, Jacob Devlin, Mark Díaz, Nan Du, Ethan Dyer, Vlad Feinberg, Fangxiaoyu Feng, Vlad Fienber, Markus Freitag, Xavier Garcia, Sebastian Gehrmann, Lucas Gonzalez, Guy Gur-Ari, Steven Hand, Hadi Hashemi, Le Hou, Joshua Howland, Andrea Hu, Jeffrey Hui, Jeremy Hurwitz, Michael Isard, Abe Ittycheriah, Matthew Jagielski, Wenhao Jia, Kathleen Kenealy, Maxim Krikun, Sneha Kudugunta, Chang Lan, Katherine Lee, Benjamin Lee, Eric Li, Music Li, Wei Li, YaGuang Li, Jian Li, Hyeontaek Lim, Hanzhao Lin, Zhongtao Liu, Frederick Liu, Marcello Maggioni, Aroma Mahendru, Joshua Maynez, Vedant Misra, Maysam Moussalem, Zachary Nado, John Nham, Eric Ni, Andrew Nystrom, Alicia Parrish, Marie Pellat, Martin Polacek, Alex Polozov, Reiner Pope, Siyuan Qiao, Emily Reif, Bryan Richter, Parker Riley, Alex Castro Ros, Aurko Roy, Brennan Saeta, Rajkumar Samuel, Renee Shelby, Ambrose Slone, Daniel Smilkov, David R. So, Daniel Sohn, Simon Tokumine, Dasha Valter, Vijay Vasudevan, Kiran Vodrahalli, Xuezhi Wang, Pidong Wang, Zirui Wang, Tao Wang, John Wieting, Yuhuai Wu, Kelvin Xu, Yunhan Xu, Linting Xue, Pengcheng Yin, Jiahui Yu, Qiao Zhang, Steven Zheng, Ce Zheng, Weikang Zhou, Denny Zhou, Slav Petrov, and Yonghui Wu. 2023. PaLM 2 Technical Report. *arXiv*.

Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. Language Models are Few-Shot Learners. In *NeurIPS*.

Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, Parker Schuh, Kensen Shi, Sasha Tsvyashchenko, Joshua Maynez, Abhishek Rao, Parker Barnes, Yi Tay, Noam Shazeer, Vinodkumar Prabhakaran, Emily Reif, Nan Du, Ben Hutchinson, Reiner Pope, James Bradbury, Jacob Austin, Michael Isard, Guy Gur-Ari, Pengcheng Yin, Toju Duke, Anselm Levskaya, Sanjay Ghemawat, Sunipa Dev, Henryk Michalewski, Xavier Garcia, Vedant Misra, Kevin Robinson, Liam Fedus, Denny Zhou, Daphne Ippolito, David Luan, Hyeontaek Lim, Barret Zoph, Alexander Spiridonov, Ryan Sepassi, David Dohan, Shivani Agrawal, Mark Omernick, Andrew M. Dai, Thanumalayan Sankaranarayana Pillai, Marie Pellat, Aitor Lewkowycz, Erica Moreira, Rewon Child, Oleksandr Polozov, Katherine Lee, Zongwei Zhou, Xuezhi Wang, Brennan Saeta, Mark Diaz, Orhan Firat, Michele Catasta, Jason Wei, Kathy Meier-Hellstern, Douglas Eck, Jeff Dean, Slav Petrov, and Noah Fiedel. 2022. PaLM: Scaling Language Modeling with Pathways. *arXiv*.

Shaoxiong Duan and Yining Shi. 2023. From interpolation to extrapolation: Complete length generalization for arithmetic transformers.

Nouha Dziri, Ximing Lu, Melanie Sclar, Xiang Lorraine Li, Liwei Jiang, Bill Yuchen Lin, Peter West, Chandra Bhagavatula, Ronan Le Bras, Jena D. Hwang, Soumya Sanyal, Sean Welleck, Xiang Ren, Allyson Ettinger, Zaid Harchaoui, and Yejin Choi. 2023. Faith and fate: Limits of transformers on compositionality.

Cheng-Yu Hsieh, Si-An Chen, Chun-Liang Li, Yasuhisa Fujii, Alexander Ratner, Chen-Yu Lee, Ranjay Krishna, and Tomas Pfister. 2023. Tool documentation enables zero-shot tool-usage with large language models.

Edward J Hu, yelong shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. 2022. LoRA: Low-rank adaptation of large language models. In *International Conference on Learning Representations*.

Shima Imani, Liang Du, and Harsh Shrivastava. 2023. Mathprompter: Mathematical reasoning using large language models.

Jaehun Jung, Lianhui Qin, Sean Welleck, Faeze Brahman, Chandra Bhagavatula, Ronan Le Bras, and Yejin Choi. 2022. Maieutic prompting: Logically consistent reasoning with recursive explanations.

Jeonghwan Kim, Giwon Hong, Kyung-min Kim, Junmo Kang, and Sung-Hyon Myaeng. 2021. Have you seen that number? investigating extrapolation in question answering models. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pages 7031–7037, Online and Punta Cana, Dominican Republic. Association for Computational Linguistics.

Nayoung Lee, Kartik Sreenivasan, Jason D. Lee, Kangwook Lee, and Dimitris Papailiopoulos. 2023. Teaching arithmetic to small transformers.

Tiedong Liu and Bryan Kian Hsiang Low. 2023. Goat: Fine-tuned llama outperforms gpt-4 on arithmetic tasks.

Grégoire Mialon, Roberto Dessi, Maria Lomeli, Christoforos Nalmpantis, Ramakanth Pasunuru, Roberta Raileanu, Baptiste Roziere, Timo Schick, Jane Dwivedi-Yu, Asli Celikyilmaz, Edouard Grave, Yann LeCun, and Thomas Scialom. 2023. Augmented language models: a survey. *Transactions on Machine Learning Research*. Survey Certification.

Rodrigo Nogueira, Zhiying Jiang, and Jimmy Lin. 2021. Investigating the limitations of transformers with simple arithmetic tasks.

Maxwell Nye, Anders Johan Andreassen, Guy Gur-Ari, Henryk Michalewski, Jacob Austin, David Bieber, David Dohan, Aitor Lewkowycz, Maarten Bosma, David Luan, Charles Sutton, and Augustus Odena. 2021. Show your work: Scratchpads for intermediate computation with language models.

OpenAI. 2023. GPT-4 Technical Report. *arXiv*.

Bhargavi Paranjape, Scott Lundberg, Sameer Singh, Hannaneh Hajishirzi, Luke Zettlemoyer, and Marco Tulio Ribeiro. 2023. Art: Automatic multi-step reasoning and tool-use for large language models.

Aaron Parisi, Yao Zhao, and Noah Fiedel. 2022. Talm: Tool augmented language models.

Yujia Qin, Shihao Liang, Yining Ye, Kunlun Zhu, Lan Yan, Yaxi Lu, Yankai Lin, Xin Cong, Xiangru Tang, Bill Qian, Sihan Zhao, Runchu Tian, Ruobing Xie, Jie Zhou, Mark Gerstein, Dahai Li, Zhiyuan Liu, and Maosong Sun. 2023. Toolllm: Facilitating large language models to master 16000+ real-world apis.

Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. 2023. Toolformer: Language models can teach themselves to use tools.

Ross Taylor, Marcin Kardas, Guillem Cucurull, Thomas Scialom, Anthony Hartshorn, Elvis Saravia, Andrew Poulton, Viktor Kerkez, and Robert Stojnic. 2022. Galactica: A large language model for science.

Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, Aurelien Rodriguez, Armand Joulin, Edouard Grave, and Guillaume Lample. 2023a. Llama: Open and efficient foundation language models.

Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, Dan Bikel, Lukas Blecher, Cristian Canton Ferrer, Moya Chen, Guillem Cucurull, David Esiobu, Jude Fernandes, Jeremy Fu, Wenyin Fu, Brian Fuller, Cynthia Gao, Vedanuj Goswami, Naman Goyal, Anthony Hartshorn, Saghar Hosseini, Rui Hou, Hakan Inan, Marcin Kardas, Viktor Kerkez, Madian Khabsa, Isabel Kloumann, Artem Korenev, Punit Singh Koura, Marie-Anne Lachaux, Thibaut Lavril, Jenya Lee, Diana Liskovich, Yinghai Lu, Yuning Mao, Xavier Martinet, Todor Mihaylov, Pushkar Mishra, Igor Molybog, Yixin Nie, Andrew Poulton, Jeremy Reizenstein, Rashi Rungta, Kalyan Saladi, Alan Schelten, Ruan Silva, Eric Michael Smith, Ranjan Subramanian, Xiaoqing Ellen Tan, Binh Tang, Ross Taylor, Adina Williams, Jian Xiang Kuan, Puxin Xu, Zheng Yan, Iliyan Zarov, Yuchen Zhang, Angela Fan, Melanie Kambadur, Sharan Narang, Aurelien Rodriguez, Robert Stojnic, Sergey Edunov, and Thomas Scialom. 2023b. Llama 2: Open Foundation and Fine-Tuned Chat Models. *arXiv*.

Jason Weston and Sainbayar Sukhbaatar. 2023. System 2 attention (is something you might need too).

Denny Zhou, Nathanael Schärli, Le Hou, Jason Wei, Nathan Scales, Xuezhi Wang, Dale Schuurmans, Claire Cui, Olivier Bousquet, Quoc V Le, and Ed H. Chi. 2023a. Least-to-most prompting enables complex reasoning in large language models. In *The Eleventh International Conference on Learning Representations*.

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

A Appendix

A.1 finetuning Details

Finetuning (and evaluation) were done on NVIDIA H100, A100, and RTX A6000 GPUs, depending on availability and the compute requirements of the job. Rather than finetune the full model, we finetune using low-rank adapters (Hu et al., 2022). Hyperparameters for finetuning jobs are in Table 3. These hyperparameters apply to finetuning all models across all tasks, with two notable exceptions: (1) when training parity baselines, we used a slightly higher learning rate of 5e-4 for better stability, and (2) the scratchpad finetuning job for the dynamic programming problem on LLaMA 13B used a batch size of 64, along with 64 gradient accumulation steps, so that the training job could be done on a single A100 GPU.

Parameter	**Value**
Learning Rate	2×10^{-4}
LR Schedule	Constant
Optimizer	AdamW
Batch Size	128
Gradient accumulation steps	32
Lora_r	64
Lora_alpha	64
Lora_dropout	0.05

Table 3: Hyperparameters used for finetuning.

During training, checkpoints are saved every 7936 training examples. We evaluate a handful of these checkpoints on a small validation set containing 5 examples from a handful of problem lengths (both in-distribution and out-of-distribution) in order choose the best model for full evaluation. We compute training loss (using cross entropy loss) only on the parts of the problem that the model will generate at inference time (c.f. grey vs. green highlighted text in the upper right of Figure 1).

A.2 Data Details

A.2.1 Resampling methodology

In general, we upsample examples with smaller lengths and downsample those with larger lengths in our training data. There are 2 reasons for this. One is that the examples with larger lengths are more numerous than examples with smaller lengths (there are many more examples of adding 2 15-digit numbers than there are adding 2 1-digit numbers). The other reason is specific to Re-Tuning. Since Re-Tuning generates calls to all examples except the base case, it has trouble learning the what to do in the base case if there are not enough examples. Without resampling, the base cases for each problem would be far less than 1% of the training data. We do not follow any specific methodology for resampling. We simply try to bring the training data distribution closer to uniform than it would be without resampling. See Figure 6 for a comparison of training data on the dynamic programming problem before and after resampling. Note the y-axis scales on each plot. Without resampling, examples of lengths 1 and 2 make up far too little of the data distribution for the model to learn them. We apply similar resampling in the other problem settings. See Figure 7 for a comparison of training data on the addition problem before and after resampling. See Figure 8 for a comparison of training data on the parity problem before and after resampling.

A.3 Psuedo-code for Re-Tuning generation

Psuedo-code for the recursive generation using in Re-Tuning is shown in Figure 9.

A.4 Problem Examples

A.4.1 Re-Tuning Addition

The original addition data from Liu and Low (2023) consists of numbers of up to 15 digits. For this we have the model add each digit individually. The model outputs the current sum and the current carry separately. For adding 637 and 123, the finetuning data is

```
'637 + 123\nSolution:
Call: 37 + 23\n
Return: Carry 0, Output 60\n
Answer: Carry 0, Output 760'

'37 + 23\nSolution:
Call: 7 + 3\n
Return: Carry 1, Output 0\n
Answer: Carry 0, Output 60'

'7 + 3\nSolution:
Carry 1, Output 0'
```

To compute the accuracy at generation, we extract the carry and output from the initial context. If the carry is 1, we prepend it to the output, else we leave the output as is. Then we cast the output sum to an integer and compare with the true sum to get the accuracy.

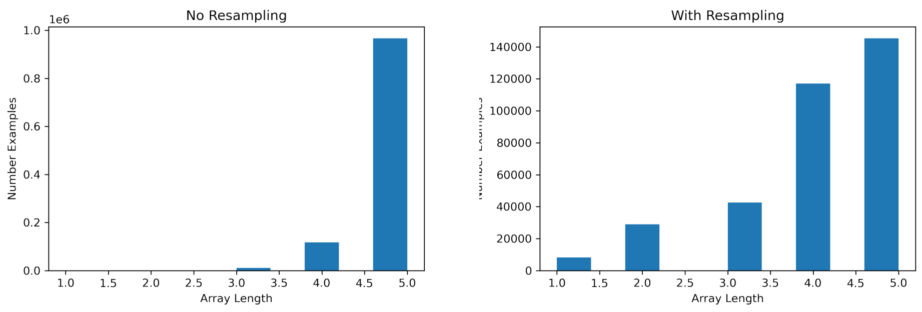


Figure 6: Dynamic Programming Training Data Resampling Comparison

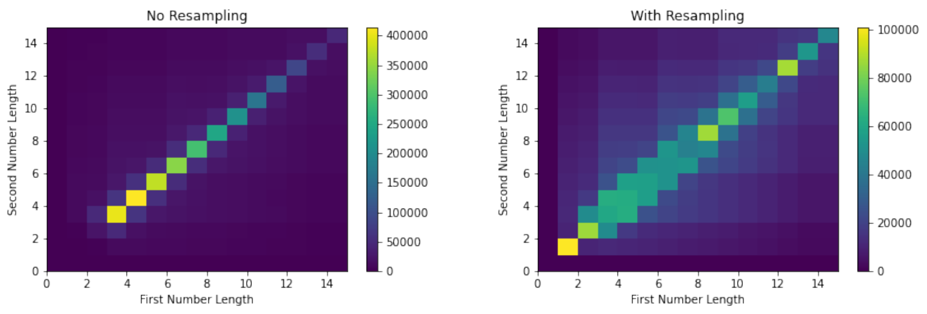


Figure 7: Addition Training Data Resampling Comparison

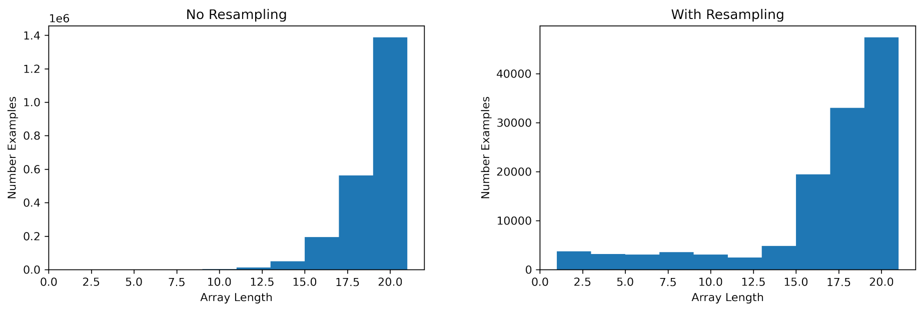


Figure 8: Parity Training Data Resampling Comparison

A.4.2 Re-Tuning Dynamic Programming

The initial data for this task consists of arrays of up to 5 numbers and each number is an integer from -5 to 5.

The finetuning data for the dynamic programming problem consists of 3 types of data: initial context, dp array contexts, and indices array contexts. In the initial context, the model creates prompts to create the dp array and the indices array (the indices array is the final output). The dp array contexts solve the subproblem to create an array with the sub-array sums and the indices array contexts solve the subproblem to create the final solution given the dp array. Examples of each of these training examples are shown below:

Listing 1: Initial Context Examples

```
'Compute the maximum sum of
nonadjacent subsequences of
[1, −1, 3]\nSolution:
Call: Create dp array [1, −1,
3]\nReturn:
[4, 3, 3]\nAnswer:
Call: Create chosen indices array:
sum array [4, 3, 3],
item array [1, −1, 3],
can use item True\n'

'Compute the maximum sum of
nonadjacent subsequences of
[3, 2]\nSolution:
Call: Create dp array [3, 2]
\nReturn: [3, 2]\nAnswer:
Call:
```

```
def recursive_generate(model, prompt):
    context = model.generate(prompt)

    # loop until all calls executed
    while exists_unexecuted_call(context):
        call = get_latest_call(context) # get latest call
        if correct_format(call):
            retrn = recursive_generate(model, call)
        else: Raise Exception

        call_answer = extract_answer(retrn) # get final answer from context
        new_prompt = context + call_answer # append answer from recursive context
        context = model.generate(new_prompt) # continue generation

    return context
```

Figure 9: Pseudo-code for Re-Tuning generation.

```
Create chosen indices array:
sum array [3, 2],
item array [3, 2],
can use item True\n'
```

Listing 2: DP Array Context Examples

```
'Create dp array [1, -1, 3]\n
Solution: Call with array
minus first element.
Call: Create dp array [-1, 3]\n
Return: [3, 3]\n
Answer: Append max(return[0],
array[0] + return[1])
to return.\n
Answer: [4, 3, 3]'

'Create dp array [-1, 3]\n
Solution: Call with array
minus first element.
Call: Create dp array [3]\n
Return: [3]\nAnswer: Append
max(return[0],array[0] +
return[1]) to return.\n
Answer: [3, 3]'

'Create dp array [3]\n
Solution: Return [0] if
negative else element.\n
Answer: [3]'
```

Listing 3: Indices Array Context Examples

```
'Create chosen indices array:
sum array [4, 3, 3], item array
[1, -1, 3], can use item True\n
Solution: If there is only 1
item, return 1 if we should
use it else 2. If we should use
the first item to get the sum,
call False else True.
Call: Create chosen
indices array: sum array
[3, 3], item array [-1, 3],
can use item False
\nReturn [2, 1]\nAnswer:
Append 1 if False else 2.\n
Answer: [1, 2, 1]'

'Create chosen indices array:
sum array [3, 3], item array
[-1, 3], can use item False\n
Solution: If there is only 1
item, return 1 if we should
use it else 2. If we should
use the first item to get the
sum, call False else True. Call:
Create chosen indices array: sum
array [3], item array [3],
can use item True\n
Return [1]\nAnswer:
Append 1 if False else 2.\n
Answer: [2, 1]'

'Create chosen indices array:
sum array [3], item array [3],
can use item True\nSolution:
If there is only 1 item, return
1 if we should use it else 2.\n
Answer: [1]'
```

To compute the accuracy for the dynamic programming problem, we parse out the index array in the initial context and compare to the string of the true solution array.

A.4.3 Re-Tuning Parity

For the parity problem the model computes the parity of the binary array in the prompt given the parity of the sub-list of the last n-1 items. So the finetuning examples for the parity of `[1, 0, 0, 1]` are:

```
'What is the parity of
[1, 0, 0, 1]?\n
Solution: Call:
What is the parity of
[0, 0, 1]?\n\n
Return: 1\nAnswer: 0'

'What is the parity of
[0, 0, 1]?\n
Solution:
Call: What is the parity of
[0, 1]?\n\n
Return: 1\nAnswer: 1'

'What is the parity of
[0, 1]?\n
Solution:
Call: What is the parity of
[1]?\n\n
Return: 1\nAnswer: 1'

'What is the parity of [1]?\n
Solution: 1'
```

To evaluate accuracy, when the model has finished we simply take the last (non-empty) element of the resulting string, and compare that to the true answer (as a string).

A.4.4 Scratchpad Addition

We have the model learn a scratchpad to add the digits in the 2 numbers individually. The data is very similar to the recursive addition data except there are no generated prompts and all the computations are in 1 context. To add 3116 + 5923, the data would be:

```
'3116 + 5923\nSolution:
Carry 0, Output 9\nCarry 0,
Output 39\nCarry 1,
Output 039\nCarry 0, Output 9039'
```

Similar to the recursive finetuning, we do not compute training loss on the prompt, which would be `'3116 + 5923\nSolution: '` in this example. We compute the accuracy in the same way as with the recursive addition.

A.4.5 Scratchpad Dynamic Programming

For this we use the same scratchpad as Dziri et al. (2023). So to solve the problem for the array [3, -2, 2], the training example is

```
'Question: Let's solve
input = {arr}.
Scratchpad: dp[2] =
max(input[2], 0) =
max(2, 0) = 2
dp[1] = max(input[1]
, input[2], 0)
= max(-2, 2, 0)
= 2
dp[0] =
max(dp[1], input[0]
+ dp[2], 0) =
max(2, 3 + 2, 0)
= 5

Finally, we reconstruct
the lexicographically
smallest subsequence that
fulfills the task objective
by selecting numbers as
follows. We store the
result on a list named "output".

Let can_use_next_item = True.
Since dp[0] == input[0] +
dp[2] (5 == 3 + 2) and
can_use_next_item == True,
we store output[0] = 1.
We update can_use_next_item
= False.
Since dp[1] != input[1] (2 != -2)
or can_use_next_item == False,
we store output[1] = 2. We
update can_use_next_item = True.
Since dp[2] == input[2] (2 == 2)
and can_use_next_item == True,
we store output[2] = 1.

Reconstructing all together,
output=[1, 2, 1].'
```

To compute the accuracy, we parse out the index array and compare this string to the string of the true answer. This is the same as with the recursive dynamic programming method.

A.4.6 Scratchpad Parity

The original data consists of arrays up to length 21 with entries that are either 0 or 1. We construct the finetuning data in the following way.

We have the model learn to sequentially sum the elements of the input array modulo 2, similar to the recursive method. However all the computations are done in 1 context with no recursive calls. The scratchpad we use is similar to that of Anil et al. (2022). For the input [1, 0, 1] the finetuning data is:

```
'What is the parity of
[1, 0, 1]?
\nSolution: Compute one
element at a time. 1 1 0'
```

The last number in the generation is the final answer. We compute the accuracy exactly the same as in the recursive case.

A.4.7 Baseline Addition

The baseline addition data consists of prompts to add 2 numbers. The target is just the sum of the numbers.

```
'24 + 97\nAnswer: 121'
```

A.4.8 Baseline Dynamic Programming

The prompt for baseline dynamic programming data and the response are below:

```
'Given a sequence of integers,
find a subsequence with the
highest sum, such that no
two numbers in the subsequence
are adjacent in the original
sequence.\n\nOutput a list
with "1" for chosen numbers
and "2" for unchosen ones. If
multiple solutions exist, select
the lexicographically smallest.
Input = [1, -3, 2].\n
Answer: [1, 2, 1]'
```

A.4.9 Baseline Parity

The prompt for baseline parity and the response are below:

```
What is the parity of
[0, 1, 0]?\n
Answer: 1
```