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 solving smaller instances of the same problem. We propose a natural approach to solve compositional tasks recursively. Our method, Re-Tuning, tunes models to break down a problem into subproblems, solve those subproblems, and combine the results. We show that our method significantly improves 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.

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 1a. When calculating '1234 + 4567', we first break the problem into a smaller subproblem '234 + 567'. After obtaining the solution to this subproblem, the original problem is partially solved. Similarly, to solve

'234 + 567', we first sum '34 + 67'. This recursion is the fundamental operation to solve compositional tasks. However, no existing approach has explicitly captured the recursive nature of compositional tasks.
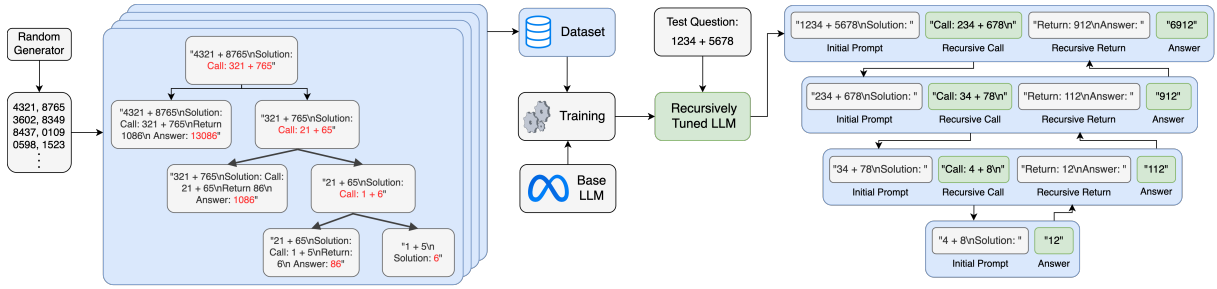
In this paper, we propose a recursion-based method for LLMs to better solve compositional tasks. More specifically, we adopt a top-down approach to solve problems recursively. We train LLMs to recursively call themselves on subproblems of reduced size, recognize and solve the base case directly, and combine the solutions up the associated call stack to obtain the solution to the original problem (Figure 1a). The above procedure is referred to 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 train models, which is computationally costly, Re-Tuning breaks down the problems into smaller and smaller subproblems. Each subproblem runs independently within its own context in the associated call stack. The solution to each subproblem is then propagated up the call stack to produce the final solution. Since each level of the call stack only includes the information necessary to solve the current subproblem, models can more easily attend to the relevant context, improving the accuracy of 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 output based on the output of the tool, with Re-Tuning, the models call themselves on a subproblem and resume generating
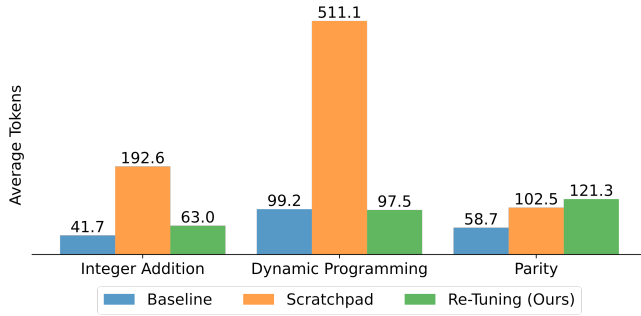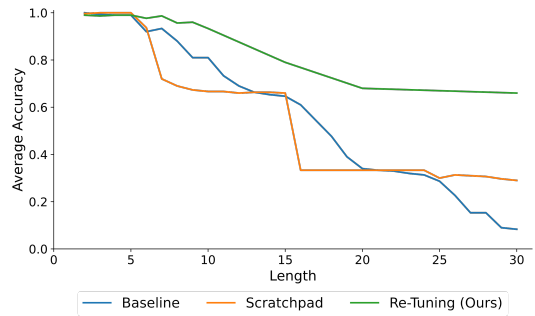

(a) Re-Tuning pipeline.

(b) Average tokens per context.

(c) Average performance vs. problem length.

Figure 1: Summary of our approach and results. Top: Our Re-Tuning pipeline generates and processes all the recursive subproblems for each randomly generated problem instance in order to train the base LLM. For a new question, our Re-Tuning 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 to the initial context. The top right shows the generation procedure to solve 1234+5678. Each separate context is indicated by a blue bubble. The arrows indicate copying of generated prompts or solutions. Bottom Left: On most problems, Re-Tuning trains on significantly fewer 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.

after receiving the subproblem's solution.

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 training. Compared to scratchpad training, our improvement is striking, with average improvements of 34.5% and 36.7% 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 involves (1) recursively decreasing the size of the problem, (2) solving the base case, and (3) passing the solutions up the recursion stack, solving subproblems of increasing complexity along the way.

First, with Re-Tuning, an LLM recursively calls itself on subproblems of decreasing length or complexity. For example, when adding 1234 + 5678, the LLM calls itself to add 234 + 678. This call is then sent to a new context in which the LLM calls itself to add 34 + 78, which is again sent to a new context where the LLM calls itself to add 4 + 8.

Next, the base case is solved. The base cases are easy 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 (e.g., adding 4 + 8).

Finally, the subproblem solutions are passed up the recursive call stack. Specifically, subproblem solutions are appended directly after the associated call in the context one level up the call stack.

2

```
function RecursiveGenerate(model, tokenizer, prompt)
    context ← Generate(model, tokenizer, prompt)
    while ContainsUnexecutedCall(context) do
        call ← ExtractCall(context)
        result ← RecursiveGenerate(model, tokenizer, call)
        context ← context + result
        context ← Generate(model, tokenizer, context)
    end while
    return context
end function
```

Algorithm 1: Psuedocode for the RecursiveGenerate method, a lightweight recursive wrapper around the standard generation function used with the baseline and scratchpad methods.

Again, sticking with integer addition, it helps to know the sum of 4 + 8 when tasked with adding 34 + 78. As such, the LLM-generated solution to 4 + 8 is appended to the context tasked with solving 34 + 78. This process of propagating subproblem solutions continues up the recursive call stack until the solution to the first recursive call is passed to the initial context, which helps to solve the initial problem.

To accomplish this, we train LLMs to (1) generate recursive subproblems, (2) solve base cases, and (3) use the answers propagated up from these recursive calls in their computation for the problem in the current context. We do so by randomly generating a set of seed data from which we programmatically construct training instances for all three types (see Figure 1a).

During generation, the model can designate some of its generated text to be a recursive call by enclosing the text between 'Call: ' and '\n'. Once a recursive call is made, we stop generating in the current context and prompt the model with the call 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 rather than making another recursive call. When generation in the new context is complete, we take the subproblem solution, which is separated by the text '\nAnswer: ', and append that to the context one level higher in the associated call stack. Then we continue generation in the initial context. The pseudo-code for the recursive generation procedure is in Figure 1.

In the integer 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 single 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.

During evaluation, we consider both in-distribution and out-of-distribution (OOD) data. The in-distribution data are those with problem lengths that were seen in training and the OOD data are those with problem lengths longer than seen in training. For example, on the integer addition task, the training data consists of numbers with lengths up to 15 digits. Evaluation examples with 1-15 digits are considered in-distribution and examples with 16 or more digits are considered OOD.

We train LLaMA 7B and 13B (Touvron et al., 2023a) using Low-Rank Adapters (Hu et al., 2022). See Appendix A.2 for additional details on the training setup. Additionally, we provide results on the smaller Galactica (Taylor et al., 2022) 125m and 1.3B parameter models, in Appendix B.

3.1 Experimental Setup

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. Additional details are provided in Appendix A.1 and examples are provided in Appendix C.

Integer addition This problem challenges LLMs to add two integers. The input to the model is simply a prompt to add 2 numbers. For example, '45 + 97'. Pretrained language models have some capability to perform addition without any training, 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 inte-

gers larger than those seen during training. Zhou et al. (2023b) suggests that addition is particularly hard for LLMs since it requires precise indexing operations and 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 focus on adding two numbers with the same number of digits and sample 100 problems per length up to 60 digits.

Dynamic programming   We borrow 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: (1) recursively generate an array of sub-array sums and (2) recursively identify which indices correspond to the highest sum. With Re-Tuning LLMs generate recursive calls for each of these steps, which are then solved in separate contexts. For example, consider the sequence [3, 2, -2, 5, 3]. The subsequence with the highest sum with no adjacent numbers would contain 3 (element 0) and 5 (element 3). Internally, the LLM represents the selected subsequence as a list of 1's and 2's, with 1's corresponding to numbers chosen and 2's corresponding to numbers not chosen. For the sequence [3, 2, -2, 5, 3], the expected output would be [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, again with each integer element restricted to $[-5, 5]$.

Parity   The parity problem is to determine if there is an even or odd number of 1's in a binary input array. 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. For an array with an even number of 1's, the output should be 0. This problem has been previously studied by Anil et al. (2022) and Zhou et al. (2023b). Traditionally, this problem is solved by traversing the input array to compute the sum modulo 2, which is the method we train our models to use. We generate binary arrays up to length 21 for training. For evaluation, we sample 100 binary arrays per length up to length 60.

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 paragraphs. Across all problems and model sizes, the Re-Tuning method outperforms the baseline and scratchpad methods, with the clearest difference being on integer addition. In particular, Re-Tuning exhibits significantly better OOD generalization compared to the baseline or scratchpad methods. We find this to be true even on language models with very few parameters, including Galactica 125M and Galactica 1.3B (see Appendix B).

Integer addition   The Re-Tuning method considerably outperforms the baseline and scratchpad methods. The scratchpad method performs the worst, achieving 0% accuracy on every problem longer than those seen during training on both LLaMA 7B and LLaMA 13B. The baseline method has non-zero OOD accuracy for problems up to length 20, but accuracy falls to 0% on longer problems with both models. In contrast, the Re-Tuning method maintains near-perfect accuracy in regimes where the baseline and scratchpad models have 0% accuracy, only falling below 90% accuracy on problems of length 40 and 45 for LLaMA 7B and LLaMA 13B respectively. Astonishingly, with Re-Tuning, both models maintain near 50% accuracy on adding up to 60 digit numbers. The model by Liu and Low (2023), which is also trained on addition up to 15 digits, has similar OOD performance to our baseline models, and falls to 0% accuracy when adding 21-digit numbers.

Dynamic programming   Again, Re-Tuning outperforms both the baseline and scratchpad approaches, though the gap between Re-Tuning and baseline is narrower for LLaMA 13B than it is for LLaMA 7B. Still, with Re-Tuning, both models achieve near 90% accuracy on problems of length 10, twice as long as the longest examples in the training data. Moreover, on problems of length 15, LLaMA 7B achieves 40% accuracy with Re-Tuning and 0% accuracy with the baseline and scratchpad methods. Dziri et al. (2023) trains and evaluates GPT3 models with and without scratchpad. Both reach 0% accuracy on problems of length
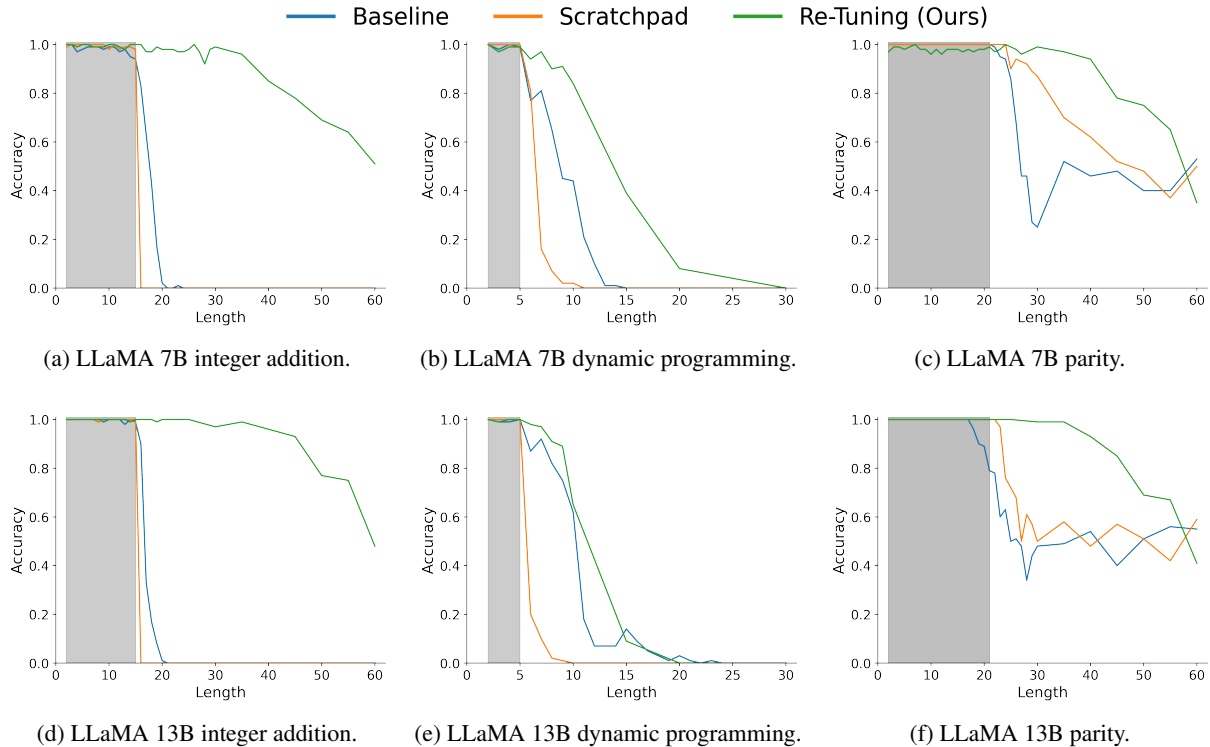
Figure 2: Performance of LLaMA 7B (top) and LLaMA 13B (bottom) on Addition (left), Dynamic Programming (middle), and Parity (right). The in-distribution range is shaded in gray.

10, which is similar to our scratchpad results but slightly worse than our baseline results.

Parity Re-Tuning performs as well or better on inputs smaller than size 60. Specifically, the accuracy of the baseline and scratchpad methods falls to that of random chance (50%) on problems with lengths greater than 40 on both models. Meanwhile, with Re-Tuning, LLaMA 7B and LLaMA 13B maintain an accuracy over 90% on problems with the length 40.

4 Analysis and Further Discussion

In this section, we conduct additional experiments in order to better understand the effects of various 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 than 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 toward longer problem instances. Due to Re-Tuning's recursive design, it's important that an appropriate

	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 training.

number of small problems are included in the training data. As such, we upsample examples with shorter lengths and downsample examples with longer lengths. Our resampling methodology is described in Appendix A.1

To better understand the impact of resampling, we train 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 Table 1.

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 achieves 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
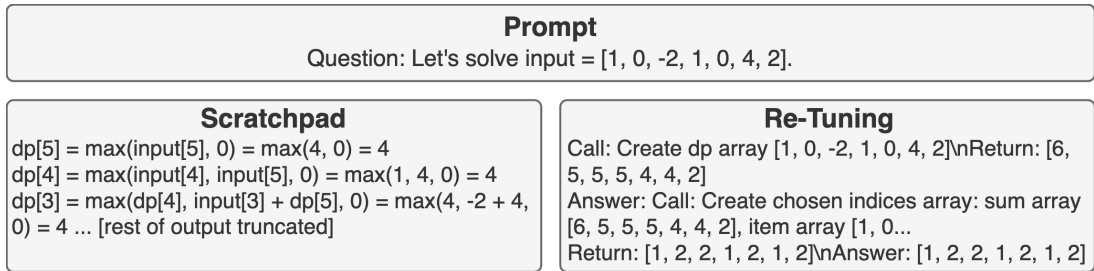
5

Figure 3: Case study on dynamic programming problem. With scratchpad, the model makes an indexing error, while with Re-Tuning, the model correctly generates the recursive call.

resampling maintains an accuracy of 69%, which suggests that resampling is an important contributor towards 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 training data includes arrays up to size 5. In Figure 3 we see that the model-generated scratchpad indexes element 5 instead of element 6 of the array of sub-array sums (dp array), which is incorrect on an input array of length 7. Once the model makes this indexing error on the scratchpad it is unable to recover. In other cases, 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. In contrast, the Re-Tuning method is shown in Figure 3. Only the initial context is shown to save space. With Re-Tuning, the model is able to generate recursive calls correctly with no difficulty indexing, enabling it to correctly solve the problem.

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 task, we randomly sample 20 problems per problem length and use LLaMA 7B with Re-Tuning to generate the outputs. We categorize these samples into the following error types:

- **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 model 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 on LLaMA 7B. Importantly, across all problems, the prevalence of errors increases with the size of the problem.

Integer addition On the integer addition task, very few call errors and compute errors occur before a problem size of 30. On more complex instances call errors frequently occur, suggesting that the model has a difficult time constructing the subproblem. Importantly, this aligns with Zhou et al. (2023b), which suggests 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 suggests that once a call error is made, the model has a very hard time recovering.

Dynamic programming For the dynamic programming problem, we perform error analysis on each subproblem separately. The first subproblem deals with constructing the array of sub-array sums,

6

(a) Errors on integer addition.

(b) Errors on dynamic programming (subproblem 1).

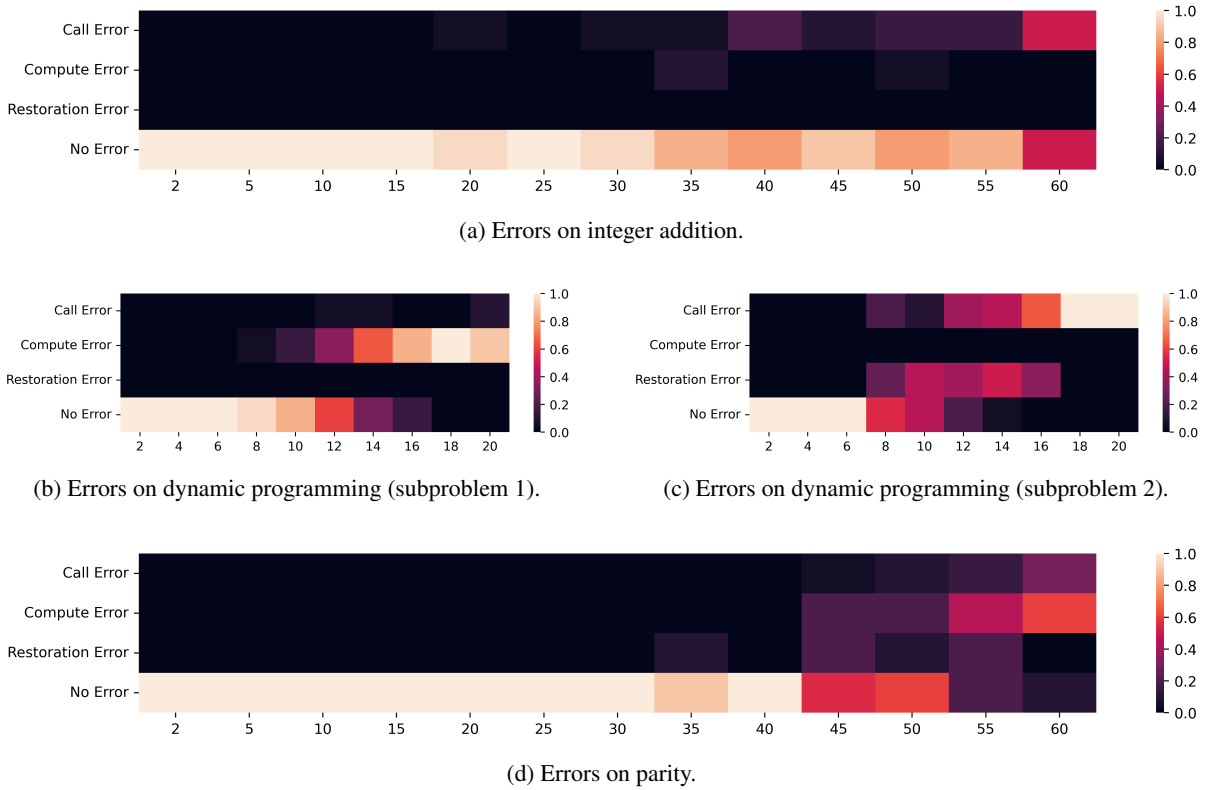(c) Errors on dynamic programming (subproblem 2).

(d) Errors on parity.

Figure 4: Error classifications for each problem across samples of 20 instances per problem lengths.

while the second 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 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 suggests that these call errors are easier to recover from than the computer errors made in subproblem 1.

Parity 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 parity flip in the parity problem.

4.4 Improved Sample Efficiency

We also run experiments to see the performance of Re-Tuning in the low-data regime on integer 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 two 1-digit numbers, 10 examples of adding two 2-digit numbers, etc. So, it sees 150 examples in total when n is 10. We train baseline, scratchpad, and Re-Tuning variants of LLaMA 7B on these examples for 5 epochs each. Unlike the other experiments, we do not use any resampling here.

After seeing only 50 examples per problem length, the Re-Tuning model achieves 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 checkpoint trained on the integer addi-

(a) 10 examples per length.   (b) 25 examples per length.   (c) 50 examples per length.
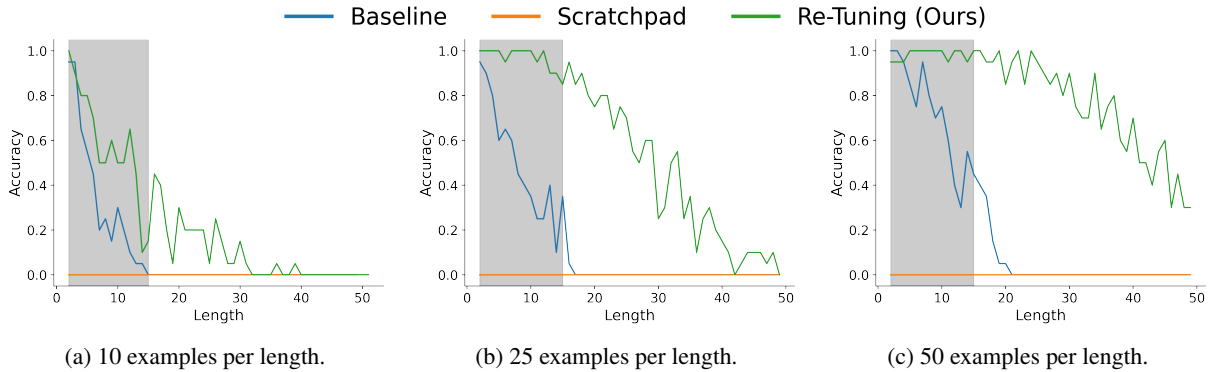
Figure 5: Results on integer addition with limited training data of 10 examples per length (left), 25 examples per length (middle), and 50 examples per length (right) on LLaMA 7B. Note that the scratchpad model does not get any problems correct.

	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.

tion task 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.

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: ' 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.

4.6 Why is Re-Tuning so Effective?

As discussed in Section 4.4, Re-Tuning exhibits significantly higher sample efficiency than the scratchpad or baseline methods. However, there are other factors at play that also contribute to the success of Re-Tuning.

With Re-Tuning, an LLM generates a recursive call with a subproblem to be solved in a separate recursive context. Once the subproblem is solved, the solution is returned to the original context. This

approach has two advantages. First, the computation required in each context is limited. In contrast, the scratchpad approach requires a long chain of computations must be performed in the same context. Second, each context includes only the necessary information to solve the current subproblem, as information that is irrelevant to solving the subproblem is filtered out. For example, to add 2 10-digit numbers, one of the prompts generated by Re-Tuning will be to add 2 3-digit numbers. This prompt has filtered out 7 digits from each number that would be irrelevant when adding these 3-digit numbers. This is in contrast to scratchpad prompting, where the model would also add these 3-digit numbers, but would have the full 10-digit numbers in context.

Furthermore, an important contributor to the success of LLMs on arithmetic tasks is the consistent tokenization of numbers (Nogueira et al., 2021; Kim et al., 2021). Fortunately, the LLaMA family of models (as well as the Galactica models presented in Appendix B) performs digit-level tokenization, where long numbers are split into individual digits for tokenization. Since all of our tasks are arithmetic in nature, it's likely that some of the success of Re-Tuning can be attributed to digit-level tokenization.

4.7 Efficiency Comparison

Due to Re-Tuning's high degree of sample-efficiency (see Figure 5) and shorted training sequences relative to the scratchpad method (see Figure 1b), Re-Tuning is an efficient and effective training paradigm to improve the performance of LLMs on compositional tasks. Still, generation takes longer with Re-Tuning than with baseline or

8

Length	Re-Tuning	Scratchpad	Baseline
5	4.462	2.492	0.346
30	76.494	32.509	1.496
60	265.002	80.699	2.030

Table 3: Generation times (in seconds) on the integer addition task with LLaMA 13B for a selection of problem lengths.

scratchpad methods. This is because Re-Tuning generates additional tokens related to calling the subproblem(s) within each context. To better understand this, we track the average generation time (in seconds) on the integer addition task across a selection of problem lengths. Specifically, we use LLaMA 13B at half-precision running on a single NVIDIA A100 GPU. Table 3 displays the results.

Though this would appear to be a limitation with Re-Tuning, there are two additional factors to consider. First, Re-Tuning prioritizes effectiveness over efficiency. Though all three methods see near-perfect accuracy on problems of length 5, the baseline and scratchpad approach fail to solve a single problem of length 30 or 60 correctly, while Re-Tuning maintains near-perfect accuracy on problems of length 30, and near 50% accuracy on problems of length 60. Second, unlike the baseline and scratchpad methods, Re-Tuning can leverage cache-based optimizations to retrieve solutions to subproblems without the need to generate using the model, saving time and compute resources.

5   Related Work

Several works have explored the length generalization ability of LLMs 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 training on a combination of in-context learning and scratchpad prompting could enable better performance. Similarly, Re-Tuning involves training pretrained LLMs to make recursive calls in order to improve performance on compositional 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 transformer models from scratch on scratchpad data could enable better length generalization. Recently, McLeish et al. (2024) showed that transformers can achieve strong OOD performance on

addition by using special positional embeddings.

Various papers have explored the idea of LLMs prompting themselves or other LLMs, although, to our knowledge, no papers explicitly train 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 a way to improve the logical consistency of the generated responses (Crispino et al., 2024; Imani et al., 2023; Madaan et al., 2023). Weston and Sukhbaatar (2023) use a language model to generate prompts by extracting relevant information from the context. Similarly, Re-Tuning generates a recursive call 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 continuing 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 (Shen et al., 2024; 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 seen during training.

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 type, 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 compositional tasks. Experimental results on three representative compositional tasks demonstrate the effectiveness of our method. Our method not only significantly outperforms standard training and state-of-the-art methods, especially on out-of-distribution problem instances, but is also more memory efficient during training. We hope our method can be applied to more tasks where recursive computation is inherent and computational 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 recursive calls 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.

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*.

Nicholas Crispino, Kyle Montgomery, Fankun Zeng, Dawn Song, and Chenguang Wang. 2024. Agent instructs large language models to be general zero-shot reasoners. In *International Conference on Machine Learning*.

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.

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.

Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, Shashank Gupta, Bodhisattwa Prasad Majumder, Katherine Hermann, Sean Welleck, Amir Yazdanbakhsh, and Peter Clark. 2023. Self-refine: Iterative refinement with self-feedback.

Sean McLeish, Arpit Bansal, Alex Stein, Neel Jain, John Kirchenbauer, Brian R. Bartoldson, Bhavya Kailkhura, Abhinav Bhatele, Jonas Geiping, Avi Schwarzschild, and Tom Goldstein. 2024. Transformers can do arithmetic with the right embeddings.

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 multistep 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.

Rafael Rafailov, Archit Sharma, Eric Mitchell, Christopher D Manning, Stefano Ermon, and Chelsea Finn. 2023. Direct preference optimization: Your language model is secretly a reward model. In *Thirty-seventh Conference on Neural Information Processing Systems*.

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.

Jianhao Shen, Ye Yuan, Srbuhi Mirzoyan, Ming Zhang, and Chenguang Wang. 2024. Measuring vision-language stem skills of neural models. In *The Twelfth International Conference on Learning Representations*.

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 Experimental Details

In this section, we provide additional experimental details, including details related to the synthetic construction of the data, training process, and inference pipeline.

A.1 Data Construction

We highlight our pipeline for synthetic data construction in this section.

Seed data To construct the training data, we start by randomly generating a collection of seed data. On the dynamic programming and parity problems, this seed data is exhaustive (e.g., all possible binary arrays up to length 20). On the integer addition task, we randomly generate 304,000 pairs of numbers up to 15 digits long. Next, we generate the recursive solution, including the solutions to the recursive sub-problems, for each instance of the seed data. The union of the seed data and recursive sub-problems from the training data, which we format according to the method (baseline, scratchpad, and Re-Tuning).

Resampling In general, we upsample examples with smaller lengths and downsample those with larger lengths in our training data. There are 2 reasons for this. First, 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). Second, since Re-Tuning generates calls to all examples except the base case, it has trouble learning 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. Figure 6 displays the distributions of the training data before and after resampling with respect to length for the

integer addition, dynamic programming, and parity tasks.

Training dataset sizes After resampling, the final training datasets contain the following number of instances: 3,676,055 for integer addition, 342,187 for dynamic programming, and 124,780 for parity.

Validation and testing datasets For each problem, we generate 5 and 100 instances of seed data for a variety of problem lengths (both in-distribution and out-of-distribution) for the validation and testing splits respectively.

A.2 Training Details

Training and evaluation were done on NVIDIA H100, A100, and RTX A6000 GPUs, depending on the compute requirements of the job. Rather than train the full model, we train using low-rank adapters (Hu et al., 2022). Hyperparameters for training jobs are in Table 4. These hyperparameters apply to training 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 training 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. Our training code is a heavily modified version of the code from Rafailov et al. (2023).

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 4: Hyperparameters used for finetuning.

During training, cross-entropy loss is computed only on the parts of the sequence that the model will generate at inference time (c.f. grey vs. green highlighted text in the upper right of Figure 1a).

We train on 500,000 samples, performing multiple epochs if necessary, and checkpoint the state model at fixed intervals. 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). We

(a) Integer addition (before resampling).

(b) Dynamic programming (before resampling).

(c) Parity (before resampling).

(d) Integer addition (after resampling).

(e) Dynamic programming (after resampling).
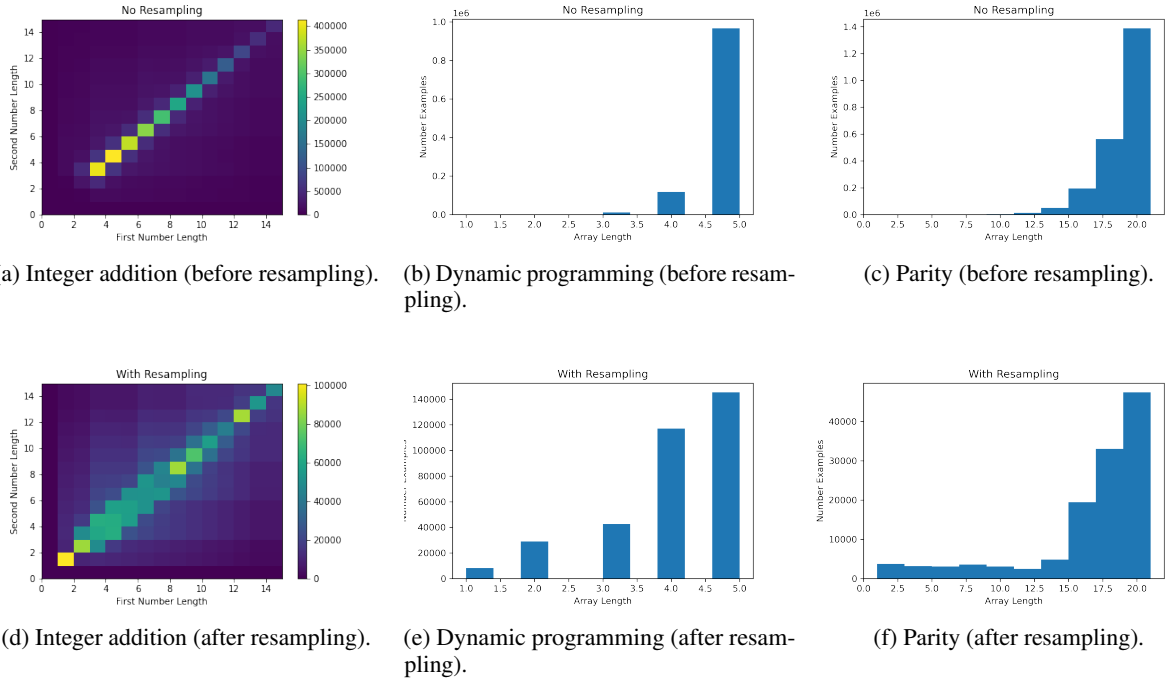
(f) Parity (after resampling).

Figure 6: Comparison of the training dataset distributions before (top) and after (bottom) resampling on integer addition (left), dynamic programming (middle), and parity (right).

Algorithm	Re-Tuning	Scratchpad	Baseline
Addition	71424	79360	349184
Parity	63488	71424	206336
DP	79360	71424	23808

(a) LLaMA 7B.

Algorithm	Re-Tuning	Scratchpad	Baseline
Addition	79360	103168	349184
Parity	404736	79360	206336
DP	404736	104000	23808

(b) LLaMA 13B.

Table 5: Number of samples the selected checkpoints were trained on for LLaMA 7B (top) and LLaMA 13B (bottom).

report full results on the checkpoint that achieves the highest accuracy on the validation set, selecting the earlier checkpoint in the event of a tie. Table 5 and provide the number of examples trained on for each task (integer addition, dynamic programming, and parity) and method (baseline, scratchpad, and Re-Tuning) for the selected LLaMA 7B and LLaMA 13B models.

A.3 Inference Pipeline

For baseline and scratchpad methods, our evaluation procedure is rather standard: we sample at a low temperature (0.01) and impose no additional

context limitations beyond those of the models themselves, in which case the input is truncated from the left. With Re-Tuning, we use a recursive wrapper around the same generation procedure, the pseudocode for which is shown in Algorithm 1.

To better understand the recursive generation procedure of Re-Tuning, let's consider the following integer addition example: "687 + 891\nSolution: ". With Re-Tuning, the model is trained to return the following subproblem call "Call: 87 + 91\n". In this case, we would extract the text "87 + 91" and prompt for the solution to this subproblem in a new context. Once we have the solution to this subproblem, it's returned to the main context "687 + 891\nSolution: Call: 87 + 91\nReturn: 178\nAnswer: ", and we again call the model to generate the final answer.

B Additional Results

In this section, we share results on two additional models: Galactica 125M and Galactica 1.3B (Taylor et al., 2022). Results across our 3 tasks are shown in Figure 7. In general, we observe that Re-Tuning enables Galactica 1.3B to maintain higher accuracy on more complex problem instances and Galactica 125M to perform as good, if not better, than either the baseline or scratchpad methods.

(a) Galactica 125M integer addition.   (b) Galactica 125M dynamic programming.   (c) Galactica 125M parity.

(d) Galactica 1.3B integer addition.   (e) Galactica 1.3B dynamic programming.   (f) Galactica 1.3B parity.
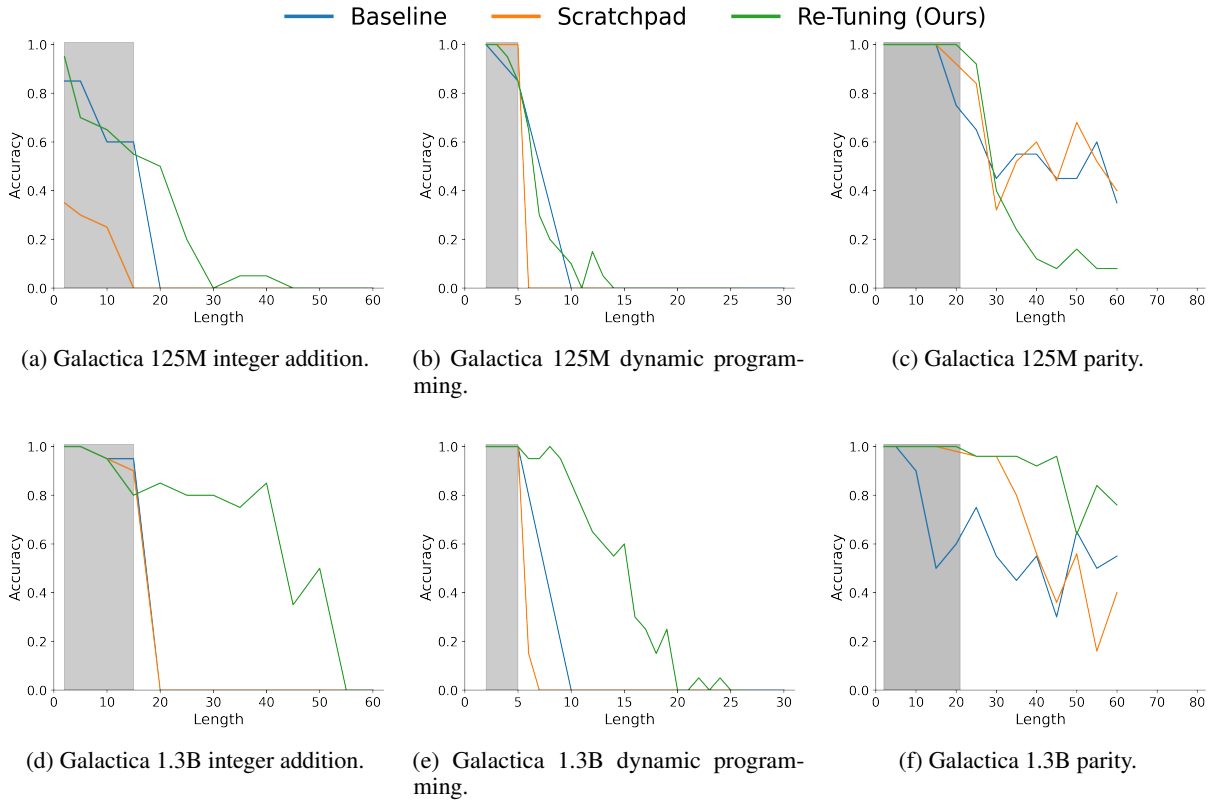
Figure 7: Performance of Galactica 125M (top) and Galactica 1.3B (bottom) on Addition (left), Dynamic Programming (middle), and Parity (right). The in-distribution range is shaded in gray.

Integer addition On the addition task on Galactica 125M, all three methods fail to achieve perfect in-distribution performance. With scratchpad, accuracy never tops 40%, even for lengths seen during training, and quickly falls to 0% at length 15. Though both baseline and Re-Tuning methods perform similarly on in-distribution problems, Re-Tuning generalizes better to more complex instances. With Re-Tuning, Galactica 125M is able to achieve greater than 50% accuracy at length 20, while the baseline accuracy at length 20 is 0%. With Galactica 1.3B, again all 3 methods display similar in-distribution accuracy above 80%. However, while the baseline and scratchpad methods are unable to solve any problems of length 20 or more correctly, Galactica 1.3B with Re-Tuning maintains near 80% accuracy on problems up to length 40.

Dynamic Programming On the dynamic programming task, Galactica 125M performs similarly with baseline and Re-Tuning methods, reaching 0% accuracy on problems of length 10 and 11 respectively. Galactica 125M with scratchpad is unable to solve any instances with out-of-distribution lengths correctly. On Galactica 1.3B, all three methods achieve perfect in-distribution

performance. Still, Re-Tuning displays better out-of-distribution performance, maintaining an accuracy near 60% on problems of length 15, while the baseline and scratchpad methods achieve 0% accuracy on lengths 10 and 7 respectively. Interestingly, while Galactica 1.3B with scratchpad fails to solve a single problem of length 7 correctly, Galactica 125M with Re-Tuning can still solve above 30% correctly, with just 1/10th of the parameters.

Parity Galactica 125M performs poorly on instances of the parity task with out-of-distribution lengths. All three methods perform similarly, and accuracy falls to that of random chance or worse on problems of length 30 or more. On Galactica 1.3B, the baseline method performs the worst, performing at the level of random chance on problems of length 15 or more. The scratchpad and Re-Tuning maintain near-perfect performance on problems up to lengths 30 and 45 respectively.

C Example Problems

In this section, we provide additional details and examples of the training data for all three tasks (integer addition, dynamic programming, and parity) with all three methods (baseline, scratchpad, and

Re-Tuning).

Integer addition   Following Liu and Low (2023), we generate pairs of numbers up to 15 digits in length. With the baseline method, we train the model to output the answer directly. With scratchpad and Re-Tuning, we train the model to add digits starting from the right. With these methods, the models learn to propagate the higher-order carry term separately from the rest of the output, which we found improved the accuracy of both methods. Evaluation is done on adding numbers up to 60 digits in length. To compute accuracy on testing examples, we extract the last number from the output and check for equivalence with the target solution. For the scratchpad and Re-Tuning methods, we first prepend the carry term to the output prior to computing accuracy. Figure 8 displays example training instances for the baseline, scratchpad, and Re-Tuning methods.

Dynamic programming   Following Dziri et al. (2023), the training data for this task consists of arrays up to 5 elements long, with each element restricted to $[-5, 5]$. With the baseline method, we train the model to directly output an indices array indicating which elements should be selected to maximize the sum given the constraints. The scratchpad design is copied form Dziri et al. (2023). Re-Tuning requires two recursive calls in each context: the first to create an array with sub-array sums and the second to construct the indices array. Evaluation is done on arrays up to length 30, with each element still restricted to $[-5, 5]$. For all three methods, we extract the last array from the generated text and check for equivalence to the target indices array. Figure 9 displays example training instances for the baseline, scratchpad, and Re-Tuning methods.

Parity   This problem was inspired by Anil et al. (2022), and requires the model to compute the parity of a binary array. Specifically, the training data includes binary arrays up to length 21, while the evaluation data includes arrays up to length 60. With the baseline method, the models learn to output the parity directly, which can be computed as the sum of the input array modulo 2. Our scratchpad design is similar to that of Anil et al. (2022), and involves keeping track of the parity sequentially as the array is traversed from left to right. With Re-Tuning, the parity is computed from right to left by recursive calls made to compute the par-

ity of the last $n - 1$ elements of the array. With all three methods, we extract the last digit from the generated sequence and check for equivalence with the target parity. Figure 10 displays example training instances for the baseline, scratchpad, and Re-Tuning methods.

```
Baseline:
637 + 123\nAnswer: 760

Scratchpad:
637 + 123\nSolution: Carry 1, Output 0 \nCarry 0, Output 60 \nCarry 0, Output 760

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

Figure 8: Example training instances for addition. The models learn to generate the text in bold.

```
Baseline:
637 + 123\nAnswer: 760
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]

Scratchpad:
Question: Let's solve input = [1, -3, 2]. Scratchpad: dp[2] = max(input[2], 0) = max(2, 0) =
2\ndp[1] = max(input[1], input[2], 0) = max(-3, 2, 0) = 2\ndp[0] = max(dp[1], input[0] + dp[2],
0) = max(2, 1 + 2, 0) = 3\n\nFinally, 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".\n\nLet can_use_next_item = True.\nSince dp[0] == input[0] + dp[2] (3 == 1 + 2)
and can_use_next_item == True, we store output[0] = 1. We update can_use_next_item = False.\nSince
dp[1] != input[1] (2 != -3) or can_use_next_item == False, we store output[1] = 2. We update
can_use_next_item = True.\nSince dp[2] == input[2] (2 == 2) and can_use_next_item == True, we
store output[2] = 1.\n\nReconstructing all together, output=[1, 2, 1].

Re-Tuning:
Compute the maximum sum of nonadjacent subsequences of [1, -3, 2]\nSolution: Call: Create dp array
[1, -3, 2]\n Return: [3, 2, 2]\nAnswer: Create chosen indices array: sum array [3, 2, 2], item
array [1, -3, 2], can use item True\nSolution: 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 [2, 2], item array [-3, 2], can use item False\nReturn [2, 1]\nAnswer:
Append 1 if False else 2.\nAnswer: [1, 2, 1]
```

Figure 9: Example training instances for dynamic programming. The models learns to generate the text in bold.

```
Baseline:
What is the parity of [1, 0, 1]?\nAnswer: 0

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

Re-Tuning:
What is the parity of [1, 0, 1]?\nSolution: Call: What is the parity of [0, 1]?\n\nReturn: 1\nAnswer:
0
```

Figure 10: Example training instances for parity. The models learns to generate the text in bold.