Case-Based or Rule-Based: How Do Transformers Do the Math?

Yi Hu [1]   Xiaojuan Tang [1 2]   Haotong Yang [1]   Muhan Zhang [1 2]

Abstract

Despite the impressive performance in a variety of complex tasks, modern large language models (LLMs) still have trouble dealing with some math problems that are simple and intuitive for humans, such as addition. While we can easily learn basic *rules* of addition and apply them to new problems of any length, LLMs struggle to do the same. Instead, they may rely on similar *cases* seen in the training corpus for help. We define these two different reasoning mechanisms as "*rule-based reasoning*" and "*case-based reasoning*". Since rule-based reasoning is essential for acquiring systematic generalization ability, we aim to explore exactly whether transformers use rule-based or case-based reasoning for math problems. Through carefully designed intervention experiments on five math tasks, we confirm that transformers are performing case-based reasoning, no matter whether scratchpad is used, which aligns with the previous observations that transformers use subgraph matching/shortcut learning to reason. To mitigate such problems, we propose a Rule-Following Fine-Tuning (RFFT) technique to teach transformers to perform rule-based reasoning. Specifically, we provide explicit rules in the input and then instruct transformers to recite and follow the rules step by step. Through RFFT, we successfully enable LLMs fine-tuned on 1-5 digit addition to generalize to up to 12-digit addition with over 95% accuracy, which is over 40% higher than scratchpad. The significant improvement demonstrates that teaching LLMs to use rules explicitly helps them learn rule-based reasoning and generalize better in length. Code is available at https://github.com/GraphPKU/Case_or_Rule.

1. Introduction

Large language models (LLMs) such as ChatGPT (OpenAI, 2022) and GPT-4 (OpenAI, 2023) have exhibited remarkable capabilities in a wide range of tasks from some classical NLP tasks such as translation and summarization to complex reasoning tasks about commonsense, math, logic and so on (OpenAI, 2022; 2023; Brown et al., 2020; Touvron et al., 2023a; Chowdhery et al., 2023; Thoppilan et al., 2022). Some people believe LLMs present a seemingly promising route to AGI (Bubeck et al., 2023). At the same time, some theoretical work also gives their support to this applauded prospect by proving that transformer-based LLMs can learn an intrinsic mechanism for some complex tasks such as linear regression (Akyürek et al., 2023), dynamic programming (Feng et al., 2023) or modular addition (Zhong et al., 2023; Nanda et al., 2023; Power et al., 2022; Liu et al., 2022).

Although LLMs have demonstrated impressive results and possibility both in performance and theory, they are, surprisingly, still puzzled by some basic calculation tasks (Qin et al., 2023; Bian et al., 2023; Koralus & Wang-Maścianica, 2023; Dziri et al., 2023; Xu et al., 2023; Zhou et al., 2023b). Notably, there has been a line of work paying efforts to teach transformers to perform *addition of two large numbers* (Nye et al., 2021; Qian et al., 2022; Zhou et al., 2022; 2023b; Shen et al., 2023; Kazemnejad et al., 2023; Lee et al., 2023; Zhou et al., 2024). Despite ongoing efforts, transformers have yet to successfully generalize to new inputs that are significantly longer than the training data, without relying on external tools. In contrast, humans can easily solve addition of two numbers of any length after learning basic rules of column addition. Language models often astonish us with their proficiency in complex tasks, yet they can also perplex us with unexpected failures in seemingly straightforward tasks. This dichotomy in performance raises intriguing questions about their underlying reasoning mechanisms.

Previous work has argued over the open questions. Nanda et al. (2023); Zhong et al. (2023) study how transformers do modular addition and claim that they derive certain algorithms to solve the problem, such as the clock algorithm where input numbers are represented as angles and then added together. However, another line of work (Dziri et al., 2023; Wu et al., 2023; Zhang et al., 2023) worries that the

[1]Institute for Artificial Intelligence, Peking University [2]National Key Laboratory of General Artificial Intelligence, BIGAI. Correspondence to: Muhan Zhang <muhan@pku.edu.cn>.
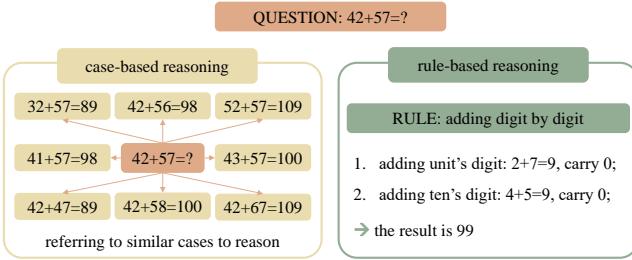
Figure 1. Illustrations of case-based and rule-based reasoning.

impressive reasoning ability of LLMs can be mainly attributed to the extensive training corpus. They argue that transformers are just recalling similar instances from seen data to solve reasoning tasks instead of capturing underlying rules and applying them to new problems.

In this paper, we study the hotly-debated questions more directly through intervention experiments. We hypothesize that transformers significantly depend on certain cases in training data to do math reasoning, which we denote as "case-based reasoning". It should be noted that here by "case-based reasoning" we do not mean a non-parameterized machine learning algorithm that really retrieves similar cases from a database. Rather, we describe a behavior that transformers show in reasoning. Specifically, if a model employs case-based reasoning, the removal of those dependent cases from the training set would significantly affect its accuracy on certain test examples. On the contrary, if a model does not rely on similar cases but instead masters the underlying *rules* for reasoning—a mechanism we define as "rule-based reasoning"—the absence of these cases should not affect the performance. An illustration of these two contrasting reasoning paradigms is shown in Figure 1.

To verify our hypothesis, we fine-tune LLMs respectively on five basic and representative math tasks: addition, modular addition, base addition, linear regression, and chicken & rabbit problems. As a sanity check for each task, we first make sure that the model achieves 100% performance on the test set when the dataset is randomly split. Then, we artificially split the dataset by leaving out some continuous regions of examples as the test set with the remaining ones as the training set and re-train the model. This method ensures that most test examples do not have close training cases to support their inference. Our results show that in all tasks, the model performance drops significantly in the second setting, despite that the size of the training set (above 95% of the whole dataset) is entirely sufficient to achieve 100% accuracy under random split. See Figure 2 and Figure 3 for example.

The results of our intervention experiments provide direct evidence suggesting that transformers perform case-based

reasoning for math problems. This also aligns with previous work (Dziri et al., 2023) showing transformers rely on seen computation subgraphs for multi-step reasoning. However, there are notable distinctions in our approach and findings: Dziri et al. (2023) look at the frequency difference of seen subgraphs in correct and incorrect samples respectively as indirect evidence that models rely on seen subgraphs to generate correct answers, while we present direct evidence of case-based reasoning by showing the performance gap before and after removing the cases. Besides, we study both single-step and multi-step reasoning while Dziri et al. (2023) mainly focus on compositional reasoning.

So why is rule-based reasoning so important? Rule-based reasoning is essential for models to achieve systematic and length generalization so that they can be applied to new, unseen scenarios without re-training. As our last contribution, we propose a method to shift transformers from case-based to rule-based reasoning, thereby fostering a more robust and generalizable reasoning paradigm. Focusing again on the addition of large numbers, we propose a technique that teaches transformers to follow rules step by step. Specifically, we explicitly put rules in the input and enforce the model to step-by-step recite and follow the necessary rules to complete reasoning, which we call Rule-Following Fine-Tuning (RFFT). Through RFFT, LLMs trained on addition of numbers of 1-5 digits successfully generalize to up to 12-digit addition, verifying its effectiveness in teaching LLMs to perform rule-based reasoning. It is noteworthy that the training set is as small as 100 samples, demonstrating that RFFT enables models with sufficient fundamental capabilities to grasp the rules through a small set of examples, which aligns with humans' few-shot rule learning ability.

2. Related Work

LLM reasoning. Recent years have seen enormous improvement in LLMs' capabilities. LLMs show impressive performance in a wide range of tasks (OpenAI, 2022; 2023; Brown et al., 2020; Touvron et al., 2023a; Chowdhery et al., 2023; Thoppilan et al., 2022). However, various tasks of complex reasoning are still challenging for LLMs (Srivastava et al., 2022). In particular, Dziri et al. (2023); Xu et al. (2023); Zhou et al. (2023b; 2024) show that LMs still struggle with math reasoning, even with basic calculation operations.

Previous work has come up with methods to simplify the tasks by decomposing them to simpler intermediate steps. For example, Nye et al. (2021); Zhou et al. (2022) introduce finetuning models with cases containing scratchpads to improve arithmetic reasoning of LLMs. Wei et al. (2023); Kojima et al. (2023); Zhou et al. (2023a); Khot et al. (2023); Zhu et al. (2023) propose various prompting methods to teach the model to generate rationales before the final an-

swer with in-context learning. However, even with these methods, LLMs are still far from completely solving arithmetic reasoning tasks. The failures inspire us to study how exactly LLMs perform math reasoning. Besides, we study the effects of the methods of task simplification on case-based reasoning in our paper. Specially, Zhu et al. (2023) improve the model performance by providing the cases the reasoning process may depend on in the input, which in fact aligns with our case-based reasoning paradigm.

Memorization or generalization. As reasoning capabilities of LLMs can be mainly attributed to the scaling effects of the training corpus and the model size, the question of whether the seemingly impressive reasoning abilities are the results of capturing general rules lying under the natural language or just reciting seen cases from the huge training corpus is drawing more and more attention. Wu et al. (2023); Zhang et al. (2023) investigate into the gap of capabilities of LLMs to conduct reasoning over factual and counterfactual situations and show the significant performance drop in counterfactual cases, suggesting LLMs are reciting answers of common cases in the training corpus. A recent work Dziri et al. (2023) models reasoning tasks as computation graphs and show empirically that LLMs conduct reasoning via subgraph matching instead of developing systematic problem-solving skills. We study the question of interest in a straightforward way by removing certain samples from the training set and show significant performance gap. By tracing back to the effective training datapoints, we confirm that transformer-based LLMs are relying on surrounding cases in the training set to do math reasoning instead of learning generalizable rules. On the other hand, Hou et al. (2023) study the problem through probing the models' attention patterns and claim that transformers are implementing reasoning trees in the reasoning process. Yang et al. (2023) propose that LLM's reasoning ability comes from memorizing some templates, which are some fixed parts in the reasoning process, enabling generalization within tasks.

Grokking. Recent work has shown the phenomenon of model capturing generalizable rules of arithmetic reasoning tasks long after overfitting the training set, known as grokking (Power et al., 2022; Liu et al., 2022). Nanda et al. (2023); Zhong et al. (2023) study the algorithms transformers learn in the task of modular addition. The series of work show through experiments that the model learns systematic rules to solve modular addition through embedding the numbers as angles and operating on their trigonometric functions. We also try to observe the phenomenon in the same setting as in Zhong et al. (2023) with certain samples removed from the training set. Although we observe the growth of test performance after the model overfitting the training set, there is still a wide gap between training accu-

racy and test accuracy, suggesting the model fails to learn the rules. This phenomenon indicates that even the ability to learn and apply generalizable arithmetic algorithms in grokking deeply depends on certain cases in the training set. The results and experiments are described in Appendix H.

Theoretical expressiveness. (Feng et al., 2023; Akyürek et al., 2023; Dai et al., 2023; von Oswald et al., 2023; Garg et al., 2023) There have been a large number of work studying the expressive power of transformers. Yun et al. (2020) proved that transformers are universal approximators of continuous sequence-to-sequence functions on a compact domain. More recently, Garg et al. (2023) reveals that auto-regressive transformers can learn basic functions including sparse linear functions, MLPs and decision trees. Furthermore, Akyürek et al. (2023) demonstrates that transformers can in-context learn linear regression by implementing the algorithm of gradient descent (Dai et al., 2023; von Oswald et al., 2023). Feng et al. (2023) shows how chain-of-thought prompting help transformers complete tasks including basic calculations, linear equations and dynamic programming. In our work, we conduct empirical experiments and show how auto-regressive transformers do basic math reasoning in practice. We include tasks like addition, linear regression and linear functions that have been studied in the theoretical work.

Length generalization. Length generalization calls for the ability to generalize to longer sequences than seen in training samples, which remains a challenge for transformers (Abbe et al., 2023; Anil et al., 2022; Zhou et al., 2023b). Previous work has shown that data format and positional encoding are crucial to length generalization ability through experiments on small transformers across various tasks such as arithmetic reasoning (Lee et al., 2023; Kazemnejad et al., 2023; Shen et al., 2023; Zhou et al., 2023b; 2024). However, these works require specifically designed tricks for each task and train small transformers from scratch. Our work explores length generalization in the settings of fine-tuning pre-trained LLMs and shows that the technique of RFFT we propose in §5 greatly enhances length generalization. Furthermore, we demonstrate that the models with sufficient fundamental capabilities can generalize well with only a small set of training samples.

3. Case-based and Rule-based Reasoning

One main focus of our paper is to discuss whether auto-regressive transformer-based language models are solving basic math problems based on cases or rules. In this section, we intuitively motivate these two reasoning paradigms and provide a direct method to distinguish them through data intervention.

Case-based Reasoning. A model engaging in case-based reasoning exhibits sensitivity in its test performance to the division of the dataset into training and test sets. Specifically, if a model relies on shortcuts, either by referencing similar cases encountered during training or by merely repeating previously seen examples to solve new problems, its effectiveness diminishes when these cases are removed from the training set. This reduction in relevant training data results in a notable decrease in the model's ability to accurately respond to test questions.

Rule-based Reasoning. In contrast to case-based reasoning, the paradigm of rule-based reasoning allows the model to learn the underlying rules, which are insensitive to the data split. For example, if a model is developing the systematic rules of addition during the training process, its test performance should not be affected severely if we leave some of the training samples out of the training set and add some others to keep the same training-test ratio. It should be noted that the training set should always provide the necessities for the model to learn the underlying rule. For example, the training set should at least cover all the tokens used in the test set in order to develop a systematic rule that applies to the whole dataset. In all our experiments, we carefully design the setups to ensure the above.

Based on the above discrimination, we propose a natural method to determine whether a model is performing case-based reasoning or rule-based reasoning through data intervention. That is, we artificially remove certain regions of the training data to see its effect on test performance. For example, in math reasoning tasks such as addition, if the model is severely relying on some seen cases to do reasoning, a natural hypothesis is that it is relying on some surrounding cases of the test question, as shown in Figure 1 left. Based on the hypothesis, we can remove a small set of surrounding cases from the training set and see whether the model can still answer the question. If it succeeds when we leave the surrounding cases in the training set but fails when we take them out, we can judge that the model is relying on the small set of surrounding cases to do math reasoning. Otherwise, if the model can perform well in the test set no matter how we split the dataset, it is likely performing rule-based reasoning which guarantees robust generalization independent of dataset split.

It is important to recognize that rule-based reasoning also involves a degree of memorization. For example, in the process of digit-by-digit addition, we inherently rely on memorized knowledge of possible single-digit sums. Take the calculation of 42+57 as an instance; it is essential to know that 2+7 equals 9 and 4+5 equals 9. We refer to this fundamental knowledge required for rule-based reasoning as "unit rules". These unit rules are tailored to specific reasoning patterns. The more basic these unit rules are, the less

memorization the reasoning process requires, indicating a more pronounced reliance on rule-based reasoning. Conversely, if a model relies on case-based reasoning through sheer memorization—learning that 42+57 equals 99 only by encountering this exact case, then the unit rules for this pattern of reasoning are the cases themselves.

So how do we judge whether the unit rules are elemental enough to ensure a rule-based rather than case-based reasoning? We define the model is performing rule-based reasoning if the set of unit rules the model requires to solve the task is **finite** and can be easily covered by a training set of a reasonable size. Otherwise, if it is hard or even impossible for a training set to cover all the unit rules, we consider the model performing case-based reasoning.

4. Transformers are Doing Case-based Reasoning

In this section, we provide direct evidence that transformers perform case-based reasoning through intervention experiments on five representative math tasks.

4.1. Experimental Setup

Datasets We focus on binary operations, which take two numbers a, b as inputs. Denoting c as the target label, we construct datasets like $\mathcal{D} = \{((a_i, b_i), c_i)\}$ for five math tasks including addition, modular addition, base addition, linear regression, and chicken & rabbit problem:

- **Addition.** The input to the transformer is "$a + b$", the output is "c", where $c = a + b$. a, b range from 0 to 99.
- **Modular addition.** The input to the transformer is "$a + b$", the output is "c", where $c = a + b \mod P$. a, b range from 0 to 112. We set $P = 113$ as a constant.
- **Base addition.** This task is the same as addition, except that all numbers a, b, c are expressed in the base-n numerical system. In this paper, we set $n = 9$ as a constant.
- **Linear regression.** This task requires the transformer to learn a linear regression function. The input is "$(a, b) =$", the output is "c", where $c = m \cdot a + n \cdot b + p$. a, b range from 0 to 99. We set $m = 1, n = 2, p = 3$ as constants.
- **Chicken & rabbit problem.** We construct a dataset of chicken & rabbit problems with natural language questions and answers. The input to the transformer is "Q: Rabbits have 4 legs and 1 head. Chickens have 2 legs and 1 head. There are a legs and b heads on the farm. How many rabbits and chickens are there?". The output is "A: There are c rabbits and d chickens.", where $c = (a - 2b)/2, d = (4b - a)/2$. b ranges from 0 to 99. For each b, a ranges from $2b$ to $4b$ with a step of 2. It is a representative task involving solving a system of linear equations.
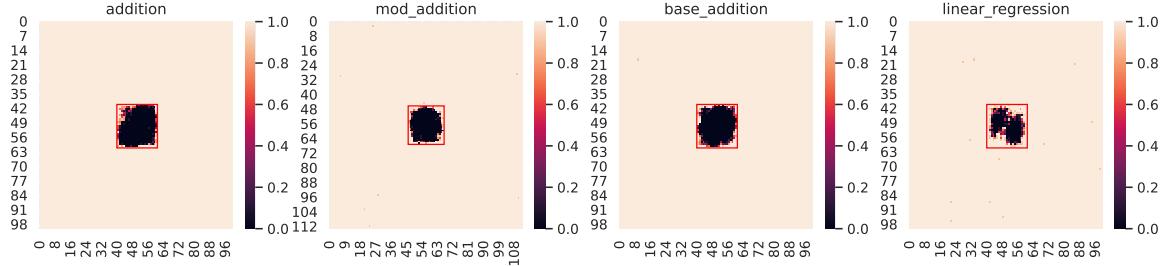
Figure 2. Accuracy of Leave-Square-Out method on addition, modular addition, base addition, and linear regression. The vertical and horizontal axes are a and b, respectively. The area inside red boxes represents the test squares. During generation, we set the model temperature to 1 and sample 10 generations to evaluate the accuracy on each test point. We only leave one test square out in this experiment. The square center (a_k, b_k) is $(50, 50)$ for addition, base addition and linear regression and $(56, 56)$ for modular addition.
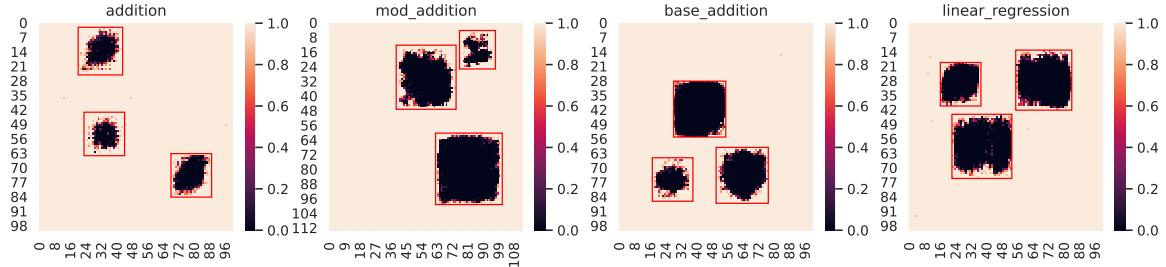
Figure 3. We randomly select 3 centers of test squares (a_k, b_k) and corresponding lengths l_k ranging from 20 to 40 to see whether the locations and the side lengths affect the case-based reasoning behavior for datasets including addition, modular addition, base addition and linear regression. The area inside red boxes represents the test squares. We sample 10 generations at each data point and report the accuracy. The figure shows that holes consistently appear with the locations and side lengths of the test squares varying.

Models We use GPT-2, GPT-2-medium (Radford et al., 2019), and Llama-2-7B (Touvron et al., 2023b) in this section. We fine-tune GPT-2 by default respectively on each dataset for 100 epochs under different training-test splits, with batch size set to 30 and learning rate set to 10^{-4}.

4.2. Method

Leave-Square-Out To test whether the model is relying on certain cases to solve the problem, we need to first locate such cases and then remove them from the training set to see whether they affect the model performance. Our hypothesis is that when facing a certain test sample, transformers tend to rely on training samples "close" to the test sample to perform reasoning. Thus, we construct a square test set to **isolate the test samples from the training samples**. For example, suppose the square center is (a_k, b_k) and the side length is l_k, we construct a square test set as $\mathcal{T}_k = \{((a_i, b_i), c_i) \mid a_k - \frac{l_k}{2} \le a_i \le a_k + \frac{l_k}{2}, b_k - \frac{l_k}{2} \le b_i \le b_k + \frac{b_k}{2}\}$. All the remaining samples constitute the training set. According to our hypothesis, case-based models should fail to generate correct answers for test samples near (a_k, b_k), as there are no close cases in the training set.

4.3. Appearance of Holes Verifies Case-Based Reasoning

In our study, we apply the Leave-Square-Out method to each dataset. Specifically, we extract a square comprising 441 samples (from a total of approximately 10,000 samples) with a side length of 20 to form our test set, leaving the remainder as the training set. It is important to note that, despite removing a small portion of training samples, we ensure that all tokens present in the dataset appear in the training set. This precaution is to prevent the models from failing simply due to encountering unseen tokens. We then proceed to fine-tune GPT-2 and GPT-2-medium models using this specific training-test split for each dataset. For comparison, we also fine-tune these models on datasets that are randomly split, where each training set comprises 70% of the total dataset.

Models achieve 100% accuracy easily in the random split settings across all datasets, which suggests that the size of training sets in the Leave-Square-Out setting (above **95%** of each dataset) is totally sufficient to complete the task. However, in the Leave-Square-Out setting, as shown in Figure 2, there are "holes" appearing in the accuracy distribution of the test squares over a and b. The appearance of holes in the figure indicates that the test samples away from the boundary of the training set are hard for the models to correctly infer, while the models can easily handle the test samples

near the boundary. This suggests that in the basic math reasoning tasks, when faced with an unseen test case, transformers **rely on the surrounding training cases** to predict the answer, verifying the case-based reasoning hypothesis. As for random split, every test sample has close training samples to support its inference, thus reaching 100% accuracy. In Figure 2, we only show the results of GPT-2 on the first four tasks; the results of GPT-2-medium and the results of chicken & rabbit problem are shown in Appendix D.

4.3.1. DO LOCATIONS SIZE OF TEST SQUARES MATTER?

To see whether the locations of test squares affect the experimental results, we randomly select three square centers (a_k, b_k) and three corresponding side lengths l_k ranging from 20 to 40 for each dataset. As shown in Figure 3, holes consistently appear in various locations of the dataset, suggesting that the model behavior of performing case-based reasoning does not change with the location of test sets.

4.3.2. DOES THE SIZE OF TEST SQUARE MATTER?

We test on various lengths of test squares including l_k set to 10, 20, 30, 40 with GPT-2 and GPT2-medium. As shown in Figure 5, test accuracy drops with the side length of the test square increasing. This phenomenon is natural in the context of case-based reasoning. As the test square becomes larger, the ratio of test samples that do not have close supporting training samples becomes higher, thus decreasing the test accuracy. Besides, it is shown in Figure 5 that GPT-2 achieves 100% accuracy when we set l_k to 10. In other words, **the hole disappears when the test square shrinks to less than a small size** where all the samples in the test set have close training samples for the model to refer to.

4.3.3. DOES ADDING SCRATCHPAD HELP?

Nye et al. (2021) has proposed a technique of teaching models to explicitly generate intermediate computation steps into a "scratchpad" before arriving at the final answer to improve their math reasoning capabilities. The scratchpad technique enables the model to decompose addition into incremental digit-by-digit operations, potentially reducing the model's dependence on surrounding cases. An example input-output pair of scratchpad is shown in the bottom left of Figure 6 (scratchpad). We employ scratchpad fine-tuning to examine its impact on the model's tendency towards case-based reasoning, specifically investigating whether the scratchpad technique can enable transformers to perform rule-based reasoning.

In particular, we alter the input of the addition dataset by providing scratchpad steps of adding two numbers digit by digit before presenting the final answer, instead of directly providing the answer following the question. Then we perform Leave-Square-Out on the altered dataset with GPT-2

and GPT-2-medium. The test accuracy vs. side length results are also shown in Figure 5. In the settings where side length of the left-out square $l_k \geq 20$, adding the scratchpad greatly boosts the model performance. However, for $l_k = 10$, models trained with scratchpad inputs lag behind those trained with direct answers. Besides, the test accuracy of models trained with scratchpad maintain relatively stable with the increase of test square's side length, in contrast to the sharp decline in performance seen in models trained with direct answers. To explain the phenomenon, we show the test accuracy distribution over a and b of models trained with scratchpad in Figure 4. It is clear that the model behavior of relying on cases "nearby" to solve new problems has changed. The holes shift to (a series of) triangles with their hypotenuses along the "carry boundary" at the unit's and ten's digits. For example, in the setting of $l_k = 20$ (the second subfigure), there are two triangle holes where the model shows almost zero accuracy. We explain why the model fails in each triangle and why the model succeeds in the rest of the test set as follows. Firstly for the small triangle, the model fails to answer questions like 47+48. 47+48 can be decomposed into 2 steps: 7+8=5, carry 1; 4+4+1=9. As **there are no cases in the training set containing the step of 4+4+1 in the ten's digit, the model fails**. In contrast, for those test points that do not involve carry in the ten's digit, like 42+43, the model succeeds because it can learn 4+4=8 from plenty of training data. Secondly for the large triangle, the model fails to answer 57+58. 57+58 can be decomposed into 2 steps: 7+8=5, carry 1; 5+5+1=1, carry 1. As there are no training cases performing 5+5 in the ten's digit which requires carry 1 to the hundred's digit, the model fails.

The shapes and locations of the holes indicate that the models succeed in test cases where **every step of the corresponding scratchpad has appeared in the training set** and fail otherwise. This conclusion aligns with Dziri et al. (2023) that transformers rely on seen computation subgraphs for complex reasoning. More importantly, this phenomenon demonstrates even **scratchpad cannot teach transformers to perform rule-based reasoning**—the models still **mechanically recite the seen unit rules**, but fail to flexibly generalize them.

4.3.4. DOES THE MODEL AND DATA SIZE MATTER?

As the emergent ability (Wei et al., 2022) suggests that the model size is crucial to unlocking a wide range of complex tasks, we first explore the effects of model size on the reasoning mechanisms through experiments on GPT-2, GPT-2-medium and Llama-2-7B. GPT-2 has 124M parameters, and GPT-2-medium has 355M parameters. As shown in Figure 5, when trained with direct answers instead of scratchpads, GPT-2-medium generally outperforms GPT-2 when $l_k \geq 20$. On the contrary, GPT-2-medium lags behind GPT-2 when $l_k = 10$. Besides, when trained with scratch-
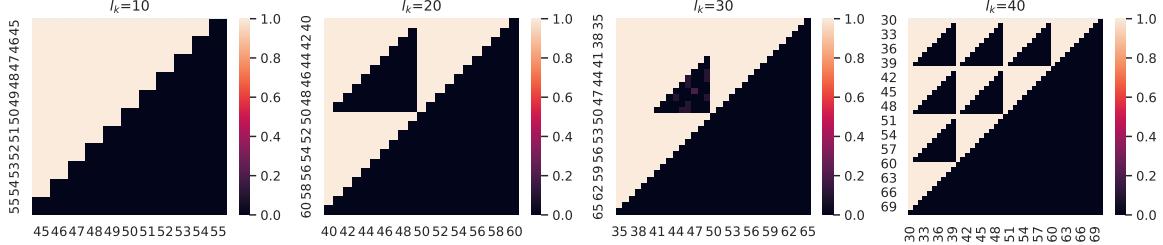
Figure 4. Test accuracy distribution of GPT-2 trained with scratchpad in the task of addition. Note that all points in the figure are test samples; each subfigure here corresponds to a left-out square in the original plane. From left to right, the side length of test square is set to $l_k = 10, 20, 30, 40$. For each test point, we sample 10 generations and show the accuracy of generating the correct answer.
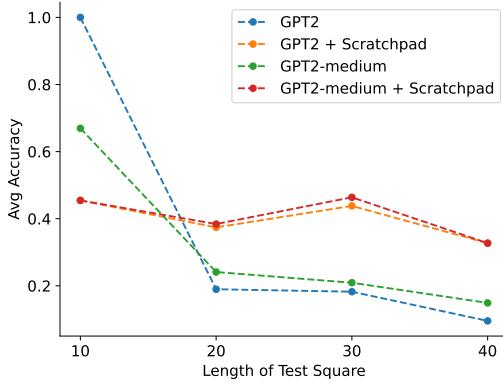
Figure 5. In the task of addition, we show the average accuracy over all test samples (samples within the square) with side length $l_k = 10, 20, 30, 40$. We test four models: GPT-2, GPT-2 with scratchpad, GPT-2-medium and GPT-2-medium with scratchpad.

pad, GPT-2-medium performs slightly better than GPT-2. Overall, model size has a more pronounced impact on test performance in scenarios of training with direct answers, as opposed to training with scratchpad, probably because the single steps in scratchpad is easier to memorize. We put the experiments on Llama-2-7B and GPT-3.5 to Appendix E.2 and Appendix E.3. Both models show holes within the test square, indicating that the trend of case-based reasoning still exists.

We also study how data size affects the behavior of case-based reasoning. We expand the range of a, b from 100 to 200 and 500, respectively. We also scale up the side length of the test square linearly with the data range. With the increasing of data size, the holes still appear, suggesting that increasing the data size helps little. We show the test accuracy distribution in Appendix E.1.

4.4. In-context Learning

Another aspect of LLMs' reasoning ability is attributed to in-context learning (ICL). This method draws upon knowledge not only ingrained during the pre-training but also from spe-

cific examples supplied within the context. The underlying mechanisms that make ICL effective are among the most intriguing and unanswered questions in the field. We extend our investigations to ICL in Appendix C, revealing that LLMs' ICL reasoning ability also exhibits characteristics of case-based learning.

5. Teaching Transformers to Do Rule-Based Reasoning by Rule-Following Fine-Tuning

In §4, we show that transformers are performing case-based reasoning in a wide range of math problems. However, the case-based reasoning behavior sets strong limits to the generalization ability of transformers. To be more specific, based on the results in §4, transformers rely on surrounding cases to do addition, so they naturally cannot generalize in length by training on finite-digit addition data. In contrast, rule-based reasoning can robustly generalize in length. In this section, we explore how to teach transformers to do rule-based reasoning.

We first revisit the failure of the scratchpad attempt. Despite providing step-by-step intermediate computations, scratchpad fine-tuning fails to teach transformers the **actually applied "rule"** behind each step. This is like teaching children addition **only by showing them examples**, without telling them the **rationales behind each step**. Motivated by this intuition, we propose Rule-Following Fine-Tuning (RFFT) to explicitly teach transformers to use rules at each step.

RFFT has two steps. First, we explicitly list the rules for solving a given task in the input. For example, in the task of addition, we provide the code of adding two long integers digit by digit in the input. It should be noted that there are various ways to represent the rules, including programs, pseudo-code, first-order logic, natural language, etc. We use programs in this section, and explore using natural language representations of rules in Appendix F.5. Second, we fine-tune the model to follow the rules step by step. Specifically, the model need to explicitly recite which rule it is using in each step, as well as updating the intermediate variables after applying this rule, as shown in Figure 6 right.

7

```
direct answer

input:
59+13=

output:
72


scratchpad

input:
59+13=

output:
59+13, , C:0
# added 9+3+0=2
5+1, 2, C:1
# added 5+1+1=7
+, 72, C:0

72
```

```
input:
Follow the code step by step to
answer the question:
[5, 9]+[1, 3]=
def sum_digit_by_digit(num1, num2):
    result=[]
    carry=0
    while num1 or num2:
        digit1=num1.pop() if num1 else 0
        digit2=num2.pop() if num2 else 0
        total=digit1+digit2+carry
        result.insert(0,total%10)
        carry=total//10
    if carry:
        result.insert(0,carry)
    return result

output:
num1 = [5,9]
num2 = [1,3]
1. Initialize Result and Carry
result = []
carry = 0
```

```
rule following

2. Loop Through Each Digit
```while num1 or num2:```
2.1 check the stop criterion
num1 = [5,9]
num2 = [1,3]
bool(num1) = True
bool(num2) = True
num1 or num2 = True
enter the loop
2.2 one iteration
```digit1=num1.pop() if num1 else 0```
num1 = [5,9]
bool(num1) = True
num1.pop()
num1 = [5]
digit1 = 9
```digit2=num2.pop() if num2 else 0```
num2 = [1,3]
bool(num2) = True
num2.pop()
num2 = [1]
digit2 = 3
```

```
```total=digit1+digit2+carry```
total = digit1+digit2+carry =
9+3+0 = 12
```result.insert(0,total%10)```
result = []
total%10 = 12%10 = 2
result = [2]
```carry=total//10```
carry = 12//10 = 1
2.3 back to the start of the loop
...
3. Check Remaining Carry
```if carry: result.insert(0,carry)```
result = [7,2]
carry = 0
bool(carry) = False
pass
result = [7,2]
4. Return Result
```return result```
result = [7,2]
```

Figure 6. Examples of input-output sequence of question $59 + 13$ in 3 different settings, including direct answer, scratchpad and rule following. In the setting of rule-following, we provide the Python program of adding two numbers together digit by digit in the input, and provide the step-by-step rule-following process in the output. Examples of the full input-output pairs are shown in Appendix I.

5.1. Experimental Setup

In this section, we use two models, Llama-2-7B and GPT-3.5-turbo-1106. We fine-tune Llama-2-7B ourselves, and fine-tune GPT-3.5-turbo-1106 through the OpenAI API service. We focus on the length generalization problem of addition of two large numbers a and b, and put additional experiments on the task of concatenating last letters to Appendix F.7. We randomly sample a and b to construct the training data, where the numbers of digits of a and b range from 1 to 5, constituting about 500k samples in total for Llama-2-7B. When fine-tuning GPT-3.5, we reduce the training set to as small as 100 samples. We expect models with sufficient fundamental capabilities to be able to grasp rules through only a small set of training cases, which aligns with how humans learn calculations. During test, we randomly generate 1,500 samples for each digit length from 1 to 9 for Llama-2-7B, and generate 500 samples for each digit length from 6 to 15 for GPT-3.5. The digit length considers the context window size of each model. For GPT-3.5, due to the smaller training set, we perform five independent experiments and report the average accuracy and standard deviation. We employ direct answer, scratchpad, and RFFT as three fine-tuning methods for comparison. The training details are shown in Appendix F.1.
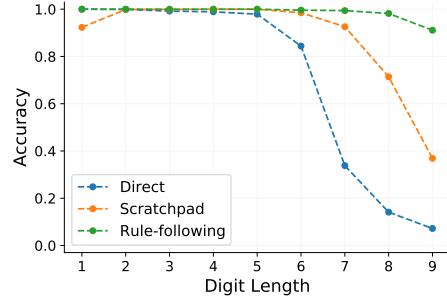
5.2. Results and Analysis

Overall Results The results are presented in Figure 7. Overall, *rule-following* significantly outperforms *direct* and *scratchpad*. When using Llama-2-7B with Rule-Following Fine-Tuning (RFFT), the model shows impressive generalization capabilities in performing addition with 6 to 9 digits, maintaining 91.1% accuracy even with 9-digit sums. In comparison, the scratchpad method achieves less than 40% accuracy in similar tasks. With GPT-3.5-turbo, which possesses more advanced foundational abilities, the RFFT method enables it to astonishingly generalize to additions involving up to 12 digits, with still over 95% accuracy on 12-digit addition despite seeing only 100 training samples. This significantly surpasses the results from the scratchpad and direct answer fine-tuning methods. These results highlight the effectiveness of our Rule-Following Fine-Tuning technique in steering transformers towards rule-based reasoning, showcasing its potential in enhancing model generalization. We provide detailed ablation studies in Appendix F.2.
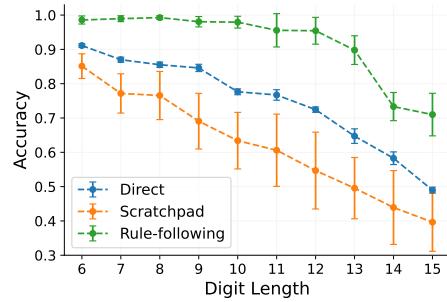
Error Analysis We also delve into failure cases to investigate why rule-following fails to achieve a perfect generalization with 100% accuracy. We find that the models can always select the right rule to execute in each step in a recursive way, but sometimes make mistakes when executing some basic operations, such as "pop". Consider the example "num2=[9,0,7,6,9,3,7]"; the expected output after "num2.pop()" should be "num2=[9,0,7,6,9,3]", while the models in some rare cases will generate "num2=[9,0,7,6,9]". As the length increases (e.g., more than 9-digit addition), the phenomenon becomes more severe, which could be attributed to hallucinations or the limited long context abilities of current LLMs (Li et al., 2023). As mentioned in Min et al. (2023), the tendency for hallucinations grows as the length of the generated content expands. These basic capabilities of LLMs might be the bottleneck that limits their strict length generalization under RFFT. It is also analogous to that we humans also tend to make sloppy mistakes when calculating long numbers by copying the wrong digits or forgetting to carry.

Comparison to Scratchpad Our RFFT technique provides the **explicit** rules in the input and also teaches LLMs to **quote** the part of rules used in each step, which helps

(a) Accuracy of Llama-7B fine-tuned with three methods tested on addition with 1-9 digits.

(b) Accuracy of GPT-3.5 fine-tuned with three methods tested on addition with 6-15 digits.

Figure 7. Accuracy of Llama-2-7B and GPT-3.5-turbo fine-tuned with direct answer, scratchpad and rule following on addition.

LLMs understand what each step is doing without having to refer to the long preceding texts. For example, with clear instructions "`total=digit1+digit2+carry`", an LLM knows it need to find and add these three variables together. In comparison, scratchpad requires LLMs to learn that the third number "0" in the formula "7+6+0=3" is the carry from last digit, increasing the difficulty of learning. Some example errors of RFFT and scratchpad are included in Appendix F.4. We also discuss RFFT's differences from scratchpad tracing in Appendix F.5.

RFFT as a Meta Learning Ability   As mentioned in §5.1, we find that Llama-2-7B requires 150k training samples to generalize to 9 digits while GPT-3.5 can grasp the rules and generalize to 12 digits with only 100 samples. Thus, we hypothesize that rule-following is a meta learning ability— it might be "learned" through pre-training on diverse rule-following data and transfer to new unseen domains, and the stronger the foundation model is, the easier it can understand and learn the rules. This also aligns with human's ability to learn new rules, where experienced learners often learn much faster. To provide more evidence, we further fine-tune a larger model Llama-2-70b than Llama-2-7b and a slightly weaker model davinci-002 than GPT-3.5. Our results show that stronger models indeed need less examples to learn

rules. See details in Appendix F.3.

Scratchpad vs Direct Answer   We observe that GPT-3.5, when fine-tuned with scratchpad, underperforms that with direct answer fine-tuning, which contradicts with our intuition that scratchpad is more suitable for arithmetic tasks as well as the results observed in Llama-2-7B. This phenomenon might be attributed to the different mechanisms of addition between scratchpad and direct answer. For example, scratchpad performs digit-by-digit addition from the lowest digit to the highest one, while direct answer always generates the highest digits first. Fine-tuning with scratchpad would strongly change the inherent addition mechanism of the model. At the same time, integer addition is in fact a relatively familiar task for GPT-3.5, wherein the model exhibits some degree of addition ability even when asked to directly generate the answer with an accuracy of 46.2% on 15-digit addition. This makes adopting scratchpad not always more helpful than direct answer fine-tuning. In contrast, RFFT explicitly interpret the step-by-step mechanism, making learning the addition rules much easier. To further support our hypothesis, we increase the number of training examples for scratchpad to 5,000 and observe much improved performance. See Appendix G for details.

5.3. In-context Learning

As we discussed in §4.4, LLMs encounter difficulties in autonomously *extracting* rules from ICL examples. The subsequent inquiry pertains to the capacity of LLMs to follow **explicit** rules supplied by in-context examples. Our conclusion is that given detailed rules, LLMs have certain abilities to follow the rules, which allows the models to show some reasoning ability on unfamiliar tasks. However, they do not gain a competitive edge from the rules in tasks already familiar to them. See Appendix C.3.

6. Conclusion

In our paper, we study whether transformers are performing "case-based reasoning" or "rule-based reasoning" when solving math problems. First, we describe the two reasoning paradigms and show how to distinguish one from the other. Then, we show through intervention experiments on five basic math tasks that transformers are relying on surrounding cases to do math reasoning. To mitigate the limitations of case-based reasoning, we propose a Rule-Following Fine-Tuning (RFFT) framework to teach transformers to perform rule-based reasoning by asking the model to explicitly quote and follow the rule used in each step. RFFT outperforms scratchpad fine-tuning by large margins, successfully enabling GPT-3.5-turbo fine-tuned on 1-5 digit addition to generalize to up to 12 digit addition.

Acknowledgements

Impact Statement

Our work provides a new perspective to understand how LLMs do math reasoning. The research for the first time defines and discriminates the two reasoning paradigms and proposes an effective method to steer transformer to perform rule-based reasoning, which is key to systematic generalization. Our work demonstrates that we can also directly teach rules to LLMs instead of just feeding data examples, just like how we teach children to perform addition.

References

Abbe, E., Bengio, S., Lotfi, A., and Rizk, K. Generalization on the unseen, logic reasoning and degree curriculum, 2023.

Akyürek, E., Schuurmans, D., Andreas, J., Ma, T., and Zhou, D. What learning algorithm is in-context learning? investigations with linear models, 2023.

Anil, C., Wu, Y., Andreassen, A., Lewkowycz, A., Misra, V., Ramasesh, V., Slone, A., Gur-Ari, G., Dyer, E., and Neyshabur, B. Exploring length generalization in large language models, 2022.

Bian, N., Han, X., Sun, L., Lin, H., Lu, Y., and He, B. Chatgpt is a knowledgeable but inexperienced solver: An investigation of commonsense problem in large language models, 2023.

Brown, T. B., Mann, B., Ryder, N., Subbiah, M., Kaplan, J., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., Agarwal, S., Herbert-Voss, A., Krueger, G., Henighan, T., Child, R., Ramesh, A., Ziegler, D. M., Wu, J., Winter, C., Hesse, C., Chen, M., Sigler, E., Litwin, M., Gray, S., Chess, B., Clark, J., Berner, C., McCandlish, S., Radford, A., Sutskever, I., and Amodei, D. Language models are few-shot learners, 2020.

Bubeck, S., Chandrasekaran, V., Eldan, R., Gehrke, J., Horvitz, E., Kamar, E., Lee, P., Lee, Y. T., Li, Y., Lundberg, S., Nori, H., Palangi, H., Ribeiro, M. T., and Zhang, Y. Sparks of artificial general intelligence: Early experiments with gpt-4, 2023.

Chowdhery, A., Narang, S., Devlin, J., Bosma, M., Mishra, G., Roberts, A., Barham, P., Chung, H. W., Sutton, C., Gehrmann, S., et al. Palm: Scaling language modeling with pathways. *Journal of Machine Learning Research*, 24(240):1–113, 2023.

Dai, D., Sun, Y., Dong, L., Hao, Y., Ma, S., Sui, Z., and Wei, F. Why can gpt learn in-context? language models implicitly perform gradient descent as meta-optimizers, 2023.

Dziri, N., Lu, X., Sclar, M., Li, X. L., Jiang, L., Lin, B. Y., West, P., Bhagavatula, C., Bras, R. L., Hwang, J. D., Sanyal, S., Welleck, S., Ren, X., Ettinger, A., Harchaoui, Z., and Choi, Y. Faith and fate: Limits of transformers on compositionality, 2023.

Feng, G., Zhang, B., Gu, Y., Ye, H., He, D., and Wang, L. Towards revealing the mystery behind chain of thought: A theoretical perspective, 2023.

Garg, S., Tsipras, D., Liang, P., and Valiant, G. What can transformers learn in-context? a case study of simple function classes, 2023.

Hou, Y., Li, J., Fei, Y., Stolfo, A., Zhou, W., Zeng, G., Bosselut, A., and Sachan, M. Towards a mechanistic interpretation of multi-step reasoning capabilities of language models, 2023.

Kazemnejad, A., Padhi, I., Ramamurthy, K. N., Das, P., and Reddy, S. The impact of positional encoding on length generalization in transformers, 2023.

Khot, T., Trivedi, H., Finlayson, M., Fu, Y., Richardson, K., Clark, P., and Sabharwal, A. Decomposed prompting: A modular approach for solving complex tasks, 2023.

Kojima, T., Gu, S. S., Reid, M., Matsuo, Y., and Iwasawa, Y. Large language models are zero-shot reasoners, 2023.

Koralus, P. and Wang-Maścianica, V. Humans in humans out: On gpt converging toward common sense in both success and failure, 2023.

Lee, N., Sreenivasan, K., Lee, J. D., Lee, K., and Papailiopoulos, D. Teaching arithmetic to small transformers, 2023.

Li, J., Wang, M., Zheng, Z., and Zhang, M. Loogle: Can long-context language models understand long contexts? 2023.

Liu, Z., Kitouni, O., Nolte, N., Michaud, E. J., Tegmark, M., and Williams, M. Towards understanding grokking: An effective theory of representation learning, 2022.

Min, S., Krishna, K., Lyu, X., Lewis, M., Yih, W.-t., Koh, P. W., Iyyer, M., Zettlemoyer, L., and Hajishirzi, H. Factscore: Fine-grained atomic evaluation of factual precision in long form text generation. *arXiv preprint arXiv:2305.14251*, 2023.

Nanda, N., Chan, L., Lieberum, T., Smith, J., and Steinhardt, J. Progress measures for grokking via mechanistic interpretability, 2023.

Nye, M., Andreassen, A. J., Gur-Ari, G., Michalewski, H., Austin, J., Bieber, D., Dohan, D., Lewkowycz, A., Bosma, M., Luan, D., Sutton, C., and Odena, A. Show your work: Scratchpads for intermediate computation with language models, 2021.

OpenAI. Introducing chatgpt, 2022. https://openai.com/blog/chatgpt.

OpenAI. Gpt-4 technical report, 2023.

Power, A., Burda, Y., Edwards, H., Babuschkin, I., and Misra, V. Grokking: Generalization beyond overfitting on small algorithmic datasets, 2022.

Qian, J., Wang, H., Li, Z., Li, S., and Yan, X. Limitations of language models in arithmetic and symbolic induction, 2022.

Qin, C., Zhang, A., Zhang, Z., Chen, J., Yasunaga, M., and Yang, D. Is chatgpt a general-purpose natural language processing task solver?, 2023.

Radford, A., Wu, J., Child, R., Luan, D., Amodei, D., and Sutskever, I. Language models are unsupervised multitask learners. 2019. URL https://api.semanticscholar.org/CorpusID:160025533.

Shen, R., Bubeck, S., Eldan, R., Lee, Y. T., Li, Y., and Zhang, Y. Positional description matters for transformers arithmetic, 2023.

Srivastava, A., Rastogi, A., Rao, A., Shoeb, A. A. M., Abid, A., Fisch, A., Brown, A. R., Santoro, A., Gupta, A., Garriga-Alonso, A., et al. Beyond the imitation game: Quantifying and extrapolating the capabilities of language models. *arXiv preprint arXiv:2206.04615*, 2022.

Thoppilan, R., De Freitas, D., Hall, J., Shazeer, N., Kulshreshtha, A., Cheng, H.-T., Jin, A., Bos, T., Baker, L., Du, Y., et al. Lamda: Language models for dialog applications. *arXiv preprint arXiv:2201.08239*, 2022.

Touvron, H., Lavril, T., Izacard, G., Martinet, X., Lachaux, M.-A., Lacroix, T., Rozière, B., Goyal, N., Hambro, E., Azhar, F., Rodriguez, A., Joulin, A., Grave, E., and Lample, G. Llama: Open and efficient foundation language models, 2023a.

Touvron, H., Martin, L., Stone, K., Albert, P., Almahairi, A., Babaei, Y., Bashlykov, N., Batra, S., Bhargava, P., Bhosale, S., et al. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288*, 2023b.

von Oswald, J., Niklasson, E., Randazzo, E., Sacramento, J., Mordvintsev, A., Zhmoginov, A., and Vladymyrov, M. Transformers learn in-context by gradient descent, 2023.

Wei, J., Tay, Y., Bommasani, R., Raffel, C., Zoph, B., Borgeaud, S., Yogatama, D., Bosma, M., Zhou, D., Metzler, D., Chi, E. H., Hashimoto, T., Vinyals, O., Liang, P., Dean, J., and Fedus, W. Emergent abilities of large language models, 2022.

Wei, J., Wang, X., Schuurmans, D., Bosma, M., Ichter, B., Xia, F., Chi, E., Le, Q., and Zhou, D. Chain-of-thought prompting elicits reasoning in large language models, 2023.

Wu, Z., Qiu, L., Ross, A., Akyürek, E., Chen, B., Wang, B., Kim, N., Andreas, J., and Kim, Y. Reasoning or reciting? exploring the capabilities and limitations of language models through counterfactual tasks, 2023.

Xu, X., Pan, Z., Zhang, H., and Yang, Y. It ain't that bad: Understanding the mysterious performance drop in ood generalization for generative transformer models, 2023.

Yang, H., Meng, F., Lin, Z., and Zhang, M. Explaining the complex task reasoning of large language models with template-content structure, 2023.

Yun, C., Bhojanapalli, S., Rawat, A. S., Reddi, S. J., and Kumar, S. Are transformers universal approximators of sequence-to-sequence functions?, 2020.

Zhang, C., Ippolito, D., Lee, K., Jagielski, M., Tramèr, F., and Carlini, N. Counterfactual memorization in neural language models, 2023.

Zhong, Z., Liu, Z., Tegmark, M., and Andreas, J. The clock and the pizza: Two stories in mechanistic explanation of neural networks, 2023.

Zhou, D., Schärli, N., Hou, L., Wei, J., Scales, N., Wang, X., Schuurmans, D., Cui, C., Bousquet, O., Le, Q., and Chi, E. Least-to-most prompting enables complex reasoning in large language models, 2023a.

Zhou, H., Nova, A., Larochelle, H., Courville, A., Neyshabur, B., and Sedghi, H. Teaching algorithmic reasoning via in-context learning, 2022.

Zhou, H., Bradley, A., Littwin, E., Razin, N., Saremi, O., Susskind, J., Bengio, S., and Nakkiran, P. What algorithms can transformers learn? a study in length generalization, 2023b.

Zhou, Y., Alon, U., Chen, X., Wang, X., Agarwal, R., and Zhou, D. Transformers can achieve length generalization but not robustly, 2024.

Zhu, Z., Xue, Y., Chen, X., Zhou, D., Tang, J., Schuurmans, D., and Dai, H. Large language models can learn rules, 2023.

A. Limitations

Rule-following fine-tuning (RFFT) is temporarily a task-specific method and requires carefully designed input-output sequences. Besides, we mainly focus on fine-tuning models for specific tasks instead of exploring the reasoning mechanism of pretrained models. There also remain further questions to be explored, for example, how various prompting techniques, such as CoT, least-to-most prompting, and program-aided prompting, affect the model behavior of case-based reasoning or rule-based reasoning. These questions are left for future work.

B. Collections of Hyper-parameters

We list all the hyper-parameters used in the paper in Table 1.

Models	training epoch	batch size	learning rate
case-based reasoning			
GPT-2	100	30	1×10^{-4}
Llama-2-7B	4	4	2×10^{-5}
rule-following fine-tuning			
GPT-3.5	4	4	OpenAI API default value
Llama-2-7B	1	8	2×10^{-5}

Table 1. Hyper-parameters

C. In-context Learning

C.1. Case-based Reasoning in ICL

We have discussed two reasoning mechanism of case-based reasoning and rule-based reasoning in experiments of fine-tuning LLMs. However, another crucial aspect of LLMs' reasoning ability is attributed to in-context learning (ICL). This method draws upon knowledge not only ingrained during the pre-training but also from specific examples supplied within the context. The underlying mechanisms that make ICL effective are among the most intriguing and unanswered questions in the field. In this section, we extend our investigations to ICL, revealing that LLMs' ICL reasoning ability also exhibits characteristics of case-based learning.

Because ICL is an emergent ability (Brown et al., 2020), we choose a stronger model: GPT-3.5-turbo-0125. We use the *base addition* task where we randomly add two base-9 integers with 3 digits. The adopted GPT-3.5 can rarely solve the task only with a task description, making sure that the investigated reasoning power comes from ICL. See the zero-shot results in Appendix C.2.

To study whether ICL reasoning relies on rules or similar cases in the context, we randomly collected pairs of base-9 integers whose zero-shot addition accuracy is less than 20%. Then, we provide 10 few-shot examples with the correct answers for each pair of integers, five of which are *randomly* selected (called random group), and another five are obtained by simultaneously replacing only one digit of the pair (thus are considered as more similar examples than the first five, and called similar group). Scratchpad is used in each few-shot example to provide step-by-step intermediate results. We choose the 14 test samples where the improvement with few-shot examples is more than 80%.

To determine the contribution of each example, we adopt an intervention experiment similar to §4.2 where we mask some in-context examples from either the similar group or the random group and compare the accuracy drop. Considering the interaction between individual examples, we choose to traverse all mask possibilities within a group instead of masking only one example. For example, for the similar group, we will have $2^5 - 1 = 31$ possible masks (excluding the empty mask). Specifically, we measure the contribution of the i-th in-context example as

$$c_i = \left(\sum_{m \in \mathcal{M}} \mathbb{1}\{i \in m\} \cdot \frac{accu_m - accu_{orig}}{accu_{icl} - accu_{orig}} \right) \Big/ N_i \,,$$

where $accu_{orig}$, $accu_{icl}$ and $accu_m$ represent the accuracy without few-shot examples, with all examples and with non-masked examples, respectively. $\mathcal{M}$ is the mask set that contains all possible combinations in the random and similar group.
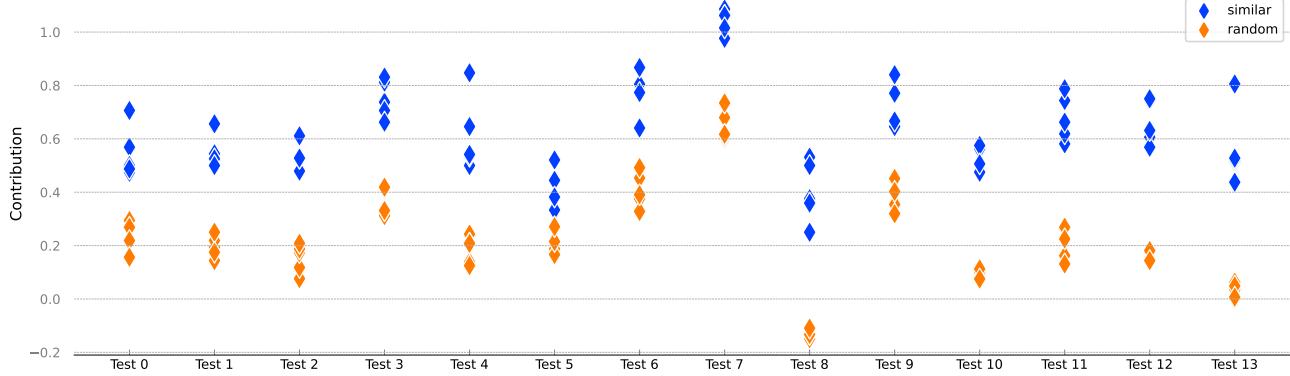
Figure 8. The contribution of *similar* and *random* ICL examples in 14 tests. For contribution in each experiment, the contribution of similar examples is significantly larger than that of random ones.

N_i is the number of masks that contain i (which is a constant 16). We report the contribution of similar and random examples to the 14 test samples in Figure 8. The contribution of the similar group is significantly greater than that of the random group in all experiments, with the p-value of 14 average values < 0.001.

These results suggest that the model relies more on directly discovering shortcuts from similar cases rather than summarizing the reasoning rules of the task. This phenomenon seems contrary to some previous views on ICL which point out that the contribution of in-context examples lies mainly on hints about "tasks" and "domains" rather than specific functions, implying a more rule-based method. We believe the difference comes from the basic capacity of LLMs to solve the task. For some tasks where LLMs have captured the essential reasoning abilities, ICL examples may help them "recall" the task so that the model can benefit from even some dissimilar examples. In contrast, when the model is unfamiliar with the task, it is difficult to solve the problem through recalling the pre-training knowledge. In this case, only similar examples can improve model performance by providing more direct shortcuts. In a word, our experiments suggest that it may not be possible to expect the model to extract rules that were not obtained during the pre-training phase by summarizing ICL examples.

C.2. Zero-shot Results

Base addition is an "unfamiliar" task that the model cannot solve without few shot examples. To show it, we test GPT-3.5-turbo-0125 with 100 pairs of base 9 integers with 3 digits. We use the system message as "You are a helpful assistant to solve arithmetic problems. You will be provided with two base 9 integers and you need to return the sum of the two integers in base 9." the user message as " Int a: a; Int b: b." The model can happen to generate the correct answer by summing two numbers in base 10. For example, 236+321, which is equal to 557 in either base 10 or base 9. So we also test the model on the test set where these easy samples are removed. The results is shown in Table 2.

Task	0-shot base addition	0-shot base addition (hard)
Accuracy	$8.8\% \pm 10.5\%$	$8.0\% \pm 10.0\%$

Table 2. Zero-shot accuracy of GPT-3.5-turbo on base addition

C.3. Rule Following Ability from In-context learning

C.3.1. ADDITION

We first conduct experiments on a the standard base-10 addition, which is familiar to GPT-3.5. Utilizing the GPT-3.5-turbo-0125 model, for the maximal digit length among these two integers from 1 to 10, we randomly selected 100 pairs of integers respectively. Each test pairs repeat 5 computations to obtain the average accuracy. We provided identical in-context examples for all inputs, consisting of 5 examples with a maximum length of 5 digits. These in-context examples are presented in *direct*, *scratchpad*, and *rule-following* formats. The accuracy of each digit under these three formats is illustrated in Figure 9 left. Rule-following prompting lags behind directly asking the model to generate the answer. This may be because the model

may find shortcuts to do addition as it has been trained on a huge corpus containing various addition calculations.
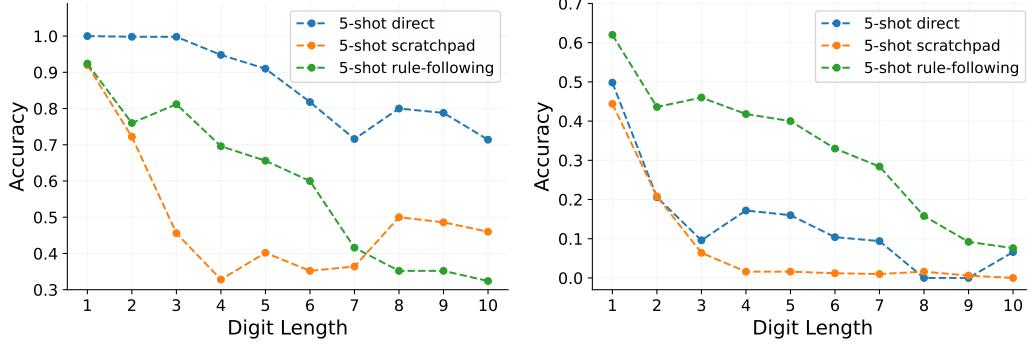
Figure 9. In-context learning performance on **addition** task (**left**) and **base addition** task (**right**).

C.3.2. BASE ADDITION

Then, we test the performance on the base-9 addition task. Here we provides 5 examples with maximal digits 5 in direct answer, scratchpad and rule-following format, shown in Figure 9 right. On this task, rule-following still shows good performance, but the performance of direct and scratchpad is greatly reduced, compared to addition task. This shows that in both direct and scratchpad prompt modes, the model still relies heavily on its basic capabilities. Therefore, the rule following method is particularly suitable for complex and unfamiliar tasks. This kind of detailed and clear rule guidance helps the model quickly master a certain degree of reasoning with little knowledge of the corresponding task, but for tasks where the model has learned some shortcuts, it may not help performance. The shortcut learned in the pre-training stage cannot help each other with the rules in ICL. Therefore, on the latter task, if you want the model to follow the rules for reasoning, finetuning is necessary.

D. Additional Results of Leave-Square-Out

Chicken & rabbit problem We show the results of leaving test squares out of the datasets chicken and rabbit problem in Figure 10. We experiment on two models including GPT-2 and GPT-2 Medium. The center and the length of the test square in the experiment of leaving 1 square out is $(70, 50)$, $l_k = 20$. The lengths of the test squares in the experiment of 3 holes are randomly sampled in $[10, 30]$.

(a) 1 hole GPT-2   (b) 1 hole GPT-2 Medium   (c) 3 holes GPT-2   (d) 3holes GPT-2 Medium
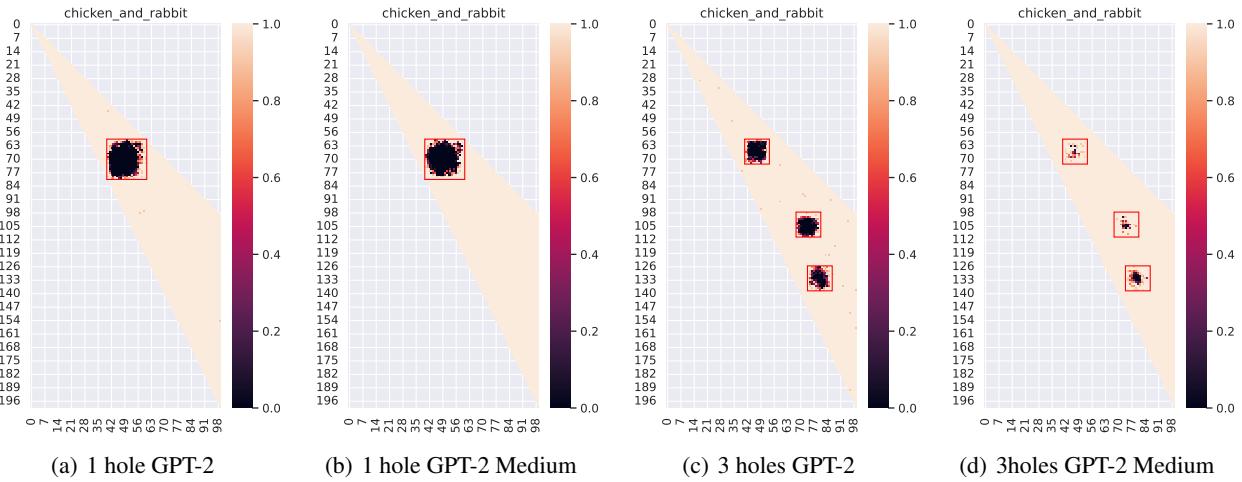
Figure 10. Accuracy distributions of GPT-2 and GPT-2 Medium on chicken & rabbit problem.

GPT-2 medium We show the results of performing Leave-Square-Out on GPT-2 Medium on datasets including addition, modular addition, base addition and linear regression in Figure 11 (leaving 1 square out) and Figure 12 (leaving 3 square out). Besides, we show the results of leaving a square out on GPT-2 Medium trained on input-output pairs containing scratchpads in Figure 13.
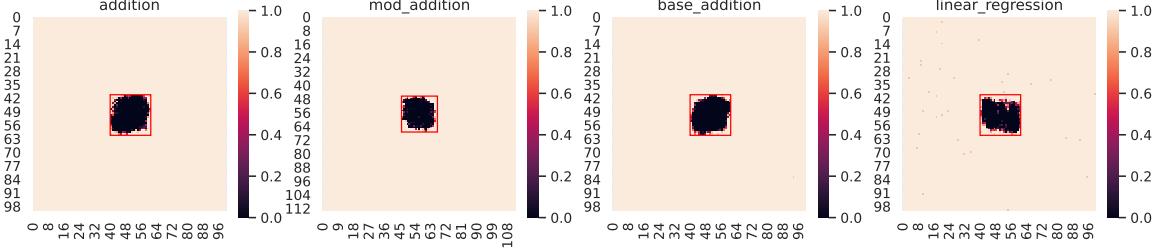
Figure 11. Accuracy of leaving a test square of length $l_k = 20$ out on GPT-2 Medium.
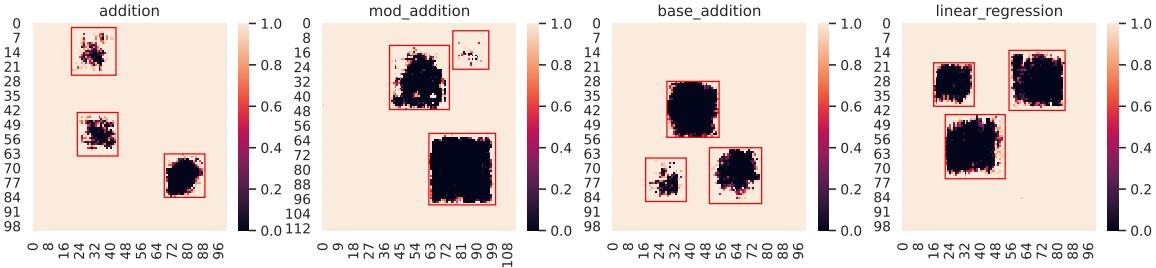
Figure 12. Accuracy of leaving 3 test squares out on GPT-2 Medium.
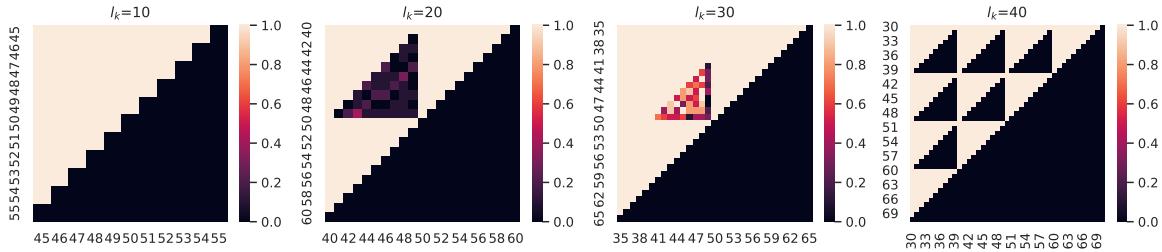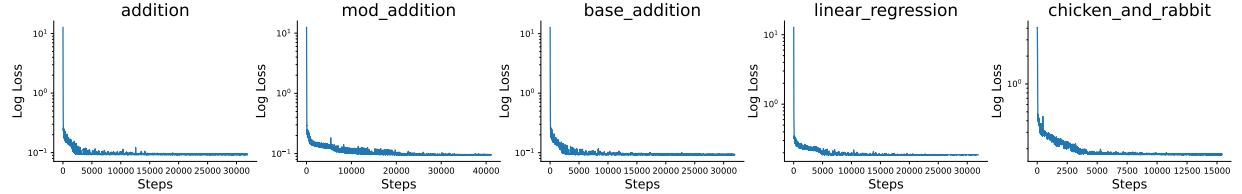
Figure 13. Test accuracy of GPT-2 Medium trained on input-output pairs containing scratchpads on the task of addition.

Training curve We show the training loss of GPT-2 and GPT-2 medium in the Leave-Square-Out experiments in Figure 14.
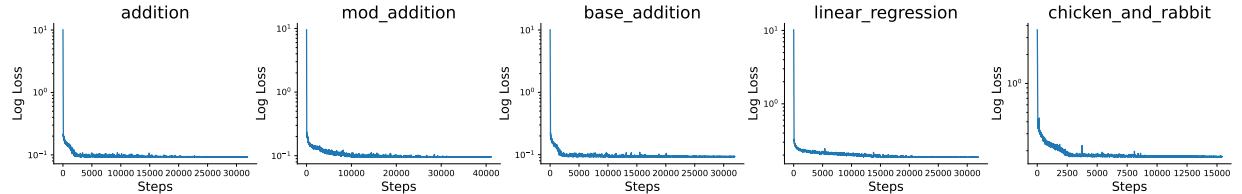
Besides, to have a clearer look into the training process, we conduct Leave-Square-Out experiments by fine-tuning GPT-2 on the addition task for 1, 2, 3,..., 10, 20, 30,..., 100 epochs, respectively. The results are in Figure 15. The center of the test square (a_k, b_k) is set to $(50, 50)$, and the length l_k is 20. During generation, we set the model temperature to 1 and sample 10 generations to evaluate the accuracy on each test point.

The results show that after the model's training loss is lower than a certain value (after epoch 4), the model exhibits obvious case-based reasoning behavior (holes appear in the test square). In the earlier epochs like epoch 1 and 2, the training has not saturated, thus both training and test accuracy are extremely low, which also indicates the necessity of fine-tuning for such tasks.

GPT-2 trained from scratch We additionally experiment on GPT-2 trained from scratch on the task of base addition to see how the process of pre-training affects the reasoning mechanism. We show the results of leaving one test square

(a) Training loss of GPT-2.

(b) Training loss of GPT-2 Medium.

Figure 14. Log training loss of GPT-2 and GPT-2 medium in the Leave-Square-Out experiments.
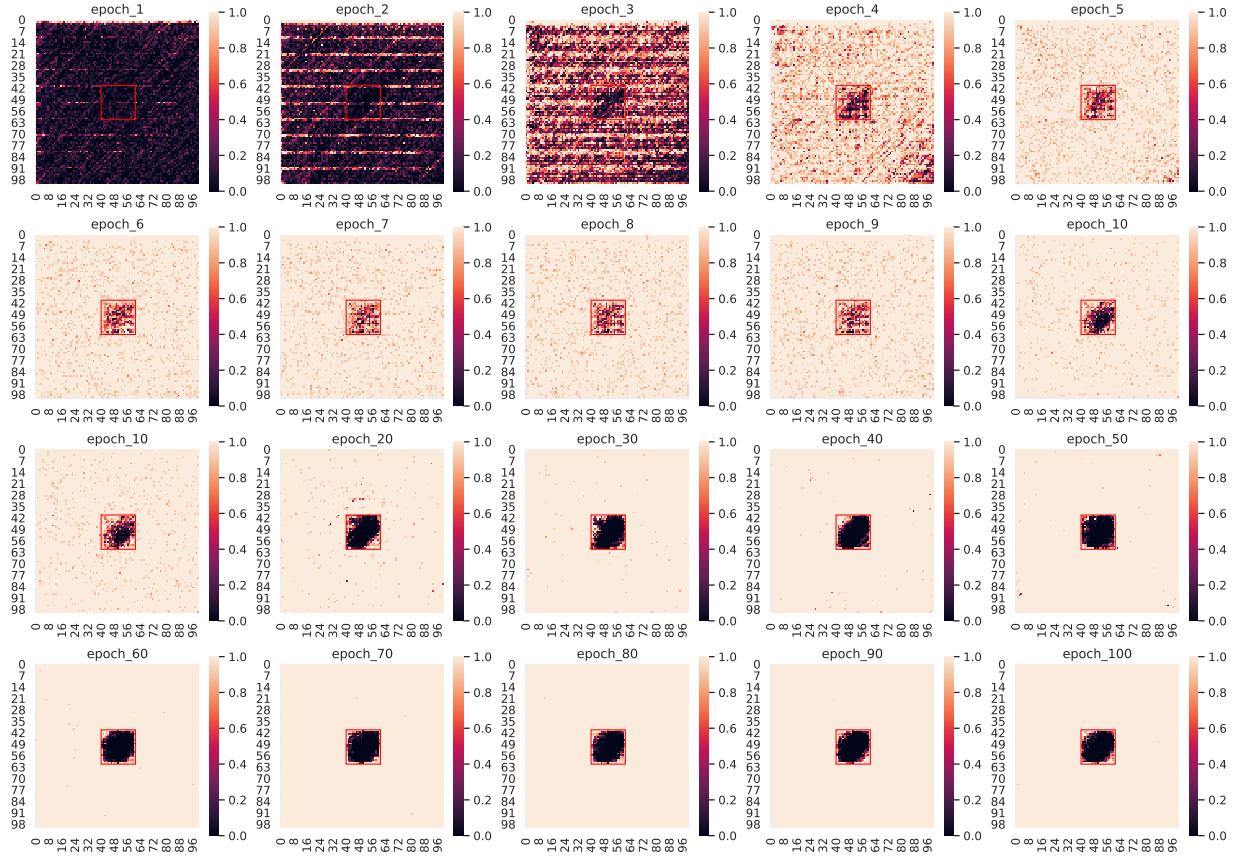
Figure 15. Model performance of GPT-2 fine-tuned after different num of epochs on the whole addition dataset. The area inside the red box represents the test square.

with length $l_k = 20$ and center $(50, 50)$ out in Figure 16. Besides, we conduct the experiments of training the model in the random-split setting with training set accounting for 70% of the whole dataset. The model can achieve more than 98% accuracy on the test set in the random-split setting. As shown in Figure 16, there is a black hole in the test square, suggesting the behavior of case-based reasoning is still obvious when we directly train the model from scratch. Besides, we observe in the training process that the training loss converges much more slowly than in the setting of fine-tuning a pre-trained model.
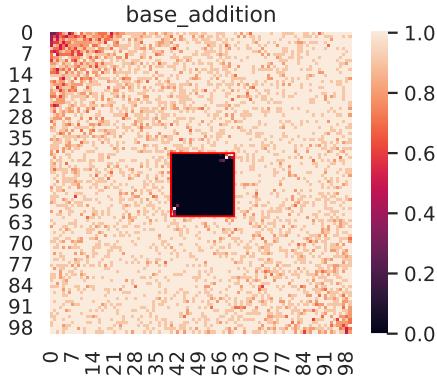
Figure 16. Accuracy of GPT-2 trained from scratch on the task of base addition when we leave a test square out. The center and the length of the test square is $(50, 50)$, $l_k = 20$

Considerations for class imbalance As some may worry that the method of leaving a test square out may cause data imbalance, thus confusing the model, we conduct an additional experiment to study the effect of class imbalance issue by upsampling those numbers that originally occur less in the training set of the addition task, thereby maintaining a balanced distribution of numbers and digits in the new training set. In Figure 17, we show the digit frequency before and after upsampling. It shows that both the numbers and digits are balanced after upsampling.

Then, we fine-tune the model on this updated training set and repeat the experiment of Figure 2. The new results are illustrated in Figure 18. As we can see, the hole still appears, demonstrating the behavior of case-based reasoning. This indicates that class imbalance is not a confounder of our results.

Besides, we also show the frequency of number a in the original training set in Figure 17. It should be noticed that our original training set is not a dataset with extreme data imbalance, as we only leave 20 out of 100 samples of certain numbers.

E. Ablations for Leave-Square-Out

E.1. Ablation for Data Size

To explore the effects of datasize on the model behavior, we conduct experiments of Leave-Square-Out on the task of addition of different range of a, b, including $[0, 100), [0, 200), [0, 500)$. Correspondingly, we scale up the length of test square to be $l_k = 20, 40, 100$ respectively. We use GPT-2 and use the same hyper-parameters in each dataset. We train the model with 100 epochs, batch size set to 30 and learning rate set to 10^{-4}. The results are shown in Figure 19. Holes can still be observed in the setting where $a, b \in [0, 500)$ (the dataset scales up 25 times), suggesting that the models are still doing case-based reasoning when the data size scales up.
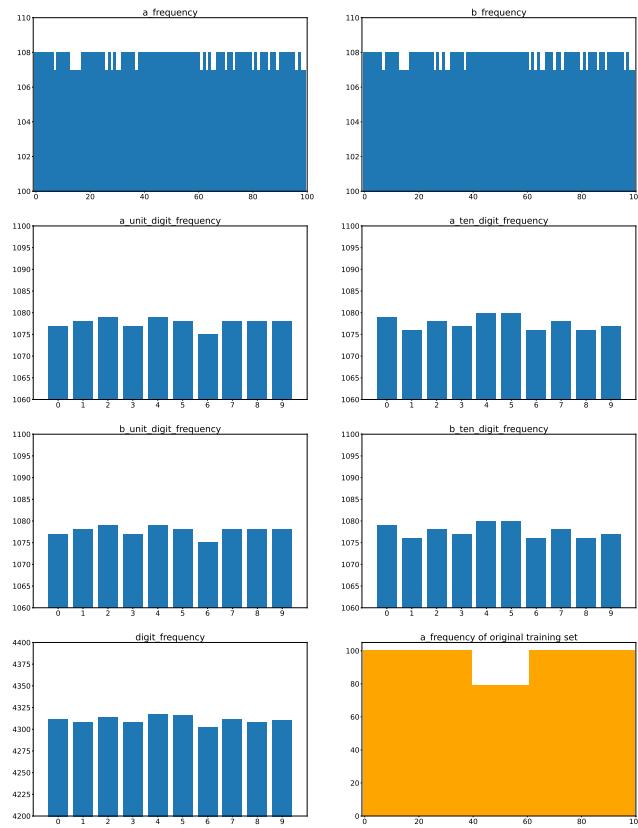
Figure 17. The frequency of numbers a and b, the frequency of unit digits and tens digits of a and b, and the frequency of digits in both a and b. We show the digit frequency after upsampling with blue histograms and that before upsampling with orange histograms.
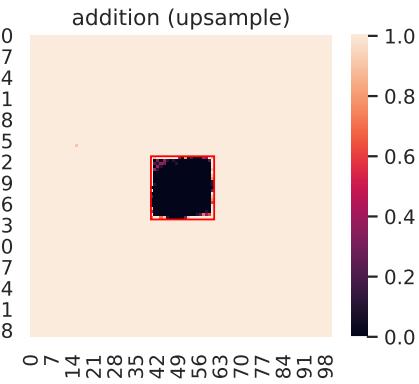
Figure 18. Performance of fine-tuned GPT-2 on addition after upsampling.
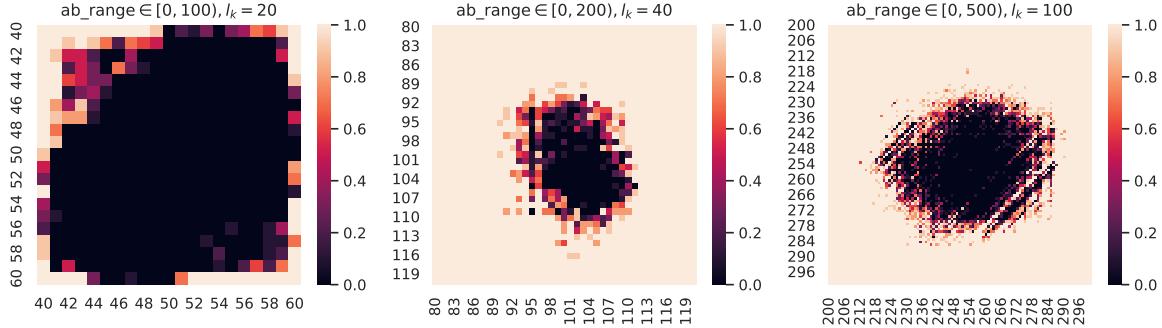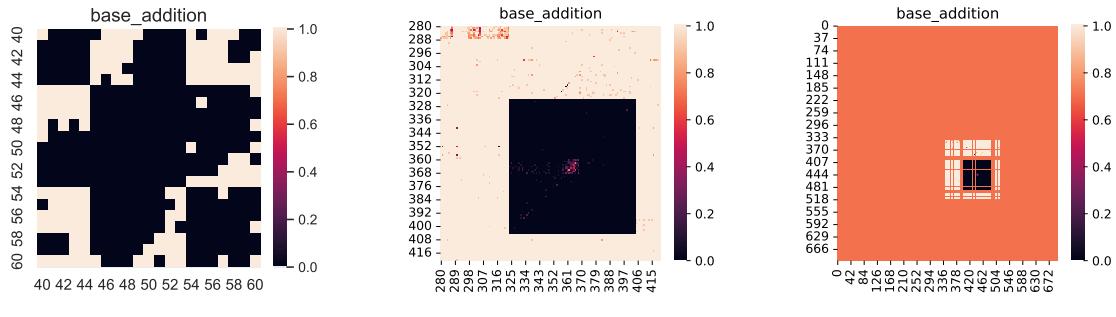
Figure 19. Test accuracy of models trained on addition of different data size.

E.2. Ablation for Model Size: Llama-2-7B

To show the effects of model size on case-based reasoning, we conduct experiments on Llama-2-7B on the task of base addition because Llama-2 has already learned addition to a some degree. To further eliminate the effect of pre-training, we train the model *from scratch* instead of fine-tuning.

We maintain the same data size as GPT-2, i.e., setting the range of a and b to $[0, 100)$ and leave out a test square with center $(50, 50)$ and side length $l_k = 20$. Despite this, the test accuracy in the Leave-Square-Out setting is 30.2%, far lower than the random split accuracy. As the comparison, the test accuracy in the random split setting reaches 92%. There are still holes in the accuracy distribution of the test squares, as shown in Figure 20(a).

Note that different from GPT-2, the test accuracy in the random split setting cannot reach 100% even after training for 500 epochs where training loss has almost converged, suggesting overfitting. In light of this, we also conduct an experiment where we correspondingly enlarge the range of a and b to $[0, 700)$ with the side length of test square $l_k = 140$ and center $(350, 350)$, forming a training set accounting for about 96% of the whole dataset. We first conducted the experiment in the random-split setting with 70%-30% training-test ratio and verified the model can reach 100% accuracy. Then we perform the Leave-Square-Out experiment. Figure 20(b) shows the results. The model still demonstrates significant case-based reasoning behavior by failing to answer a large portion of test samples. This indicates that the trend of case-based reasoning still exists when the model scales up. Furthermore, we also plot it with base-9 coordinates in Figure 20(c), which shows a highly structured pattern possibly related to the task structure. Here, we provide a preliminary analysis in Appendix E.2, leaving a more in-depth exploration for future work.

(a) Accuracy in the **test square** of Llama-2-7B on the task of base addition. Center $(50, 50)$, length $l_k = 20$.

(b) Test accuracy distribution of Llama-2-7B on the task of base addition over a and b when we leave a test square of side length $l_k = 140$ and center $(350, 350)$.

(c) Accuracy of Llama-2 trained from scratch on the task of base addition when we leave a test square out. The center and the length of the test square is $(428, 428)$ (represented in base-9), $l_k = 140$

Figure 20. Accuracy of Llama-2-7B **trained** on base addition **from scratch**.

Digit Length	4-digit	5-digit	6-digit
Accuracy	91.88%	74.14%	51.46%

Table 3. Performance of Fine-tuned Llama-2-7b on OOD samples of base addition. We test 5,000 samples.

(a) Accuracy of test square in the Leave-Square-Out experiment.

(b) Accuracy of the whole dataset in the random split experiment.
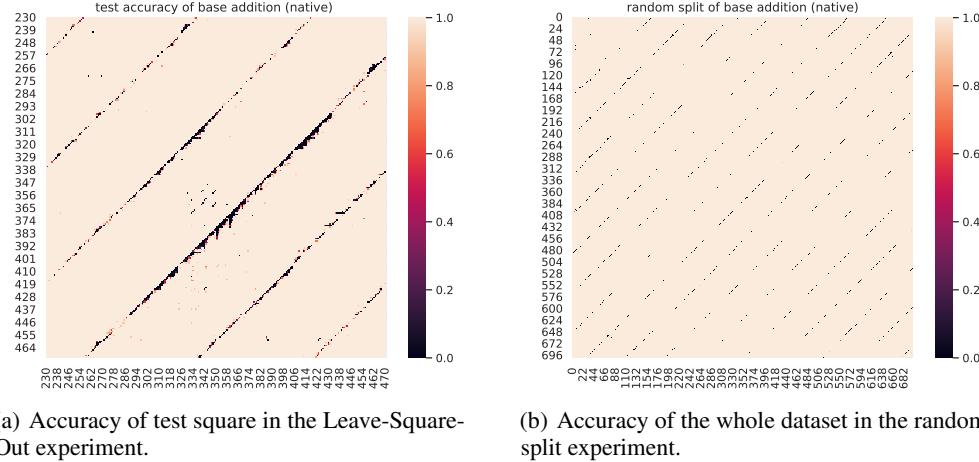
Figure 21. Performance of Llama-2-7B **fine-tuned** on base-9 addition.

Fine-tuning or train from scratch?   Also, we conduct experiments of fine-tuning pretrained Llama-2-7B on base-9 addition. The results are shown in Figure 21.

Firstly, we observe that the results are indeed different from training from scratch. There are no obvious holes in the test square. Instead, model performance drops in areas of anti-diagonals of both training and test regions.

Admittedly, the results do not provide evidence for case-based reasoning, however, it **does not necessarily indicate that the model is performing rule-based reasoning** either. The reasons are as follows:

1. It is possible that the left-out square are **not really the dependent cases** for pretrained Llama-2-7B on this task. Our hypothesis that surrounding cases are the dependent cases may not hold for this setting. It is possible that other training/test spliting method can reveal case-based reasoning behavior again.

2. There might be **data leakage** during pretraining. Without considering data leakage, pretraining may still have introduced strong biases that happen to suit base addition well, making the model generalize to most parts of the test square. It should be noticed that introducing biases is essentially different from enhancing the model's foundational reasoning abilities or equipping it with the ability to perform rule-based reasoning after fine-tuning, because the biases may only suit some specific tasks or representations, rather than uniformly helping models to learn rules for different tasks/representations (as will be discussed in the following base-9 addition with exotic digits experiment).

3. We test the fine-tuned model on OOD samples involving 4/5/6 digits. The results are listed in Table 3. The results indicate that the model at least **does not learn rules that can generalize across different lengths**. In other words, the model might learn some shortcuts working on same-length samples, but fail to learn the most faithful base addition rules that allows for length generalization.

To further investigate the reasoning mechanism of pretrained Llama-2-7B and mitigate the risk of data leakage, we alter the representations of base-9 numbers and fine-tune Llama-2-7B on the new and more challenging task. Specifically, we **replace the digits 0-8 with letters A-I** to create a counterfactual dataset which has little chance to have been exposed to the pretrained model. For example, we replace "1+8=10" with "B+I=BA". We call the new task with replaced digits base-9 addition with "exotic digits".
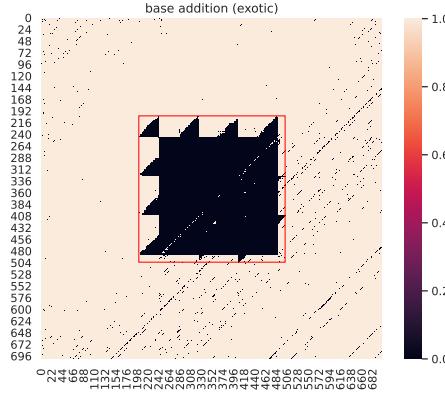
Figure 22. Performance of **fine-tuned** Llama-2-7B on base-9 addition with exotic digits. The area inside the red box represents the test square.

For Llama trained from scratch, there should be nearly no distinction between these two representations if we ignore tokenization differences. For fine-tuned Llama, if it relies on a systematic mechanism to induce rules from examples, it should be able to learn these rules regardless of whether native digits or exotic digits (i.e. A-I letters) are used. However, the test performance and accuracy distribution reveal significant discrepancies from the native-digit setting, despite the training still achieves close to 100% accuracy. We show the new accuracy distribution in Figure 22.

As we can see, after using letter representations for base-9 numbers, there is a hole in the test square, demonstrating case-based reasoning behavior. This might indicate that the large-scale pretraining does not equip models with systematic and general rule-learning abilities. Instead, it is more likely that pretraining introduces biases suitable only for certain tasks and representations, enabling certain degree of generalization.

In conclusion, we first fine-tune pretrained Llama on regular base addition and the model performance drops in anti-diagonal areas but shows no holes, demonstrating different patterns from training from scratch. However, the phenomenon does not necessarily indicate that the model is performing rule-based reasoning. To dig deeper, we change the representations of digits into exotic letters and fine-tune Llama on the new task, which shows clear evidence of case-based reasoning. This suggests that the reasoning behavior of LLMs can be highly dependent on the input representations. Besides, large-scale pretraining seems not equip the model with the ability of systematic rule learning that can adapt to various representations.

Error analysis of Llama-2-7B on base addition In the experiment of training larger model Llama-2-7B on base-9 addition task as described in Appendix E.2, we find that there is still a hole in the test set, indicating that even if the models scale up, they struggle to learn to perform rule-based reasoning. Furthermore, we conduct a more detailed analysis. We observed the model usually generates wrong answers when both "a" and 'b" input values had hundreds of digits of "4". For instance, the model can correctly output "400 + 388 = 788", however, it failed when presented with "400 + 400", generating the output as "500". It appears to draw from the "closeness" between 400 and 388, failing to grasp the difference of 1 in base-9 as opposed to a difference of 12 in base-10, resulting in an erroneous output. Moreover, for the sequences "400+401 =", "400+402 =", and "400+403 =", the model output "501", "502", "503" respectively. These findings suggest that the model relies heavily on the context of closely related cases for its calculations rather than utilizing rule-based reasoning.

E.3. Ablation for Model Size: GPT-3.5-turbo

To verify whether the conclusions are consistent for larger models, we conduct additional experiments on GPT-3.5-turbo. To be more specific, we choose the task of base-9 addition (less likely to appear in pre-training corpus than addition) and leave a test square with center $(a_k, b_k) = (350, 350)$ and length $l_k = 300$ (accounting for 18.5% of the whole dataset) out of the whole dataset with $a \in [0, 700], b \in [0, 700]$. We fine-tune GPT-3.5 for 1 epoch with batch size set to 80.

The training and test accuracy distribution of the fine-tuned model is shown in Figure 23. Due to the limited budget, we randomly sample 20% datapoints out of the test set and 10% datapoints out of the training set to do inference. For each sample, we perform single generation with model temperature set to 0. As we can see from the figure, there is still a "hole" in the test square, demonstrating case-based reasoning behavior.
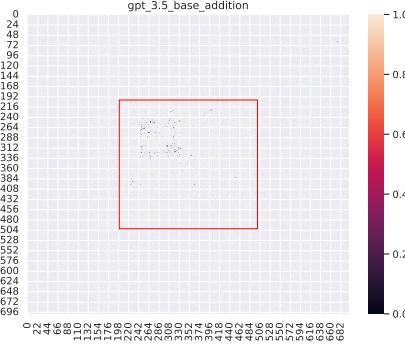
Figure 23. The training and test accuracy distribution of fine-tuned GPT-3.5-turbo on base-9 addition. The area inside the red box represents the test square. The figure needs to be scaled up to see the light pink samples that are correctly answered.

Training Accuracy (square)	Test Accuracy (square)	Test Accuracy (random)
0.9999	0.9751	0.9998

Table 4. The left two columns shows training and test accuracy of GPT-3.5 fine-tuned in the Leave-Square-Out experiments. The rightmost column shows test accuracy of the model fine-tuned in the random-split setting.

More specifically, we list train/test accuracy in the Leave-Square-Out experiment and test accuracy in the corresponding random-split setting in Table 4. In the random-split experiment, the training set accounts for 70% of the dataset, while in the Leave-Square-Out experiment, the training set accounts for 81.5% of the whole dataset. As can be seen in the table, although test accuracy (square) is high (∼97.5%), there are still gaps from training accuracy (square) (∼100%) and test accuracy (random) (∼100%). In other words, despite using more training data in the leave-square-out experiment, the test accuracy cannot saturate like that in the random-split experiment. In Figure 23, we show the samples where the model generates wrong answers with black points (corresponding to the accuracy gap) to highlight the "hole" area where the model's performance drops. This indicates that there is still a small area where the model is relatively easy to fail, and the ability to perfectly solve the test cases in this specific area relies on training on the test square. This exactly implies case-based reasoning. Nevertheless, the hole is not as large as those in the experiments of smaller models, indicating that stronger LLMs might have better capability to leverage longer-dependency cases so that most cases in the test square may still find some dependent cases in the training set.

F. Training Details and Additional Results of Rule-following Finetuning

F.1. Training Details of Rule-following Finetuning

When fine-tuning Llama, we use the same hyper-parameters for each of three different settings, excluding the model_max_length varied to 64, 512, and 4096 for direct, scratchpad and rule-following. To be more specific, we train the models with 1 epoch, 8 batch sizes, a dropout rate of 0.05 and a learning rate of 2e-5, saving the best model at the last step. We use greedy decoding in the inference stage to generate output with a temperature of 0.

When finetuning GPT-3.5-turbo, we use a training set of 100 samples and train the models with OpenAI API with 4 epochs, 4 batch sizes.

Model	6 digit	7 digit	8 digit	9-digit
Llama-2-7b	99.6	99.4	96.2	90.1
Llama-2-70b	99.3	97.8	97.0	89.8

Table 7. Accuracy of fine-tuned Llama-2-7b and Llama-2-70b on 6-9 digit addition task

input:	687042+86208=
model output:	687042+86208,,C:0
	68704+8620,0,C:1 # added 2+8+0=0
	6870+862,50,C:0 # added 4+0+1=5
	687+86,250,C:1 # added 0+2+0=2
	68+8,4250,C:1 # added 7+6+1=4
	6+,74250,C:1 # added 8+8+1=7
	+,774250,C:0 # added 6+0+1=7
	774250

Table 5. The generation of fine-tuned Llama-2-7B with scratchpad

input:	496943+2382=
model output:	496943+2382,,C:0
	49694+238,6,C:0 # added 4+2+0=6
	4969+23,26,C:1# added 4+8+0=2
	496+2,326,C:1 # added 9+3+1=3
	49+,9326,C:0 # added 6+2+1=9
	4+,99326,C:0 # added 9+0+0=9
	+,499326,C:0 #4+0+0=4
	499326

Table 6. The generation of fine-tuned Llama-2-7B with scratchpad

F.2. Ablation Study of RFFT

We conduct ablation studies of 5 modules of RFFT, including: 1. **process**: simplify the process of each iteration (addition of each digit), outputting multiple lines of code in one step; 2. **variable**: skip recalling relevant variables before executing each line of code; 3. **rule**: remove rules from the input (but still requires reciting the used rule in each output step); 4. **caption**: remove natural language instructions such as "1. Initialize Result and Carry" from the output which correspond to comments in the code; 5. **cite**: remove line-by-line recitation of the code from the output.

It is shown in Figure 5 that the model's performance deteriorates significantly when removing **cite** and **caption** components, especially cite. Both of the modules aid the model in recalling rules. **Variable** also has some performance impact as it helps the model reduce its reliance on distant text. On the contrary, **process** and **rule** do not have a significant impact, and in some cases, there is even a performance improvement. This may be because reducing the context length is beneficial for the model so that it can put more strength on rule execution, and an extremely detailed guidance in step-by-step rule-following (**process**) is not necessary for this task. In conclusion, **reciting the rule used in each step** and **reminding LLMs what the current step is doing** is crucial for RFFT's success, while there maybe room for simplifying the rule representation and execution, which is left for future research.
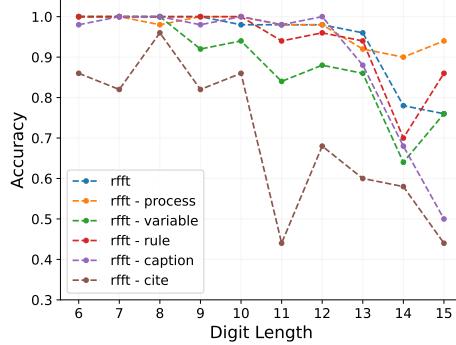
Figure 24. Ablations for RFFT.

F.3. Rule-following as a Meta Learning Ability
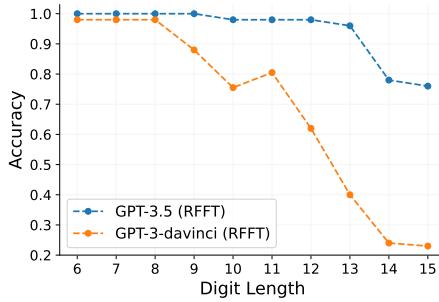
Figure 25. Accuracy of GPT-3.5 and davinci-002 fine-tuned with RFFT on addition with 6-15 digits

We fine-tune a larger model Llama-2-70b than Llama-2-7b and a slightly weaker model davinci-002 than GPT-3.5. For Llama-2-70b, we reduce the training set to 20k samples, smaller than 150k used in Llama-2-7b but achieve comparable results to Llama-2-7b as shown in Table 7. For davinci-002, we use the same data size as GPT-3.5 and Figure 25 shows that davinci-002 can achieve up to 8 digits generalization, worse than GPT-3.5. These results indicate that models with more advanced foundational abilities can achieve better length generalization after applying RFFT, even with small data, revealing that RFFT might be a meta learning (learning to learn) ability. Through more advanced pre-training, models might have "learned" the rule following ability so that a few examples are enough to learn a new task.

F.4. Failure Cases of RFFT and Scratchpad

F.4.1. SCRATCHPAD

As discussed in Section 5.2, scratchpad struggles to learn the rationales behind each step without explicitly providing rules. We offer two failure examples in Table 6 and 5. Specifically, the model makes mistakes at the "# added 4+2+0=6" step, indicating that it fails to locate the rightmost digit of the first number. Besides, the table 5 shows that the model cannot correctly compute the carry C. These issues are likely due to the model cannot comprehend the principle behind the 'added' step. In contrast, when rules are clearly provided, the model is better equipped to understand the rationale and perform the rule-following process, thereby reducing the difficulty in the learning process.

F.4.2. RFFT

Although RFFT can teach transformers to do rule-based reasoning, the basic capabilities of LLMs might limit their strict generalization, leading to occasional errors during some basic operations. Refer to Table 8. We observe that the model fails to output the correct digit popped out, resulting in its eventual incorrect answer.

F.5. Rule-following with Code Representations and Natural Language Representations

We offer two types of rule-following input-output sequences including code representations and natural language representations to show that rules can be of various formats. We provide examples of full input-output sequences for reference in Appendix I. The results are shown in Figure 26.

Besides, we will discuss the difference of our work from previous work Nye et al. (2021) which teaches LLMs to execute code (which they call scratchpad tracing) as follows. Our RFFT aims at teaching LLMs to follow explicitly provided rules to reason rather than to execute programs like an interpreter. We show in Figure 26 that rule can be of various forms including programs and natural language. Besides, we provide detailed natural language instructions in the rule-following input-output sequences with code representations. We expect the instructions may help LLMs to recall knowledge learned in the pre-training corpus.
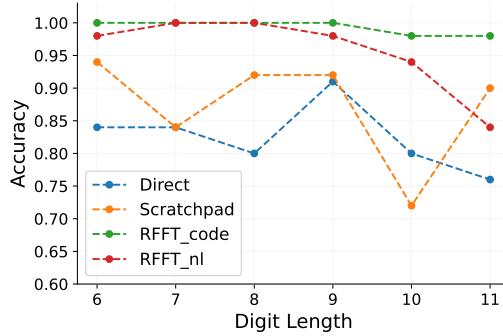
Figure 26. Test accuracy of GPT-3.5-turbo fine-tuned on 1-5 digit addition. We train the model in 4 different settings. The full input-output pair can be found in Appendix I.

F.6. Comparing RFFT with Scratchpad Tracing

We propose the technique of RFFT in §5, which instructs LLMs to follow rules of various forms. In §5, we use rules represented by programs; in Appendix F.5, we provide another option of rules represented by natural language. A related work Nye et al. (2021) introduces a method of fine-tuning LLMs to predict the program execution trace line by line, called "scratchpad tracing". We here state the difference between scratchpad tracing and our RFFT with programs as rules: 1) we provide the detailed execution process of each line of the code instead of directly giving the value of variables after line-by-line execution, through which we decompose each step of execution in a more fine-grained way; 2) we provide natural language instructions or the rationales behind each step to help the LLMs to understand the execution steps, for example "1. Initialize Result and Carry", "2. Loop Through Each Digit", etc. In summary, RFFT with programs teaches LLMs to execute code in a more human-readable way, simulating how human read, understand, and execute the code in their mind, while scratchpad tracing directly predicts program traces using raw machine formats.

To further demonstrate the effectiveness of our technique, we use scratchpad tracing to fine-tune GPT-3.5-turbo-1106 on the addition task. We still maintain the same data size and training parameters as RFFT, i.e., 100 training samples with 4 epochs and batch size of 4. The full input-output pair is provided in Appendix I. Considering the expensive cost with OpenAI API, we generate 100 test samples for each digit length and perform three independent experiments and report average accuracy and standard deviation. We show the results of RFFT and scratchpad tracing in Figure 27. RFFT significantly outperforms scratchpad tracing. The results show that the detailed execution process of each line of code and natural language instructions enhance the model's rule learning ability and improve length generalization.

F.7. RFFT on Other Tasks

Besides addition shown in §5, we conduct experiments of RFFT on two additional tasks including base-9 addition and last letter concatenation. Last letter concatenation is introduced in Wei et al. (2023). In the task, the model is asked to concatenate the last letters of words. We choose the words from top one-thousand last names from `https://namecensus.com/`.

We fine-tune GPT-3.5-turbo-1106 with a training set of 100 samples of 1-5 digit base addition or of concatenating the last
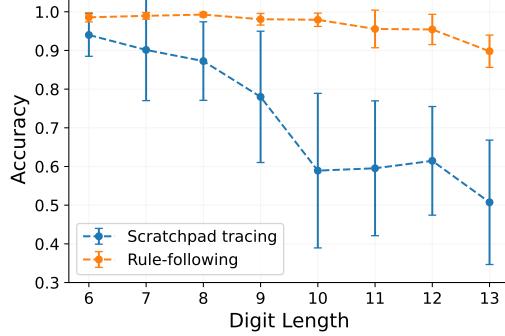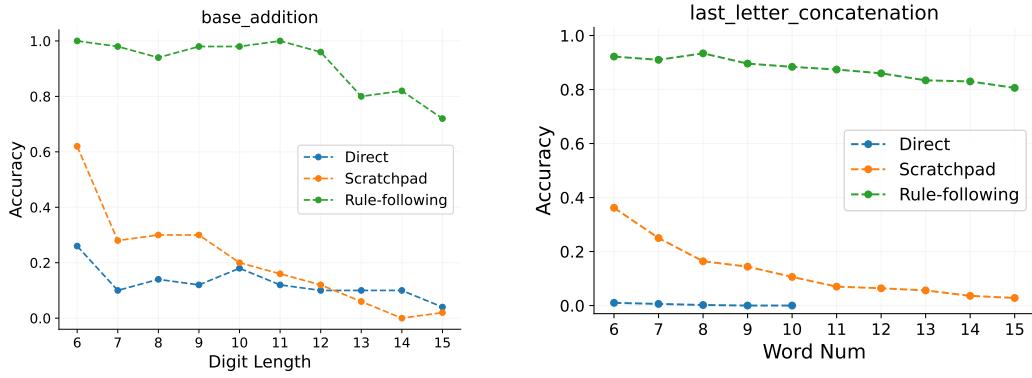
Figure 27. Results of RFFT and scratchpad tracing on addition.

Figure 28. Results of direct, scratchpad and RFFT on **base addition** (**left**) and **last letter concatenation** (**right**).

letter of 1-5 words respectively for two tasks. We train the models with OpenAI API with 4 epochs, 4 batch sizes like in Appendix F.1. Then, we test the model on test samples of 6-15 length. We list the full prompt for last letter concatenation in Appendix I.2, as the prompt for base addition is basically the same as that for addition.

The results are shown in Figure 28. RFFT outperforms the method of direct answer and scratchpad significantly, showing that RFFT enhances the model ability of following given rules to solve problems.

G. Scratchpad vs Direct Answer on Addition

We increase the number of training samples for scratchpad to 5,000 on the task of addition with GPT-3.5-turbo. In detail, we average the results over 3 models fine-tuned for 1 epoch respectively. We use a test set of 100 samples. As is shown in Figure 29, scratchpad with 5,000 training samples outperforms direct answer with 100 samples but is still worse than RFFT with 100 samples.

27
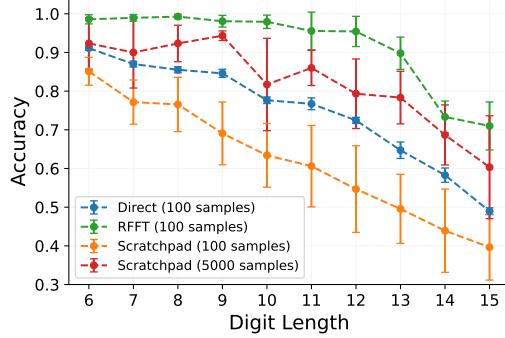
Figure 29. Results of direct answer with 100 training samples, scratchpad with 100 and 5,000 training samples respectively.

H. Experiments of Grokking

Nanda et al. (2023); Zhong et al. (2023) have claimed that transformers can learn systematic rules to solve modular addition. They show through experiments that transformers are embedding numbers as angles (points on the unit circles) and complete modular additions by operating on trigonometric functions of the angles. We perform the Leave-Square-Out method in the same settings as in Zhong et al. (2023). The only change is the training-test data split. Specifically, the task is $a + b$ mod $59 =$, $a \in [0, 58], b \in [0, 58]$. We leave a square test set of side length $l_k = 16$ (8% of the whole set) out and train 5 transformers with the same setting, while Zhong et al. (2023) split the dataset randomly with training set accounting for 80% of the whole dataset. We show that holes still appear, indicating that even such ability to learn the systematic algorithms and apply them to unseen samples rely severely on seeing similar cases. We describe the experiment in detail as follows.

The task is $a + b$ mod $59 =$, $a \in [0, 58], b \in [0, 58]$. We leave a test square of length $l_k = 16$ and center $(29, 29)$ out. Our training set accounts for about 92% of the whole dataset while the training set accounts for 80% in Zhong et al. (2023). The model can achieve 100% accuracy when the dataset is randomly split with training set accounting for 80%. This shows that the size of our training set is entirely sufficient for the model to solve the problem. Besides, we use the same hyper-parameters and model settings as given in the code of Zhong et al. (2023). We list them in Table 9.

The results are shown in Figure 30. There are holes in the test square, indicating that the model can not perform well in the test square. This shows that even in the settings where grokking happens, the ability to learn the systematic rules and apply it to test samples may still rely on seeing similar cases.
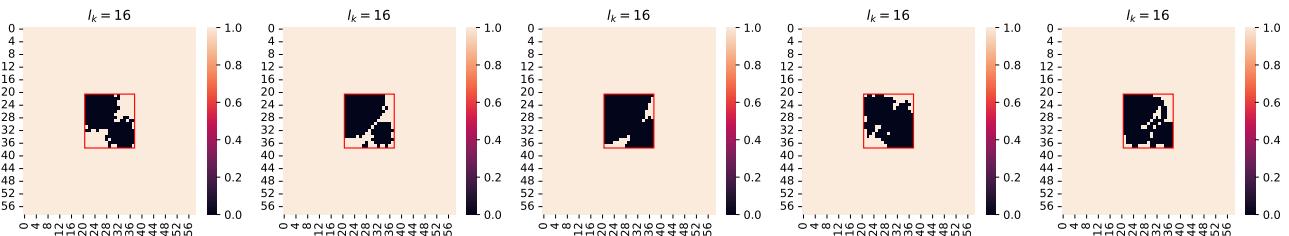
Figure 30. Accuracy distribution of the whole dataset when we leave a test square out. The test square is of length $l_k = 16$ and center $(29, 29)$. We repeat the experiment for 5 times with the same settings.

epoch	20000
batch size	3481
learning rate	0.001
weight decay	2.0
model dimension	128
n_head	4
n_layer	1

Table 9. Hyperparameters and model details of the experiments of grokking.

I. Full Input-Output Sequences

In this section, we list the full input-output pairs used in §5 and in Appendix F.7.

I.1. Addition

We list the input-output sequences of addition as follows.

Direct answer   We list an example of the input-output sequences for direct answer in Table 10.

Scratchpad   We list an example of the input-output sequences for scratchpad in Table 11.

Scratchpad tracing   We list an example of the input-output sequences for scratchpad tracing Table 12.

Rule-following   We list an example of the input for rule-following in Table 13 and an example of the output in Table 14.

Rule-following with natural language representations   We list an example of the input for rule-following with NL representations in Table 19 and an example of the output in Table 20.

I.2. Last letter concatenation

We list the input-output sequences of last letter concatenation as follows.

Direct answer   We list an example of the input-output sequences for direct answer in Table 15.

Scratchpad   We list an example of the input-output sequences for scratchpad in Table 16.

Rule-following   We list an example of the input for rule-following in Table 17 and an example of the output in Table 18.

input:	Follow the code step by step to answer the question:
	[7, 8, 5, 3]+[8, 6, 8, 8, 1, 0]=
	def sum_digit_by_digit(num1, num2):
	# Initialize the result list and carry
	result=[]
	carry=0
	...

model output:	num1=[7, 8, 5, 3]	```
	num2=[8, 6, 8, 8, 1, 0]	*digit2=num2.pop() if num2 else 0*
	1. Initialize Result and Carry	```
	result=[]	num2=[8,6,8,8,1]
	carry=0	bool(num2)=True
	2. Loop Through Each Digit	num2.pop()
	``	num2=[8,6,8,8]
	while num1 or num2:	digit2=8
	``	```
	2.1 check the stop criterion	*total=digit1+digit2+carry*
	num1=[7,8,5,3]	```
	num2=[8,6,8,8,1,0]	total=digit1+digit2+carry=5+8+0=13
	bool(num1)=True	```
	bool(num2)=True	result.insert(0,total%10)
	num1 or num2=True	```
	enter the loop	result=[3] total%10=13%10=3 result=[3,3]
	2.2 one iteration	carry=13//10=1
	``	...
	digit1=num1.pop() if num1 else 0	**2.3 back to the start of the loop**
	``	```
	num1=[7,8,5,3]	*while num1 or num2:*
	bool(num1)=True	```
	num1.pop()	**2.1 check the stop criterion**
	num1=[7,8,5]	num1=[]
	digit1=3	...
	...	num1 or num2=False
	result=[3]	end the loop
	``	**3. Check Remaining Carry**
	carry=total//10	```
	``	*if carry:*
	carry=3//10=0	*result.insert(0, carry)*
	...	```
	2.2 one iteration	result=[8,7,6,7,3,3]
	``	carry=0
	digit1=num1.pop() if num1 else 0	bool(carry)=False
	``	pass
	num1=[7,8,5]	**4. Return Result**
	bool(num1)=True	```
	num1.pop()	*return result*
	num1=[7,8]	```
	digit1=5	result=[8,7,6,7,3,3]

Table 8. The generation of fine-tuned Llama-2-7B with RFFT.

input:	7+6=
output:	13

Table 10. Input-output sequence for direct answer on addition.

input:	7+6=
output:	7+6,,C:0
	# added 7+6+0=3
	+,3,C:1
	# added 0+0+1=1
	+,13,C:0
	13

Table 11. Input-output sequence for scratchpad on addition.

input:	```def sum_digit_by_digit(num1, num2):
result=[]
carry=0
while num1 or num2:
digit1=num1.pop() if num1 else 0
digit2=num2.pop() if num2 else 0
total=digit1+digit2+carry
result.insert(0,total%10)
carry=total//10
if carry:
result.insert(0, carry)
return result
``

output = sum_digit_by_digit([7, 3], [4, 8])

output: state: {}
line: def sum_digit_by_digit(num1, num2):
state: {'sum_digit_by_digit': '<callable_object sum_digit_by_digit>'}
line: result=[]
state: {'num1': [7, 3], 'num2': [4, 8], 'result': []}
line: carry=0
state: {'num1': [7, 3], 'num2': [4, 8], 'result': [], 'carry': 0}
line: while num1 or num2:
state: {'num1': [7, 3], 'num2': [4, 8], 'result': [], 'carry': 0}
line: digit1=num1.pop() if num1 else 0
state: {'num1': [7], 'num2': [4, 8], 'result': [], 'carry': 0, 'digit1': 3}
line: digit2=num2.pop() if num2 else 0
state: {'num1': [7], 'num2': [4], 'result': [], 'carry': 0, 'digit1': 3, 'digit2': 8}
line: total=digit1+digit2+carry
state: {'num1': [7], 'num2': [4], 'result': [], 'carry': 0, 'digit1': 3, 'digit2': 8, 'total': 11}
line: result.insert(0,total%10)
state: {'num1': [7], 'num2': [4], 'result': [1], 'carry': 0, 'digit1': 3, 'digit2': 8, 'total': 11}
line: carry=total//10
state: {'num1': [7], 'num2': [4], 'result': [1], 'carry': 1, 'digit1': 3, 'digit2': 8, 'total': 11}
line: while num1 or num2:
state: {'num1': [7], 'num2': [4], 'result': [1], 'carry': 1, 'digit1': 3, 'digit2': 8, 'total': 11}
line: digit1=num1.pop() if num1 else 0
state: {'num1': [], 'num2': [4], 'result': [1], 'carry': 1, 'digit1': 7, 'digit2': 8, 'total': 11}
line: digit2=num2.pop() if num2 else 0
state: {'num1': [], 'num2': [], 'result': [1], 'carry': 1, 'digit1': 7, 'digit2': 4, 'total': 11}
line: total=digit1+digit2+carry
state: {'num1': [], 'num2': [], 'result': [1], 'carry': 1, 'digit1': 7, 'digit2': 4, 'total': 12}
line: result.insert(0,total%10)
state: {'num1': [], 'num2': [], 'result': [2, 1], 'carry': 1, 'digit1': 7, 'digit2': 4, 'total': 12}
line: carry=total//10
state: {'num1': [], 'num2': [], 'result': [2, 1], 'carry': 1, 'digit1': 7, 'digit2': 4, 'total': 12}
line: if carry:
state: {'num1': [], 'num2': [], 'result': [2, 1], 'carry': 1, 'digit1': 7, 'digit2': 4, 'total': 12}
line: result.insert(0, carry)
state: {'num1': [], 'num2': [], 'result': [1, 2, 1], 'carry': 1, 'digit1': 7, 'digit2': 4, 'total': 12}
line: return result
state: {'sum_digit_by_digit': '<callable_object sum_digit_by_digit>', 'output': [1, 2, 1]}

Table 12. Input-output sequence for scratchpad tracing on addition.

input: Follow the code step by step to answer the question:

[7]+[6]=

```
def sum_digit_by_digit(num1, num2):
    # Initialize the result list and carry
    result=[]
    carry=0
    # Loop through each digit
    while num1 or num2:
        # Get the current digits, defaulting to 0 if one number is shorter
        digit1=num1.pop() if num1 else 0
        digit2=num2.pop() if num2 else 0
        # Calculate the sum of the current digits and the carry
        total=digit1+digit2+carry
        # Insert the last digit of total to the beginning of the result and update carry
        result.insert(0,total%10)
        carry=total//10
    # If there's a remaining carry, insert it to the beginning of the result
    if carry:
        result.insert(0, carry)
    # Return the result
    return result
```

Table 13. Input for rule-following on addition.

output: num1=[7]
num2=[6]
1. Initialize Result and Carry
result=[]
carry=0
2. Loop Through Each Digit
```
while num1 or num2:
```
2.1 check the stop criterion
num1=[7]
num2=[6]
bool(num1)=True
bool(num2)=True
num1 or num2=True
enter the loop
2.2 one iteration
```
digit1=num1.pop() if num1 else 0
```
num1=[7]
bool(num1)=True
num1.pop()
num1=[]
digit1=7
```
digit2=num2.pop() if num2 else 0
```
num2=[6]
bool(num2)=True
num2.pop()
num2=[]
digit2=6
```
total=digit1+digit2+carry
```
total=digit1+digit2+carry=7+6+0=13

```
result.insert(0,total%10)
```
result=[]
total%10=13%10=3
result=[3]
```
carry=total//10
```
carry=13//10=1
2.3 back to the start of the loop
```
while num1 or num2:
```
2.1 check the stop criterion
num1=[]
num2=[]
bool(num1)=False
bool(num2)=False
num1 or num2=False
end the loop
3. Check Remaining Carry
```
if carry:
    result.insert(0, carry)
```
result=[3]
carry=1
bool(carry)=True
result=[1,3]
4. Return Result
```
return result
```
result=[1,3]

Table 14. Output for rule-following on addition.

input:	Concatenate the last letter of the following names: King, Kaur
output:	gr

Table 15. Input-output sequence for direct answer on last letter concatenation.

input:	Concatenate the last letter of the following names: King, Kaur
output:	# Concatenating the last letter of King g # Concatenating the last letter of Kaur gr

Table 16. Input-output sequence for scratchpad on last letter concatenation.

input:	Follow the code step by step to concatenate the last letter of the following names:
	King, Kaur
	def last_letter_concat(names):
	   # Initialize Result
	   result = ""
	   # Main Loop
	   for name in names:
	     result += name[-1]
	   return result

Table 17. Input for rule-following on last letter concatenation.

output:	names = ['King','Kaur'] **1. Initialze result** result = "" **2. Main Loop** **2.1 one iteration** ` ` ` for name in names: ` ` ` name = "King" ` ` ` result += name[-1] ` ` ` result = "" name[-1] = "g" result += "g" result = "g"	**2.1 one iteration** ` ` ` for name in names: ` ` ` name = "Kaur" ` ` ` result += name[-1] ` ` ` result = "g" name[-1] = "r" result += "r" result = "gr" **3. Return Result** ` ` ` rerturn result ` ` ` result = "gr"

Table 18. Output for rule-following on last letter concatenation.

input:	Follow the rules step by step to answer the question: '6 '+ '7 '=
	Add two numbers in order from the lowest digit to the highest digit. The operation rules are as follows:
	1. In the initial state, the carry from the previous digit is 0 and the result is <empty>.
	2. Begin the loop through each digit:
	2.1. Determine whether the two numbers are <empty>. If both are <empty>, the loop is over. If one of them is not <empty>, go to the iteration.
	2.2. One iteration:
	2.2.1. Find the rightmost digit of each number and remove the digit from the number. If one of them is <empty>, the digit is 0.
	2.2.2. Calculate the sum of the current digits and the carry.
	2.2.3. Insert the last digit of the summation to the beginning of the result and update carry. If the sum is larger than 9, carry is 1. Otherwise, carry is 0.
	2.3. Back to the start of the loop.
	3. When the loop through each digit is over, if there's a remaining carry, insert it to the beginning of the result.
	4. Return the result.

Table 19. Input for rule-following-NL.

output: num1= '6 '; num2= '7 '
```
1. In the initial state, the carry from the previous digit is 0 and the result is <empty>.
```
Initialize the carry as 0 and the result is '<empty> '.
```
2. Begin the loop through each digit:
2.1. Determine whether the two numbers are <empty>. If both are <empty>, the loop is over. If one of them is not <empty>, go to the iteration.
```
num1 '6 ' is not <empty>. num2 '7 ' is not <empty>.
Because both are not <empty>, go to the iteration.
```
2.2. One iteration:
2.2.1. Find the rightmost digit of each number and remove the digit from the number. If one of them is <empty>, the digit is 0.
```
The rightmost digit of num1 '6 ' is '6 '. The rightmost digit of num2 '7 ' is '7 '. Then the remaining num1 '6 ' is '<empty> ' and the remaining num2 '7 ' is '<empty> '.
```
2.2.2. Calculate the sum of the current digits and the carry.
```
The sum of '6 ', '7 ' and carry '0 ' is 6 + 7 + 0 = 13.
```
2.2.3. Insert the last digit of the summation to the beginning of the result and update carry.
```
The last digit of the summation is 13 % 10 = 3. The previous result is '<empty> ', now the result is: '3 '.
The carry is 13 // 10 = 1.
```
2.3. Back to the start of the loop. ```
```
2.1. Determine whether the two numbers are <empty>. If both are <empty>, the loop is over. If one of them is not <empty>, go to the iteration.
```
num1 is '<empty> '. num2 is '<empty> '.
Because both are <empty>, the loop is over.
```
3. When the loop through each digit is over, if there's a remaining carry, insert it to the beginning of the result.
```
The carry is 1, so add an 1 to the beginning of the result '3 '. So the final result is '1,3 '.
```
4. Return the result. ```
The final result is '1,3 '.

Table 20. Output for rule-following-nl.