IGC: Integrating a Gated Calculator into an LLM to Solve Arithmetic Tasks Reliably and Efficiently

Anonymous ACL submission

Abstract

Solving arithmetic tasks is a simple and fundamental skill, yet modern Large Language Models (LLMs) have great difficulty with them. We introduce the Integrated Gated Calculator (IGC), a module that enables LLMs to perform arithmetic by emulating a calculator on the GPU. We finetune a Llama model with our module and test it on the BigBench Arithmetic benchmark, where it beats the State of the Art, outperforming all models on the benchmark, including models almost two orders of magnitude larger. Our approach takes only a single iteration to run and requires no external tools. It performs arithmetic operations entirely inside the LLM without the need to produce intermediate tokens. It is computationally efficient, interpretable, and avoids side-effects on tasks that do not require arithmetic operations. It reliably achieves 98% to 99% accuracy across multiple training runs and for all subtasks, including the substantially harder subtask of multiplication, which was previously unsolved.

1 Introduction

Motivation. Large Language Models (LLM) have shown impressive abilities in many different fields in recent years (Thoppilan et al., 2022; Chowdhery et al., 2023; Brown, 2020). This makes it all the more intriguing that even advanced LLMs still perform very poorly on basic arithmetic tasks: GPT-3 has trouble adding numbers with more than three digits (Brown, 2020) and GPT-4 (Achiam et al., 2023) still fails to solve multiplication tasks (Dziri et al., 2024). The reasons for this surprisingly poor performance have been studied extensively (Yuan et al., 2023; Brown, 2020; Dziri et al., 2024). Even so the BigBench Arithmetic benchmark (bench authors, 2023), which tests the four basic arithmetic operations on merely 5-digit long numbers, remains unsolved.

The Impact of Number Representations. The arithmetic abilities of LLMs strongly depend on the way numbers are represented (Thawani et al., 2021). McLeish et al. (2024) improve arithmetic performance by adding positional encodings to numbers. Similarly, Liu and Low (2023) show that good performance can be achieved on addition and subtraction tasks using finetuning only, and they attribute this to the especially well-suited tokenization method of their model. However, we are not aware of any finetuning method that solves the more difficult subtask of multiplication effectively.

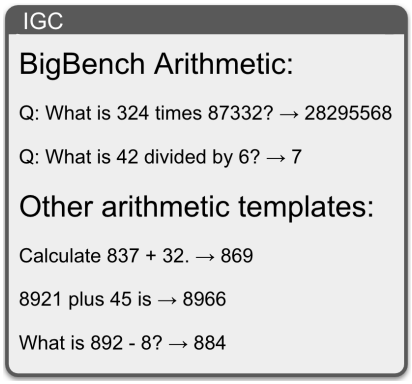

Figure 1: Examples of arithmetic tasks.

Chain of Thought. Chain of Thought (COT) is a prompting method that works by breaking the task down into smaller subtasks and solving them step-by-step (Wei et al., 2023). This approach makes many difficult tasks solvable, but it also increases the model's runtime as it requires many intermediate outputs to produce the final result.

Tool Use. Schick et al. (2023) introduced the Toolformer, which teaches LLMs to call external tools. This method is very powerful, but it increases the inference time due to costly transfers of data between the GPU and CPU. Moreover, since this method is only added after pretraining, the LLM can't learn to condition its predictions on the results of arithmetic operations during pretraining. Since arithmetic is a fundamental building block of more complex tasks, it would be worthwhile to

enable LLMs to solve arithmetic tasks directly during pretraining, so that it can use this ability as a subroutine for more complex problems.

Our Solution. We develop the *Integrated Gated Calculator* (IGC), a module that enables an LLM to accurately perform arithmetic operations, in a single iteration and without using external tools. It extracts numbers from the tokens in a categorical representation and then emulates the calculation directly on the GPU.

Our contributions are:

- **Innovation**. We introduce the *Integrated Gated Calculator*, a novel module that can emulate a calculator (Section 3). We modify a pretrained Llama 3.1 8B model (Touvron et al., 2023; Dubey et al., 2024) with it and finetune on synthetic data. This enables the LLM to solve complex arithmetic tasks reliably, directly on the GPU, without using COTs, a scratchpad, or calling any tools.

- **Results**. We achieve near-perfect generalization on the BigBench Arithmetic Benchmark, outperforming SOTA models that are almost two orders of magnitude larger (Section 4). To the best of our knowledge, we are the first to enable an LLM to solve multiplication tasks without the use of external tools or lengthy multi-step procedures.

- **Analysis**. We compare our approach with alternatives (Section 5) and find that it has numerous advantages besides its strong performance on the benchmark: It is both computationally efficient and interpretable, and it can learn to avoid side effects and destructive interference for problems that do not require arithmetics.

- **Future Work**. We describe how the IGC could be integrated into an LLM during pretraining instead of finetuning (Section 6). This would allow the LLM to learn to use it as a subroutine for more complex tasks, an ability that is missing from alternative approaches. We further describe how our approach could be generalized and extended to other non-differentiable operations, such as looking up items in a database.

2 Related Work

Word Problems. We want to highlight the fact that arithmetic tasks are different from math word problems. In some cases, models fail to solve word problems even though they follow correct reasoning, because they get the arithmetic operations wrong (Schick et al., 2023; Cobbe et al., 2021; Gao et al., 2023). In other cases, models fail to solve word problems with trivial arithmetic operations because they fail to extract the numbers or to format the output correctly. We see examples of this in our analysis of existing benchmark data in Section 4. In this paper we focus on arithmetic. We discuss in Future Work (Section 6) how our method could be integrated into an LLM more effectively than alternative approaches, which should help greatly with word problems, too.

Chain of Thought. Chain of Thought methods have shown promising results on a variety of different tasks and for many different models (Nye et al., 2021; Chung et al., 2024). Lee et al. (2023) investigated ways to teach arithmetic to small transformers and found that COT can help significantly on this task as well.

Tool Use. Schick et al. (2023) introduced the Toolformer, a generic method to enable an LLM to interact with an external tool. By interacting with a calculator, this method can perfectly solve arithmetic tasks of any complexity. Tool use is a very generic technique with applications for many different tasks and domains (Qu et al., 2024).

Other Approaches for Arithmetic Tasks. Cobbe et al. (2021) train verifiers to solve math word problems. Nye et al. (2021) add scratchpads to the COT approach. Imani et al. (2023) compares several Chains of Thoughts to improve reliability. Chen et al. (2022) and Gao et al. (2023) combine COT and Tool Use by generating executable code.

Modifying LLMs. Many different techniques for modifying and finetuning LLMs exist Ding et al. (2022). Our approach is most similar to Adapter-based methods (Houlsby et al., 2019), which work by injecting a separate smaller neural network into a pretrained LLM. This Adapter module is trained to modify one of the intermediate activations of the LLM, while the base LLM's parameters are kept frozen. However, our method has several important differences to typical Adapter methods, which we explain in the next Section.

3 Methods

Approach. Figure 2 gives an overview of our approach. We introduce the Integrated Gated Calculator (IGC), a new module that modifies the output

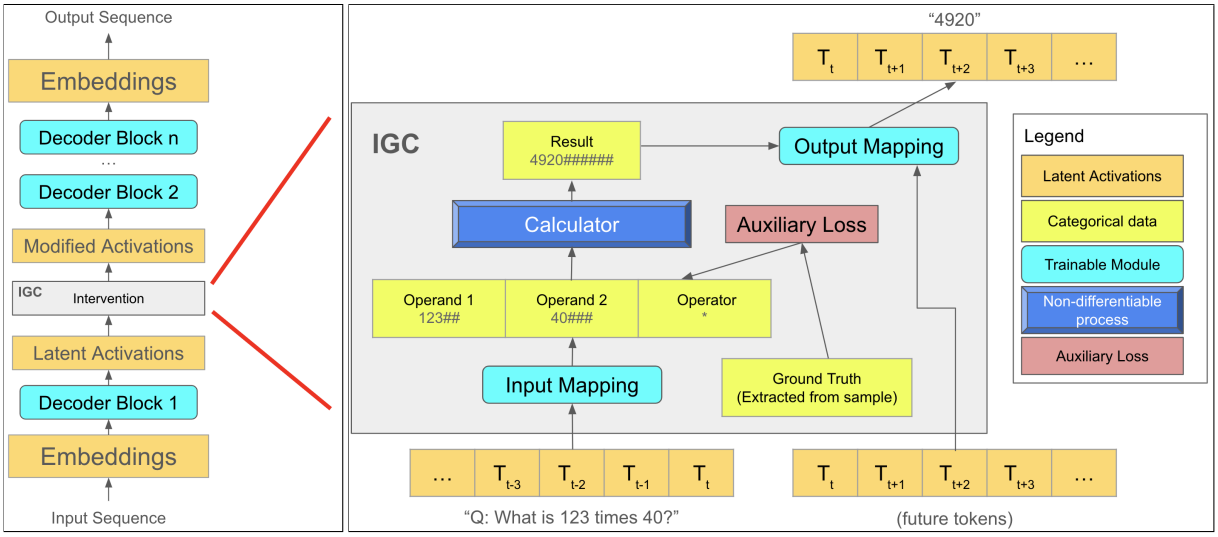

Figure 2: **Left**. The IGC is inserted into a pretrained LLM after a fixed layer, in this case layer 1. It modifies the output produced by that layer. **Right**. During training, the IGC takes the latent activations produced by the layer as its inputs and splits them into two parts: Before and after the anchor token T_t at time step t, which has a special role for argument selection. The IGC comprises three components, two of which are trainable submodules: The Input Mapping submodule (Figure 3, left) uses the tokens before T_t to extract the arithmetic task from the text and to format it for the calculator. It is trained through an auxiliary loss. The calculator itself is emulated on the GPU through a sequence of non-differentiable tensor operations. It is not a trainable component. The Output Mapping submodule (Figure 3, right) uses the results of the calculator to modify the tokens after T_t. It is trained by the LLM's normal loss function. Note that this image shows the training process using teacher forcing. During inference, the Input Mapping and the calculator are executed only on the iteration when the anchor token arrives. Their outputs are cached and reused on subsequent iterations.

of an existing layer of a pretrained LLM. We keep the LLM's existing weights frozen and train only the weights of the IGC. This is similar to Adapter-based tuning methods, but with several important differences:

- It has non-differentiable components.
- It operates on multiple tokens at once.
- It is executed in discrete steps.
- It uses gated connections on its outputs.

We explain the reasons for and implications of each of these differences in the following.

3.1 Main Considerations

Non-differentiable Components. The IGC uses tensor operations to emulate a calculator directly on the GPU. The calculator's input and output digits are represented with discrete categorical data, which makes it a non-differentiable operation. Therefore, the calculator itself is not a trainable component and it blocks the gradient coming from the LLM's main loss. We therefore have two separate trainable components, which are illustrated in Figure 3: The Output Mapping, which is trained as normal, and the Input Mapping, which does not receive a gradient because of the calculator. This necessitates a custom training method using an auxiliary loss, which we describe in Section 3.2.

Dependency on Multiple Tokens. Most applications of Adapter-based methods care about abstract concepts like sentiment, or about the presence or absence of specific named entities, since this type of information is often tested in Natural Language Benchmarks such as GLUE (Wang et al., 2019; Houlsby et al., 2019). LLMs can encode such information in a single token or a fixed set of tokens. However, numbers are encoded in several sequential tokens instead, and their relative position is crucial. Our architecture needs to reflect this. We therefore have to learn a mapping from a variable-length set of tokens to the fixed-size input of the calculator. Conversely, for the output, we need to map the fixed-size number back to all subsequent tokens. We implement this through attention mechanisms and dynamic gating weights. We did try a simpler variant for comparison, in which the Input Mapping submodule used fixed inputs, and this architecture performed much worse.

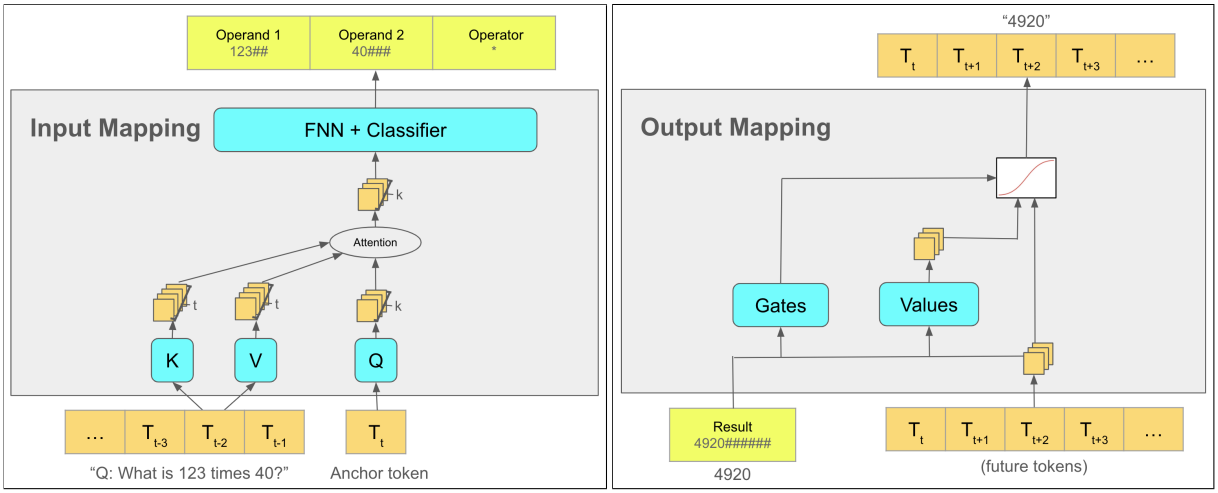

Figure 3: **Left**. The Input Mapping submodule takes variable-length textual embeddings and extracts the numbers and operator as fixed-length categorical data. The operands and operator are produced as probability distributions over possible digit values for each digit. **Calculator (not shown)**. The calculator discretizes the distributions produced by the Input Mapping submodule by sampling the most probable number and operator. It then emulates the arithmetic operation. The resulting number is formatted using one-hot encoding. **Right**. The Output Mapping submodule uses the fixed-length output of the calculator to modify each of the output tokens. This uses a separate learned gating weight for each token so that it can easily learn to leave tokens unchanged.

Discrete Execution. Performing an arithmetic operation is a discrete task and not a distributed heuristic process. One does not perform 5% of a calculation one moment and $\%7$ the next. A calculation is either performed or it is not. Our module is designed to reflect this: Most adapter methods work by applying a module to the *current* token on *each* iteration. We instead perform no operations until we are sure that the arithmetic task has been fully described, at time t: When the chat switches from the user to the system. The IGC is then run once, using all tokens available. The token T_t at time t is called the anchor token. The Input Mapping submodule uses it to construct a Query in an attention mechanism, to find all tokens relevant for the arithmetic operation. During training, we use a form of teacher-forcing described in Section 3.2. During inference, the results of our module at time t are cached for later iterations. This unusual implementation has several benefits:

- **More Effective Training**. We can not be certain what arithmetic operation is needed until all relevant tokens have been encountered. If we try to train the Input Mapping before all of the information it needs is available, it can only learn to guess. This would introduce noise and disrupt the training process.

- **Avoiding Redundancy**. If there is only one arithmetic operation, then every iteration after t should learn to perform the exact same Input Mapping. There would be no benefit in repeating the operation multiple times.

Gated Outputs. The Output Mapping submodule uses gated connections with learned and dynamically calculated weights to modify the tokens after T_t. It can learn how much each of the output tokens needs to be modified, and it can simply use near-zero weights to avoid making changes for tasks that do not require arithmetic. As a consequence, the IGC causes no destructive interference in tasks where it is not needed. We have experimentally confirmed that this works by testing our architecture on non-arithmetic tasks after training it for the arithmetic tasks described in Section 4. These tasks used easily recognizable input templates, so confirming it for more complex word problems remains as future work.

3.2 Training Method

Our training is based on the usual teacher-forcing approach. However, because the calculator is non-differentiable and blocks the gradient to the Input Mapping submodule, we have to slightly modify the algorithm.

Ground Truth Data. We annotate our training data with ground truth data that specifies for each sample which arithmetic operation needs to be called, and with which operands. For simple arithmetic templates, this information can easily

be extracted automatically. For more difficult examples, such as word problems, training data can be created by the LLM itself, through a process analogous to the one described in the Toolformer paper (Schick et al., 2023): The LLM annotates its existing training data with arithmetic operations that it believes would have been helpful. We then measure if annotating the data with the result of that operation reduces perplexity compared to the un-annotated data. If it does, we add the annotation to our training data.

Modified Training Process. We use this helper information to modify the training process in two ways: Firstly, we apply an auxiliary loss to the Input Mapping component. This is just a simple cross-entropy loss that teaches the Input Mapping submodule to produce the correct input to the calculator. Secondly, we replace the output of the calculator with the correct output, so that the Output Mapping can begin training immediately, before the Input Mapping has converged. This second step is not strictly necessary, but speeds up training.

One IGC execution per sample. In our training data, each sample requires exactly one use of the calculator, although these can happen at different times for each sample. This limitation simplifies the training process and allows for more efficient parallelization without loss of generality: Multi-step arithmetic tasks can simply be broken up into individual samples for each step of the calculation.

3.3 Implementation Details

Effects of Tokenization. Previous studies have found that tokenization has a large impact on arithmetic abilities (Kim et al., 2021; Ding et al., 2022; Liu and Low, 2023; Garreth, 2024). For training to be efficient, we need to ensure an inductive bias so that similar tokens can easily be mapped to similar internal representations, for both the input and the output of our module. The key consideration is that numbers are tokenized from left to right and the calculator's representation of digits must reflect this by being left-aligned: The most significant digit must be assigned to a fixed index, not the least significant one. All architectures described in this paper use this left-aligned format. Additionally, we need to consider how digits are chunked into tokens. While older versions of Llama used one token per digit (Yuan et al., 2023; Liu and Low, 2023), Llama 3.1 (Dubey et al., 2024) groups numbers together in chunks of up to three digits. This makes it harder for the model to infer the position of each digit, which caused significant problems when we used the more intuitive right-aligned format instead of the left-aligned one. However, using the left-aligned pattern solved the problem and allowed us to meet and exceed the performance of other approaches. We expect that the IGC can be adjusted to the tokenization methods and number representations used by other LLMs in a similar manner. See Section A in the Appendix for details.

Choosing the Layer. We tried applying our module at several different layers of the LLM. We obtained the best results at early layers. The results reported in our experiments are all based on applying the module to layer 1.

4 Experiments

The BigBench Arithmetic Benchmark. In this paper we focus on arithmetic accuracy and not mathematical reasoning in general. We therefore picked the BigBench Arithmetic benchmark for our evaluations (bench authors, 2023). It uses a deliberately simple template that focuses on raw arithmetic and has a good balance of different operators and input lengths.

Alternate Templates. To ensure that the simplicity of the BigBench Arithmetic template does not skew results, we additionally trained and tested on several custom templates, as shown in Figure 1, in order to make the task more diverse and challenging. We noticed no difference in performance between these templates.

Comparisons. We modify a pretrained and instruction-finetuned Llama 3.1 8B model with an IGC, train it on synthetic data, and test it on the benchmark. The best existing results on the benchmark were achieved by PALM 535B (Chowdhery et al., 2023), which is significantly larger than our model. We therefore also report the performance of relatively smaller models that are no more than one order of magnitude larger than our model. Unfortunately, a direct comparison with existing benchmark results is slightly unfair, since these were obtained with n-shot prompting instead of finetuning. We deliberately make things harder for ourselves in order to strengthen the validity of our results: We report the *average* performance of our runs and compare it to the *best* performance of the n-shot runs. Additionally, we also compare our model to a second baseline, which is based on finetuning: We enhance a Llama model with an Adapter method, using the same parameter count as our model, and

finetune it on the same data. It should also be noted that finetuning more accurately tests for arithmetic abilities than n-shot prompting does. This is evident from inspecting the benchmark results and comparing n-shot results for different values of n: There is a lot of variance in performance, even though the arithmetic tasks are exactly the same, and often the 1-shot version outperforms n-shot variants with higher n. The loss of accuracy comes from an inability to understand the formatting required for the output, not an inability to perform the calculation.

Results. Our model outperforms all baselines by a large margin.

- Our model's performance is close to perfect across the board, even for the substantially harder subtask of multiplication.
- Our model is much better than the best of the smaller benchmark models.
- Compared to PALM 535B, which is optimized for mathematics and almost two orders of magnitude larger, our model is still slightly better for most subtasks and significantly better at multiplication.
- We significantly outperform our finetuning-based baseline, which shows that our model's great performance can be attributed to our novel architecture and not to the difference in evaluation methods.
- Our experiments had low variance and consistently converged to the reported values for multiple random seeds. In contrast, many of the n-shot benchmarks had high variance and the worst runs of our method still outperformed the best n-shot variants in the benchmark.

Investigating Anomalies. Curiously, our model performs slightly worse at the division subtask than the other operations, even though division is much simpler than multiplication. After investigating the possible causes, we attribute this to the fact that the BigBench Arithmetic benchmark used a different algorithm for generating random operands than we did. The test data therefore follows a different distribution than the training data.

Ablations. We also created hybrid modules in which the Input Mapping produces an additional output that is given to the Output Mapping submodule directly, making it end-to-end trainable just like a normal finetuning method. Figure 4 shows convergence behavior and final performance for several architectures: Our original IGC module without this shortcut, the IGC module with the shortcut, and a baseline that uses only the shortcut and uses no integrated calculator. All architectures start at zero accuracy because the LLM makes trivial formatting mistakes. They rapidly improve as they learn the template. The variant that uses only the shortcut showed no improvements after this initial adaptation and its performance is equal to the finetuning baseline. This is unsurprising, since the pretraining likely already included basic arithmetic operations. The IGC module converges within 70 epochs and its test performance remains stable after convergence. We observe high variance in the test accuracy for the IGC+shortcut hybrid models, but their final performance is overall lower than the pure IGC without a shortcut connection. We hypothesize that this is caused by overfitting and destructive interference: The finetuning component converges faster than the calculator, but it does not generalize well.

Model Sizes and Efficiency. The IGC has 17 million parameters and is integrated into a pretrained Llama model with 8 billion parameters. Meanwhile, PALM 535B has almost two orders of magnitude more parameters than the Llama model, and four orders of magnitude more than the IGC. The size of our module is trivial compared to the gain in performance it provides. The training process is fast and efficient as well: We generated an optimized dataset by filtering out frequently-occurring subsequences of tokens. Using this technique, our models converged within two to four days of training with a set of only 10,000 samples, on a single GPU. We note that such a small dataset is not enough for a normal model to learn how to generalize, as evidenced by our ablations. This shows that our approach is less data hungry than alternative approaches. We suspect that the reason for this is that the IGC's internal calculator is a perfect emulator, resulting in a much better inductive bias than a randomly initialized neural network.

5 Comparison to Other Methods

Table 2 shows a high-level comparison between our method and other methods for solving arithmetic tasks.

Task	Best small model	PALM 535B	Llama 8B baseline	IGC
#Parameters	< 100B	535B	8B+17M	8B+17M
Method	n-shot	n-shot	finetuning	finetuning
Overall	0.49	0.94	0.70	**0.99**
Addition	0.49	0.94	0.95	**0.99**
Subtraction	0.69	0.96	0.88	**0.99**
Multiplication	0.35	0.91	0.22	**0.99**
Division	0.71	0.97	0.75	**0.98**

Table 1: The accuracy of different models on the BigBench Arithmetic benchmark. The two columns about n-shot methods were extracted from official benchmark results, while the two finetuned variants were trained by us. We report the averages for the finetuned models and the best value for any n for the n-shot benchmarks. Despite this lopsided comparison, our model still outperforms everything else.

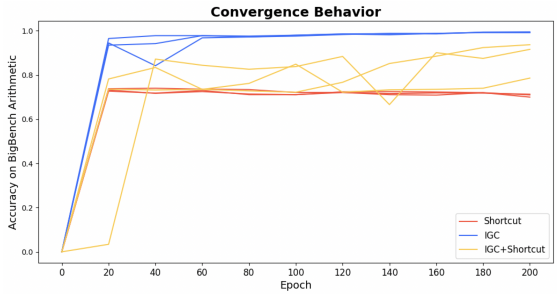

Figure 4: The accuracy of various architectures on the BigBench Arithmetic benchmark as training proceeds. Multiple lines with the same color correspond to different random seeds for the same architecture.

Capability. The only method that can solve arbitrary arithmetic tasks as reliably as the IGC is the use of external tools.

Efficiency. Our method runs in a single step and avoids expensive transfers of data between GPU and CPU, making it more efficient than both COT and tool use.

Interpretability. The IGC is highly interpretable because the numbers and operator are represented explicitly. Moreover, the results of the calculation are mapped back to the LLM using learned gates, which allows us to measure how much they affect future tokens.

Integration. We have reason to believe that our method can be more cleanly integrated into an LLM than other methods. Firstly, the IGC is entirely *internal* to the model and does not affect output tokens directly. This is important, because the output is an information bottleneck. If the arithmetic operation is only a subtask of a larger task, needing to generate tokens for it may distract the LLM from its main task. Additionally, teacher forcing generates separate gradients for each output token, which means that later steps taken for the same arithmetic operations can not repair mistakes made at earlier steps. In contrast, since the IGC is internal to the model and only executed once, it receives a single, coherent gradient. Secondly, all three approaches (IGC, COT, tool use) share a common weakness, but only the IGC offers a path to fix that weakness: All three approaches are *added after pretraining*. This implies that the model can not know what the true result of a calculation is during pretraining. It is therefore forced to learn how to make plausible guesses. Later, when we add the technique to solve arithmetic tasks correctly, the LLM has to unlearn these heuristics again. The results of our ablation studies in Section 4 show that the presence of these incorrect heuristics is harmful and leads to a severe reduction in training speed and generalization ability. Unlike COT and tool use methods, our method can be modified to avoid these problems: We describe in Future Work how the IGC could be trained during pretraining (Section 6).

Extensibility. Our method is specialized for arithmetic and trained on numbers up to a fixed length. By design, it can not help with any other type of task and it needs to be trained up to the largest number length we expect to see. We address this in the Limitations (Section 8) and explain why it is not much of a hindrance in practice. We also note that the basic design of our module could be generalized and adjusted to other tasks. We describe how in Future Work (Section 6).

6 Future Work

Using the IGC during Pretraining. As explained in Section 5, it would be preferable if the IGC was trained directly during pretraining. The difficulty here is that the IGC requires annotated training

Feature	Feature Type	Basic LLM	COT	Tool use	IGC
Addition & Subtraction	Capability	✗	✓	✓	✓
Multiplication & Division	Capability	✗	✗	✓	✓
Single-step Solutions	Efficiency	✓	✗	✗	✓
On GPU	Efficiency	✓	✓	✗	✓
Explicit Representations	Interpretability	✗	✓	✓	✓
Modularity	Interpretability	✗	○	○	✓
Internal	Integration	✓	✗	✗	✓
Pretraining	Integration	✓	✗	✗	✓
Dynamic Number Lengths	Extensibility	✓	✓	✓	✗
Generically Extensible	Extensibility	✓	✓	✓	○

Table 2: A comparison of different methods for solving arithmetic tasks. **Addition & Subtraction**. Can the model generalize on addition and subtraction tasks? **Multiplication & Division**. Can the model generalize on multiplication and division tasks? **Single-step Solutions**. Does the model run in constant time? **On GPU**. Does the model run entirely on the GPU? **Explicit Representation**. Can you tell when the model performs an arithmetic operation? **Modularity**. Can you tell how the model uses the results of an arithmetic operation? **Internal**. Is the arithmetic task solved entirely within latent variables, or does it need to generate output tokens? **Pretraining**. Is it possible to train the technique during pretraining, or is it only added afterwards? **Dynamic Number Lengths**. Can the model work with inputs of any size? **Generically Extensible**. Can the technique be extended to other tasks?

data for its auxiliary loss. Fortunately, it should suffice to intersperse the LLM's normal training data with small amounts of our annotated arithmetic-specific data. During training, the IGC would be executed for all data, but its Input Mapping submodule would only be trained on this annotated data. For data that lacks this annotation, the submodule simply does not receive a gradient. This makes the training process resilient to noise or even to mistakes in the remainder of the training data, so long as we are careful about the way our annotated data is constructed.

Word Problems. Word problems are different from pure arithmetic problems: In addition to the ability to perform arithmetic operations, they also require the model to identify the correct operator and inputs from the text. The IGC is only designed to perform arithmetic operations, so this is outside of its scope. However, it is relevant to investigate how effectively the IGC can be integrated into the LLM: When an LLM extracts an arithmetic task from a word problem, does it represent this task internally in a consistent manner, so that our Input Mapping submodule can access it effectively? We expect that this integration will be better if the IGC is trained during pretraining: The Input Mapping submodule generates gradients that encourage the rest of the LLM to extract numbers in an interpretable format, but these gradients are ignored if we only train the IGC and keep the LLM's parameter's frozen.

Generalizing the IGC Mechanism. The core component of the IGC is a non-differentiable calculator. From the perspective of the trainable components, this is a blackbox. That raises the question: What other mechanisms could be implemented in such a blackbox? For example, if we replaced it with a lookup table and adjusted the training mechanism appropriately, it would enable the model to perform database lookups or knowledge graph traversals in a single iteration and without generating any tokens. Such a mechanism could be used to improve upon the popular technique of Retrieval Augmented Generation (Lewis et al., 2020).

7 Conclusion

We introduce the Integrated Gated Calculator (IGC), a module that enhances an LLM with the ability to solve arithmetic tasks. We achieve near-perfect generalization on the BigBench Arithmetic benchmark, outperforming all existing models. In addition to its impressive performance, the IGC is also more practical than competing approaches: It avoids both the need for expensive tool calls, as well as lengthy and distracting Chains of Thought. We discuss how to integrate our module into an LLM during pretraining and explain why this could improve the model's ability even further, supported by empirical evidence from our ablation studies. Lastly, we note that our method could be generalized to integrate other types of non-differentiable tools into an LLM.

8 Limitations

Fixed Maximum Length. By construction, the IGC uses a fixed maximum input length for its numbers. This size can be arbitrarily large, but can not be adjusted later. To mitigate this issue, we can simply train the IGC with the largest size that we expect to see for the majority of practical tasks. That number may be surprisingly small: In the MATH dataset (Hendrycks et al., 2021), a standard benchmark for math word problems, 99% of numbers have four digits or less. In the rare cases when the model does have to deal with larger numbers, the LLM can still use the IGC as a very reliable approximator. It should also be noted that the IGC is not mutually exclusive with other methods: If the model encounters an arithmetic task that is both too large for the IGC *and* that requires high accuracy, it can resort to calling external tools. *This mirrors human behavior*: We learn to solve numbers up to a certain size in our heads. If we encounter tasks more complex than that, we either perform a rough calculation, or we resort to tools. Being able to solve simple arithmetic tasks in our heads without needing to use a tool is useful, but starting at a certain level of complexity it is no longer worth the effort.

References

Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. 2023. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774*.

BIG bench authors. 2023. Beyond the imitation game: Quantifying and extrapolating the capabilities of language models. *Transactions on Machine Learning Research*.

Tom B Brown. 2020. Language models are few-shot learners. *arXiv preprint arXiv:2005.14165*.

Wenhu Chen, Xueguang Ma, Xinyi Wang, and William W Cohen. 2022. Program of thoughts prompting: Disentangling computation from reasoning for numerical reasoning tasks. *arXiv preprint arXiv:2211.12588*.

Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, et al. 2023. Palm: Scaling language modeling with pathways. *Journal of Machine Learning Research*, 24(240):1–113.

Hyung Won Chung, Le Hou, Shayne Longpre, Barret Zoph, Yi Tay, William Fedus, Yunxuan Li, Xuezhi Wang, Mostafa Dehghani, Siddhartha Brahma, et al. 2024. Scaling instruction-finetuned language models. *Journal of Machine Learning Research*, 25(70):1–53.

Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Mark Chen, Heewoo Jun, Lukasz Kaiser, Matthias Plappert, Jerry Tworek, Jacob Hilton, Reiichiro Nakano, Christopher Hesse, and John Schulman. 2021. Training verifiers to solve math word problems. *arXiv preprint arXiv:2110.14168*.

Ning Ding, Yujia Qin, Guang Yang, Fuchao Wei, Zonghan Yang, Yusheng Su, Shengding Hu, Yulin Chen, Chi-Min Chan, Weize Chen, et al. 2022. Delta tuning: A comprehensive study of parameter efficient methods for pre-trained language models. *arXiv preprint arXiv:2203.06904*.

Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Amy Yang, Angela Fan, et al. 2024. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783*.

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

Luyu Gao, Aman Madaan, Shuyan Zhou, Uri Alon, Pengfei Liu, Yiming Yang, Jamie Callan, and Graham Neubig. 2023. Pal: Program-aided language models. In *International Conference on Machine Learning*, pages 10764–10799. PMLR.

Lee Garreth. 2024. Tweet. https://x.com/garrethleee/status/1860039446311371132. A tweet about a paper, not yet published as of the time of this writing.

Dan Hendrycks, Collin Burns, Saurav Kadavath, Akul Arora, Steven Basart, Eric Tang, Dawn Song, and Jacob Steinhardt. 2021. Measuring mathematical problem solving with the math dataset. *NeurIPS*.

Neil Houlsby, Andrei Giurgiu, Stanislaw Jastrzebski, Bruna Morrone, Quentin De Laroussilhe, Andrea Gesmundo, Mona Attariyan, and Sylvain Gelly. 2019. Parameter-efficient transfer learning for nlp. In *International conference on machine learning*, pages 2790–2799. PMLR.

Shima Imani, Liang Du, and Harsh Shrivastava. 2023. Mathprompter: Mathematical reasoning using large language models. *Preprint*, arXiv:2303.05398.

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

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

Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, et al. 2020. Retrieval-augmented generation for knowledge-intensive nlp tasks. *Advances in Neural Information Processing Systems*, 33:9459–9474.

Tiedong Liu and Bryan Kian Hsiang Low. 2023. Goat: Fine-tuned llama outperforms gpt-4 on arithmetic tasks. *Preprint*, arXiv:2305.14201.

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

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

Changle Qu, Sunhao Dai, Xiaochi Wei, Hengyi Cai, Shuaiqiang Wang, Dawei Yin, Jun Xu, and Ji-Rong Wen. 2024. Tool learning with large language models: A survey. *arXiv preprint arXiv:2405.17935*.

Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. 2023. Toolformer: Language models can teach themselves to use tools. *Preprint*, arXiv:2302.04761.

Avijit Thawani, Jay Pujara, Pedro A Szekely, and Filip Ilievski. 2021. Representing numbers in nlp: a survey and a vision. *arXiv preprint arXiv:2103.13136*.

Romal Thoppilan, Daniel De Freitas, Jamie Hall, Noam Shazeer, Apoorv Kulshreshtha, Heng-Tze Cheng, Alicia Jin, Taylor Bos, Leslie Baker, Yu Du, et al. 2022. Lamda: Language models for dialog applications. *arXiv preprint arXiv:2201.08239*.

Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, et al. 2023. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971*.

Alex Wang, Amanpreet Singh, Julian Michael, Felix Hill, Omer Levy, and Samuel R. Bowman. 2019. Glue: A multi-task benchmark and analysis platform for natural language understanding. *Preprint*, arXiv:1804.07461.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed Chi, Quoc Le, and Denny Zhou. 2023. Chain-of-thought prompting elicits reasoning in large language models. *Preprint*, arXiv:2201.11903.

Zheng Yuan, Hongyi Yuan, Chuanqi Tan, Wei Wang, and Songfang Huang. 2023. How well do large language models perform in arithmetic tasks? *arXiv preprint arXiv:2304.02015*.

A Tokenization

The Problem. We experimented with different ways to construct the input and output format of our module to improve learning speed and generalization. The basic goal is that similar tokens must be mapped to similar internal representations, for both the input and the output of our module. The main hindrance to this is that tokenization groups multiple digits together into single tokens.

Analysis. We have empirically analyzed the way tokenization works in our model: Llama 3.1 8B uses a tokenization method that consistently parses numbers from left to right and groups digits together in groups of three, unless there are only two or one digit left. It also conveniently tokenizes in such a way that these 3-digit tokens do not contain non-numeric characters. When read from the left, the tokens are always multiples of 3 digits, which makes it predictable which part of which token corresponds to which significant digit. For example, the 4th most significant digit always maps to the second token. Only the three least significant digits have any ambiguity. In contrast, when you read from the right, then figuring out which token a digit belongs to depends on the length of the number, which is non-local information and therefore much harder to learn. Table 3 illustrates this.

Consequences for the Architecture. The left-aligned format requires more work to implement because the position of the digit no longer matches the digit's significance. Adjusting the architecture to work with left-aligned numbers requires two small changes: Firstly, each of the digits now needs to be represented as a classifier with eleven options instead of ten: One for each possible digits and one for the special placeholder symbol (an asterisk in Table 3). Secondly, extracting the number from a left-aligned representation requires an additional step compared to the right-aligned representation: We need to know the index of the first placeholder and shift the number accordingly to ensure we get the right magnitude. This is demonstrated by the code in Section B.

Results. We tried a right-aligned version of our architecture as well. The difference in performance was significant and in many cases the right-aligned variant failed to converge or to improve upon the performance of a finetuned model at all.

Other Tokenization Methods. The left-aligned architecture described here works well because of the way Llama 3.1 tokenizes the data. Other tokenization models may have different requirements and work better with other types of architectures. We have written code to automatically analyze the distribution of tokens when generating training data. This code can help you to determine the optimal architecture to use. We will make it available upon acceptance of the paper.

B Code

In the following we provide python code for the three stages of our module.

B.1 Code of the Input Mapping and Auxiliary Loss

(We will provide code upon acceptance of the paper. The code in its current stage still needs to be optimized and documented for other researchers.)

B.2 Code for Emulating a Calculator

The calculator emulation works in three steps:

- Translate left-aligned digits to numbers. We take inputs in a fixed-length categorical format that represents digits and translate this into a single number.
- Perform a calculation on the numbers. To ensure that the module can be trained on batches of data, we perform all four operations in parallel and then take a weighted average.
- Translate the result back into left-aligned digits.

Numerical Accuracy. One important implementation detail to be aware of is the numerical accuracy of torch tensors. We only calculate up to 10 digits because that is all that is needed for the benchmark. If the calculator should be able to handle longer numbers that do not fit into a torch.int64 datatype, the calculation must be broken down into several smaller steps. This leads to code that is longer and harder to understand, although the operations are still fairly simple.

(We will provide code upon acceptance of the paper. The code in its current stage still needs to be optimized and documented for other researchers.)

B.3 Code of the Output Mapping

(We will provide code upon acceptance of the paper. The code in its current stage still needs to be optimized and documented for other researchers.)

Number	Tokenization	Right-aligned	Left-aligned
1234567890	"123"456"789"0"	123456789 0	123456789 0
123456789	"123"456"789"	0123456789	123456789 *
12345678	"123"456"78"	0012345678	12345678 * *
234567890	"234"567"890"	0234567890	234567890 *
34567890	"345"678"90"	0034567890	34567890 * *

Table 3: Examples of numbers, how they get tokenized, and two different variants for representing them internally. Both variants use a fixed size of 10 digits for illustration. The right-aligned variant is the intuitive, default format, and puts the *least* significant digit at a fixed index. In contrast, the left-aligned format puts the *most* significant digit at a fixed index. The right-aligned format can be padded with zeros, but the left-aligned format requires a special symbol to mark where the number ends. The underlining in the two formatted numbers indicates which digits get grouped together into the same token. Note that the underlining is the same for all examples in the left-aligned format, but is inconsistent in the default format. This makes the mapping between the tokens and the format much easier to learn for the left-aligned format. The last two lines illustrates that removing digits from the left instead of the right as in the examples above does not lead to the reverse phenomenon: The tokenization is now different, so it doesn't help that the right-aligned representation is similar.