Teaching Arithmetic to a Small Language Model:

A Chain-of-Thought Training Study with NanoGPT

Project Report

Phu Nguyen, Spring 2026

Abstract

This report documents a self-directed research project investigating whether a small transformer language model trained from scratch can learn to perform multi-digit integer addition, and whether introducing chain-of-thought (CoT) reasoning at training time can improve generalization beyond the training distribution. Using Andrej Karpathy's NanoGPT as the base architecture, the project progressed through several phases: baseline evaluation of pretrained Llama models, iterative training of NanoGPT on addition datasets of increasing complexity, redesign of the loss function to eliminate noise from non-answer tokens, and finally an attempt to incorporate CoT intermediate reasoning steps. The model achieved near-perfect accuracy on up to 4-digit addition within its training distribution, but consistently failed to generalize to 5-digit addition. The CoT implementation improved the quality of training but did not yet produce reliable out-of-distribution generalization. The results raise fundamental questions about whether small transformers can learn the true algorithm of addition or merely memorize input-output patterns.

1. Introduction

Arithmetic is one of the simplest and most well-defined tasks that a language model could be asked to perform. Yet despite the apparent simplicity of addition, transformer-based language models are known to struggle with multi-digit arithmetic, particularly when the numbers involved exceed the range seen during training. This failure mode is especially stark in small models, which lack the capacity to store large lookup tables implicitly.

This project was motivated by a central question: can a small language model learn the rule of addition, or does it only learn to pattern-match within its training distribution? A secondary question followed naturally: if standard training fails to generalize, can chain-of-thought reasoning, in which the model is trained to produce intermediate carry-and-sum steps before giving a final answer, enable true algorithmic generalization?

To investigate these questions, this project used NanoGPT, a minimal GPT-2-style implementation by Andrej Karpathy. NanoGPT was chosen because its simplicity makes it easy to modify the training loop, the data pipeline, and the loss function, which proved essential at several points in the project.

The project spanned approximately seven weeks and is organized in this report as follows: Section 2 describes the baseline experiments with pretrained Llama models. Section 3 covers the NanoGPT baseline training. Section 4 details the redesign of the loss function. Section 5 reports the move to 4-digit addition and dataset control. Section 6 describes the chain-of-thought implementation attempts. Section 7 synthesizes results and discusses limitations. Section 8 outlines future directions.

2. Baseline: Evaluating Pretrained Llama Models

2.1 Motivation

Before training from scratch, the project first evaluated two off-the-shelf models, Llama-3.2-1B and Llama-3.2-1B-Instruct, on addition problems ranging from 1 to 10 digits. The goal was to establish a performance ceiling for small pretrained models and to understand where accuracy degrades.

2.2 Methodology

Each digit length was tested with 100 random problems. For the instruct variant, prompts followed the standard instruction format. For the base (non-instruct) model, a simple prompt of the form "a + b =" was used. Due to Google Colab GPU usage limits, experiments were run locally on a personal machine.

2.3 Results

Digit Length	Accuracy (%)
1	96.0
2	96.0
3	59.0
4	29.0
5	15.0
6	3.0
7	2.0
8	1.0

Table 1. Llama-3.2-1B-Instruct accuracy on n-digit addition (100 trials each).

2.4 Discussion

The instruct model achieves near-perfect accuracy on 1 and 2-digit addition but degrades sharply starting at 3-digit problems, dropping from 59% at 3 digits to effectively zero by 7 digits. This pattern is consistent with the hypothesis that the model is retrieving memorized answers rather than applying an algorithm, since memorized answers would be reliable for small numbers and break down for larger ones.

The non-instruct base model performed poorly across all digit lengths, mostly repeating the prompt rather than generating an answer. This suggests that the instruction-following fine-tuning incorporated in the instruct variant already encodes some form of arithmetic reasoning, possibly through chain-of-thought data seen during fine-tuning.

These results motivated switching to NanoGPT, where the training process can be fully controlled from scratch, allowing the project to study the learning dynamics more directly.

3. NanoGPT Baseline Training

3.1 Experimental Setup

NanoGPT was trained on a dataset of randomly generated addition problems using the prompt format "a + b = c\n". The training dataset covered problems with summands of up to 3 digits. A custom Python evaluation script was written to measure accuracy independently of the training loss, since training loss alone does not reflect how often the model produces the correct final answer.

The evaluation script generates test problems using a fixed random seed to ensure reproducibility across runs. Each digit-length bucket is evaluated on 500 problems. Testing halts early for a given digit length if the model achieves 0% accuracy on three consecutive buckets, reflecting the expectation that accuracy will not recover once it fails.

3.2 Padding with Leading Zeros

An initial finding was that the model frequently failed on problems involving the digit 0, even for small numbers. To address inconsistent input lengths, summands were padded with leading zeros so that all inputs within a training batch had the same width. For example, a 3-digit training run would represent 7 + 8 as "007 + 008 = 015".

This alignment improved accuracy significantly. A further test padded all summands with one extra leading zero beyond the maximum in the training data, to probe whether the model could generalize to 4-digit addition from 3-digit training data. This extrapolation failed, confirming the model's reliance on pattern matching within the training distribution.

Configuration	1-digit	2-digit	3-digit	4-digit
No padding (v1)	94.6%	90.8%	95.8%	0.0%
Padded to max width	100.0%	94.2%	98.4%	0.0%
Padded with extra zero	100.0%	99.6%	99.6%	0.0%

Table 2. NanoGPT accuracy under different input padding strategies (500 trials per digit length).

4. Redesigning the Loss Function

4.1 Problem Identification

After achieving strong within-distribution accuracy with padding, attention turned to the training loss. It was observed that the standard cross-entropy loss calculated over all tokens introduces several sources of noise that prevent the model from learning the answer-generation task cleanly.

Three specific issues were identified:

- When a training batch begins mid-problem (e.g., starting at the '+' token), the model cannot reasonably predict the tokens that follow, making the loss artificially high for those samples.
- After generating the answer to one problem, the model is penalized for failing to predict the beginning of the next problem, which is a structurally different task.

- Tokens like '+', '=', and '\n' are trivially predictable from the fixed format, so the model can reduce its loss by learning these structural tokens without improving its arithmetic ability.

4.2 Solution

The solution was to restructure the data and the loss calculation so that only the answer tokens contribute to the loss. Since all problems in the 3-digit training set are exactly 16 tokens long in the format "#### + #### = \n####\n", the data loader was modified to always start batches at multiples of 16, ensuring each training sequence contains exactly one complete problem.

Loss masking was applied by setting the target values for all tokens from the start of the sequence up to and including the first '\n' (i.e., everything except the answer digits and the final newline) to -1. PyTorch's cross-entropy loss ignores index -1 by default, so only the answer portion of each problem contributes to gradient updates.

4.3 Results

With the new loss function, training loss converged to approximately zero. On the same test data used in the previous experiments, the new loss function achieved 100% accuracy for 1, 2, and 3-digit addition (up from 94.2% and 98.4% for 2 and 3 digits respectively). This confirmed that the loss noise was a meaningful obstacle to learning.

Digits	Previous Accuracy (%)	After Loss Masking (%)
1	100.0	100.0
2	94.2	100.0
3	98.4	100.0
4	0.0	0.0

Table 3. Effect of loss masking on NanoGPT addition accuracy.

5. Scaling to 4-Digit Addition

5.1 Dataset Management

Extending training to 4-digit addition increases the problem space to 10,000^2 = 100 million unique combinations. At this scale, it becomes important to ensure that the train, validation, and test sets are strictly disjoint: any overlap between training data and test evaluation would inflate accuracy measurements and obscure the model's true generalization ability.

A significant portion of development time was spent implementing an efficient disjointness check capable of operating at a scale of several hundred thousand samples. In addition, the test set was augmented with a set of manually designed edge cases (problems involving carries, zeros, and near-boundary values) to specifically probe the model's logical coverage.

5.2 Results

After retraining with controlled datasets and different train/validation/test ratios, the model was evaluated against approximately 10 million test problems. It achieved nearly 100% accuracy on 4-digit addition, including problems with long carry chains, which had been anticipated as a potential failure point.

However, accuracy on 5-digit addition remained exactly 0%. This sharp boundary confirms that the model is not learning the algorithm of addition but rather fitting the specific distribution it was trained on. The model has no mechanism to generalize its learned behavior to numbers slightly outside its training range.

6. Chain-of-Thought Implementation

6.1 Motivation

The core hypothesis behind chain-of-thought training is that if the model is required to produce intermediate reasoning steps alongside its answer, it will be forced to learn a general procedure rather than simply memorize input-output pairs. For multi-digit addition, a natural CoT format involves reversed-digit sequences, where the model processes digits from least significant to most significant and explicitly carries values between positions.

6.2 First Attempt (March 22)

An initial attempt recoded the data preparation pipeline to generate reversed-digit sequences as CoT traces. The new training data included explicit intermediate steps. However, the model failed to learn the correct reasoning syntax and produced incoherent outputs. Subsequence analysis identified two likely causes: the original NanoGPT training file lacked loss masking to focus the model on the reasoning steps, and training sequences were sometimes cut off mid-problem due to block size misalignment.

6.3 Second Attempt (Late March)

A revised CoT implementation addressed both failure modes. Multiple CoT data formats were tested to explore different ways of encoding intermediate reasoning. The training file was updated to ensure batches only contain complete problem sequences, consistent with the earlier loss masking work.

With these fixes, the model performed well on addition problems within the trained distribution (summands of up to 4 digits). However, it continued to fail on 5-digit addition, showing no meaningful extrapolation beyond the training range. An expected baseline of 10 to 20% out-of-distribution accuracy was not observed.

6.4 Open Question

At the conclusion of this project, it remains uncertain whether NanoGPT is capable of learning true algorithmic addition at all, given the current architecture and training setup. The model's behavior does not change meaningfully across configurations, suggesting the bottleneck may be architectural rather than a matter of training procedure.

7. Consolidated Results and Discussion

7.1 Summary of Accuracy Across Phases

Phase	Model / Config	1-dig	2-dig	3-dig	4-dig	5-dig
Baseline	Llama-3.2-1B-Instruct	96%	96%	59%	29%	15%
NanoGPT v1	No padding	94.6%	90.8%	95.8%	0%	0%
NanoGPT v2	Zero-padded	100%	94.2%	98.4%	0%	0%
NanoGPT v3	Loss masking	100%	100%	100%	0%	0%
NanoGPT v4	4-digit training	100%	100%	100%	~100%	0%
NanoGPT CoT	Chain-of-thought	100%	100%	100%	~100%	0%

Table 4. Accuracy by phase across all digit lengths tested.

7.2 Key Findings

- Pretrained small models degrade rapidly with digit length, consistent with memorization rather than algorithmic reasoning.
- NanoGPT can achieve near-perfect accuracy within its training distribution through careful data formatting and loss masking.
- Padding inputs to a uniform width and aligning training batches to complete problems both meaningfully improve accuracy.
- Loss masking to exclude non-answer tokens from gradient updates was the single most impactful change, eliminating noise and enabling convergence to near-zero loss.
- Scaling from 3-digit to 4-digit training was straightforward once dataset disjointness was ensured.

7.3 Limitations

The primary limitation of this work is the absence of out-of-distribution generalization. The model's sharp accuracy drop at the boundary of its training distribution, going from approximately 100% to exactly 0% when digit length increases by one, is a strong indicator of pattern memorization rather than rule learning.

A secondary limitation is that the CoT format used was not rigorously evaluated against multiple baselines. It is possible that a different CoT encoding, for example one that more explicitly represents carry propagation, would yield better results. This remains an open direction.

8. Future Work

Based on the findings and open questions raised by this project, several directions are worth pursuing:

- Systematic scaling study: evaluate how accuracy scales as a function of training data volume, number of training iterations, and model size, under fixed computational constraints.
- Alternative CoT encodings: test CoT formats that make carry propagation more explicit, such as digit-by-digit column addition traces.
- Architectural modifications: investigate whether positional encoding changes or attention pattern modifications could enable better length generalization.
- Scratchpad approaches: explore training the model to use intermediate 'scratch' tokens before committing to an answer, similar to methods used in larger reasoning models.
- Comparison with larger models: repeat the 4-digit and CoT experiments with a larger version of GPT-2 to assess whether scale alone can solve the generalization problem.

9. Conclusion

This project successfully built a NanoGPT model that achieves near-perfect accuracy on addition problems within its training distribution, demonstrating that small transformers can memorize arithmetic up to 4 digits with proper training. Key technical contributions include a carefully controlled disjoint dataset pipeline, a custom loss masking scheme that improved 2 and 3-digit accuracy to 100%, and multiple chain-of-thought training experiments.

However, the model's complete inability to generalize beyond its training range, even with chain-of-thought reasoning, suggests that NanoGPT may be fundamentally limited in its ability to learn arithmetic as an algorithm. Resolving this question, and determining whether it is a matter of architecture, training data, or reasoning format, remains the central open problem for future work.