What Tiny GPTs Can Represent and What They Actually Learn

Niloy Saha

Overview

In this project, I studied a simple question: what is the difference between a transformer that can represent a computation and a transformer that actually learns that computation from data? I explored this question by working with small GPT-style models in two different ways. First, I manually designed tiny transformers for tasks such as copy, reverse, arithmetic, and simple next-token prediction. Second, I trained similar small GPTs on the same kinds of tasks and tested whether they generalized to unseen examples. My reason for combining these two approaches was to separate two issues that are often treated as the same. I recognized that a trained model can fail because the architecture is too weak, but it can also fail because training does not discover the right computation from limited data. My hand-designed models allowed me to answer the first question directly: can this kind of transformer do the task at all? My trained models answered the second: if the model can do the task in principle, will gradient-based learning actually find that solution?

Hand-Designed GPTs

I began my project by hand-designing transformer models for copy and reverse tasks. At first, I built single-pass models that took in the full input and produced the full output at once. After receiving feedback, I shifted my focus to the more realistic autoregressive setting, where the model generates one token at a time like a real GPT. I found that this change made my project stronger because it brought my hand-designed models much closer to how language models are actually used. I then built hand-designed autoregressive models for copy, reverse, binary addition, ternary addition, decimal addition, and a simple Markov next-token process. Through these models, I showed that a very small decoder-only transformer can be programmed directly to solve each of these tasks. My copy and reverse models worked exactly, and I found that my arithmetic models also worked once I designed the internal representations carefully. For example, my decimal addition model reached perfect accuracy on all tested one-digit sums. The most important lesson I learned from the hand-designed work came from arithmetic. In my first attempt, I tried to store the result of a computation as a large value in a single hidden dimension, but I found that this failed because LayerNorm removed the magnitude information the model needed. In my experiments, this meant that different sums became almost indistinguishable. I solved this by encoding information as a pattern across dimensions rather than as a single scalar signal. This fix made my arithmetic models work and showed me that LayerNorm is an important constraint on how transformer computations must be represented. This part of my project established an important baseline; I confirmed that the small GPT architecture was expressive enough to solve these tasks, so I knew that if my trained models later failed, I could not automatically blame a lack of model capacity.

Trained GPT Experiments

After I established what the architecture could represent, I turned to the learning problem. I trained small GPTs on subsets of copy, reverse, and addition tasks and then tested them on held-out examples that the models had never seen. My goal was to see whether the models learned the actual rule or mainly memorized the training set. In my first trained experiments, I used small datasets. On the copy task, I found the model reached 16 out of 18 correct on held-out examples (88.9%). On reverse, I observed that held-out performance was much lower at 11 out of 18 (61.1%).

On one-digit decimal addition, I found that held-out accuracy was only about 16% to 22%. This produced an initial pattern that seemed clear to me: copy generalized reasonably well, reverse generalized much less reliably, and addition generalized poorly. At that stage, I could have concluded that task complexity alone explained my results. However, because I had already hand-programmed models to solve all of these tasks, I realized that the more interesting question was whether the low performance I observed was about task impossibility or simply about the amount of data I provided during training.
In the baseline comparison, the hand-designed models achieved perfect performance on both seen and unseen inputs, while the trained models matched them on seen data but dropped sharply on unseen examples, especially for reverse and addition.

Extension Results
This extension was partly motivated by reading Transformers Can Do Arithmetic with the Right Embeddings (McLeish et al.), which suggested that poor arithmetic results might come from the training setup and data regime rather than from a hard architectural limit.
To test this, I extended my experiments by increasing both the training data and the difficulty of the tasks. I ran longer versions of copy and reverse, increased the training size for one-digit addition, and tested two-digit and three-digit addition. I also examined a simple Markov prediction task. These larger experiments changed my conclusion in an important way. I found that copy reached 100% on larger variants. Reverse, which had looked weaker in my original small-data setting, also reached 100% once I trained it with enough examples. I observed that one-digit addition improved substantially with more data; my two-digit addition experiments reached 88.7% on held-out examples, and my three-digit addition reached 41.0%. In my Markov experiment, I found the trained model learned the transition structure closely enough to reach a KL divergence of about 0.002.
These results mattered to me because they changed how I understood the earlier failures. I realized that reverse did not fail because small transformers cannot reverse sequences, nor did addition fail because of architectural incapacity. Instead, I found that both tasks were simply much harder for training to learn from small datasets. Once I increased the data, performance improved substantially. My experiments showed that the limiting factor was often not representational power, but whether optimization could recover the right computation from the examples I provided.

Main Conclusion
The clearest conclusion I drew from this project is that there is a real gap between what a tiny transformer can represent and what training will actually learn from limited data. I proved with my hand-designed models that small GPTs can implement these tasks exactly. My trained models showed me that learning those same computations is a separate problem that depends on data and task structure. This was especially visible when I compared my baseline and extension experiments. If I had stopped after my first runs, I might have concluded that reverse and addition were simply less learnable than copy. But my larger-data experiments showed a more accurate picture: I found that the same small GPT family can learn those tasks much better with a stronger training regime. I believe my project shows that studying hand-designed and trained transformers side by side is a useful way to understand model behavior. In my work, the hand-designed models made the computation explicit, while the trained models showed how much of that computation learning actually recovers. Taken together, they provided me with a clearer answer than either approach alone. I found that tiny GPTs can do more than weak training results suggest, but they only learn

those rules reliably when the data regime is large enough. This gap between representational ability and learned generalization is the main result of my project.