# Fine-Tuning GPT-2 on Arithmetic

**Noah Topper**
University of Central Florida
noah.topper@knights.ucf.edu

## Abstract

The step from GPT-2 to GPT-3 brought significant improvements. With just a few examples and no fine-tuning, GPT-3 performs well on various natural language tasks. One such task is basic arithmetic. In particular, GPT-3 is highly accurate at two- and three-digit addition and subtraction, while GPT-2 has little to no capability on this task. I test if it is possible to achieve a similar level of performance with GPT-2 via fine-tuning on arithmetic problems.

## 1 Introduction

The step from GPT-2 to GPT-3 involved moving from a 1.5 billion parameter transformer model to one with 175 billion parameters (Brown et al., 2020). This led to significant improvements in few-shot (or zero- and one-shot) performance on various natural language tasks, where the examples are provided as prompts to the system. GPT-3 performs well on translation, question-answering, unscrambling words, arithmetic, and more.

In particular, in the few-shot case, GPT-3 achieves 100% accuracy on two-digit addition, 98.9% accuracy on two-digit subtraction, 80.4% accuracy on three-digit addition, and 94.2% accuracy on three-digit subtraction. Meanwhile, GPT-2 achieves around 5% accuracy on these tasks (Brown et al., 2020).

I test if it is possible to achieve a similar level of performance as GPT-3 using GPT-2 with fine-tuning on arithmetic problems. (Brown et al., 2020) states that only a small number of problems in the test sets also appeared in the prompts. Further, given how many three-digit addition problems there are, it is hard to believe that GPT-3 has seen most of them enough times in its initial training to memorize them. So it seems that in some limited sense, GPT-3 has learned the concept of addition.

I am curious to see what level of data is required for GPT-2 to perform on a similar level, if it even can. Does the model contain enough expressivity to acquire the addition concept, or must we provide enough data for GPT-2 to simply memorize the relevant information? We will see.

## 2 Methods

To fine-tune GPT-2, I used the HuggingFace API (Wolf et al., 2019). HuggingFace contains a large number of pre-trained language models, including GPT-2. Any of these models may be loaded for a particular task, such as question-answering, summarization, classification, and more. An appropriate head is added to the network, if necessary. This head has fresh weights, while the pre-trained weights of the network are left in place. None of these weights are frozen. That is, all will be subject to further change during fine-tuning.

For our task, we select causal language modeling. This is the task of predicting the next word in a sentence, given just the previous words. This task is what GPT-2 was originally trained on, and so does not require a new head for the network. A model that is good at predicting next words can then also be used for text generation, by selecting words according to its predictions. Good text generation can be very general, capable of summarizing, answering question, and more, given the right prompt.

Text generation like this can then be specialized to a particular domain by fine-tuning on a special data set. In particular, we may randomly generate a set of examples such as "12 + 10 = 22". We preprocess this data set and fine-tune the model in a typical training loop. Periodically, we evaluate it on a test set of similar randomly generated examples. This evaluation tells us how good the model is at predicting the answers to questions it likely did not see in the training set.

The loss on the test set is computed automatically, but we also want an additional metric: accuracy. To test the model's accuracy, we feed it each test example without the final token. We get the model to greedily generate one more token and compare this to the true answer. In this way, we can check the model's accuracy over the whole test set.

There are some subtleties here. In fact, the GPT-2 tokenizer sometimes splits numbers into multiple tokens, so it is not quite right to generate just one more token. First, we must pad all the sequences in the training and test sets to be the same length. Then, when checking accuracy, we allow the generated text to go all the way up to this maximum length, allowing for the possibility that the answer is multiple tokens. Finally, we pad the output, if necessary, and compare to the true answer.

Additionally, we must extract a set of prompts from the test set, with just the questions, not the answers (e.g. "12 + 10 ="). These prompts will be of varying lengths, again since some numbers will require multiple tokens. So some additional preprocessing is required to trim each example in the test set down to the correct length and separate them into batches of the same length.

(Brown et al., 2020) uses 2,000 examples in each of its test sets, so we follow suit. In particular, for two-digit addition, we sample two numbers $a, b \in [0, 100)$ uniformly at random and write $a + b = \{a + b\}$ to a text file, where $\{a + b\}$ represents the sum of $a$ and $b$ evaluated. We do this 2,000 times to get a test set. Two-digit subtraction, three-digit addition, and three-digit subtraction are all similar.

As a preliminary experiment, I first tried fine-tuning on just two-digit addition and then just three-digit addition to get an idea of how much data is necessary for high performance in each case. I also wanted to compare to what happens when we later train on all the examples at once. 5,000 training examples appeared to be a good level for two-digit addition, and 30,000 examples was good for three-digit.

After that, I combined all the training sets into one, having 5,000 two-digit addition and subtraction problems each, and 30,000 three-digit addition and subtraction problems each, for 70,000 examples overall. I used four different test sets as described above, to evaluate accuracy separately in each case. Test loss is evaluated across all four sets combined, however, for 8,000 examples in total.

I trained with HuggingFace's default optimizer and learning rate scheduler, which is the Adam optimizer with weight decay regularization and a linear schedule. I used a batch size of 512 and evaluated every 50 batches, stopping early if test loss worsened for three evaluation steps in a row. Finally, I utilized Google Colab's faster GPUs using Colab Pro.

## 3 Results

Starting with the preliminary experiments, I first fine-tuned GPT-2 on a set of 5,000 two-digit addition problems and evaluated on a test set of 2,000 similar problems. I ran for a full 100 epochs, making 1,000 optimization steps in total (with batches of size 512). See the loss and accuracy curves in Figure 1 and Figure 2. Given the loss curve, we clearly reached the limits of what we are able to learn with this architecture and level of data. In the end, we achieved 96% accuracy on two-digit addition.

Next, I fine-tuned GPT-2 on 30,000 three-digits addition problems and evaluated on a test set of 2,000. I ran for 44 epochs, or 2,600 optimization steps total. After this, loss again stopped decreasing. See the results in Figure 3 and Figure 4. We achieved 85% accuracy on three-digit addition.

Compare to GPT-3, which achieves 100% and 80.4%, respectively. Thus, with this level of data, we are able to perform on par. Inspired by this, we combine our data into one big training set and test set as described before. During evaluation, we measure accuracy separately on all four test sets. I ran for 74 epochs before lost stopped decreasing, for around 10,100 optimization steps overall. See Figure 5 and Figure 6 for the results.

In the final figure, the top two lines represent two-digit addition (red) and subtraction (blue). The next line is three-digit addition (grey), and the last is three-digit subtraction (green). Overall, we achieve 99.8% accuracy for two-digit addition, 88.4% for three-digit addition, 99.2% for two-digit subtraction, and 89.2% for three-digit subtraction. Compare again to GPT-3, which respectively achieved 100%, 80.4%, 98.9%, and 94.2%. Again, we perform comparably.

## 4 Conclusion

So, it is indeed possible to fine-tune GPT-2 to perform about as well as GPT-3 on arithmetic, at least with two and three digits. The training dynamics here are quite interesting. As the overall training process ran, I grew skeptical that GPT-2 could learn each task simultaneously; three-digit arithmetic

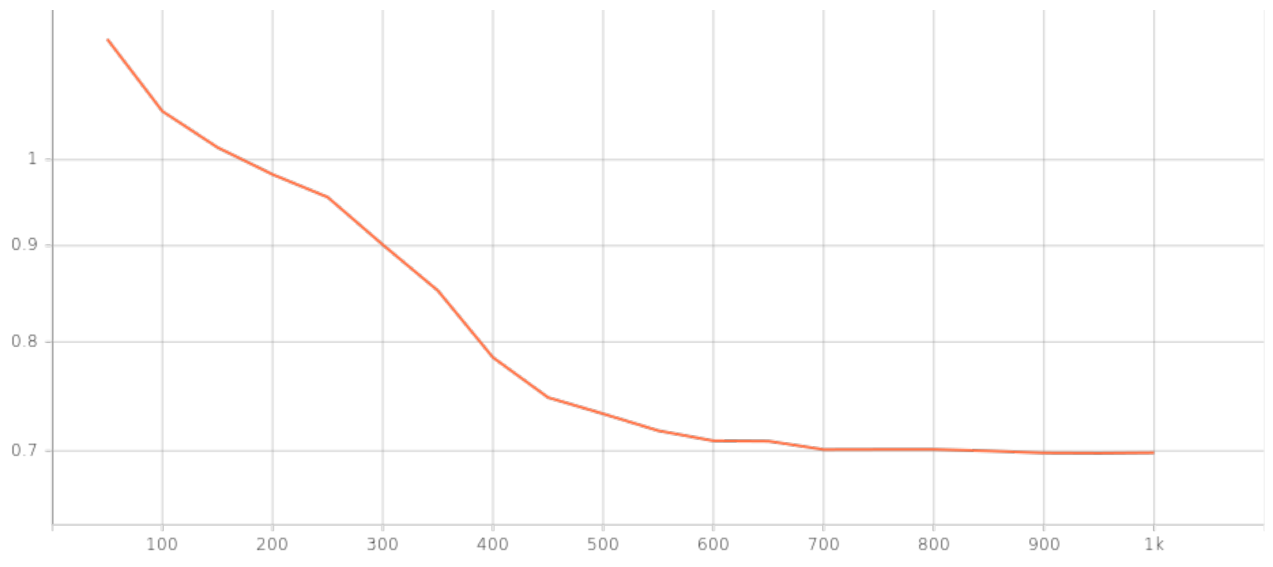

Figure 1: Steps vs test loss on two-digit addition

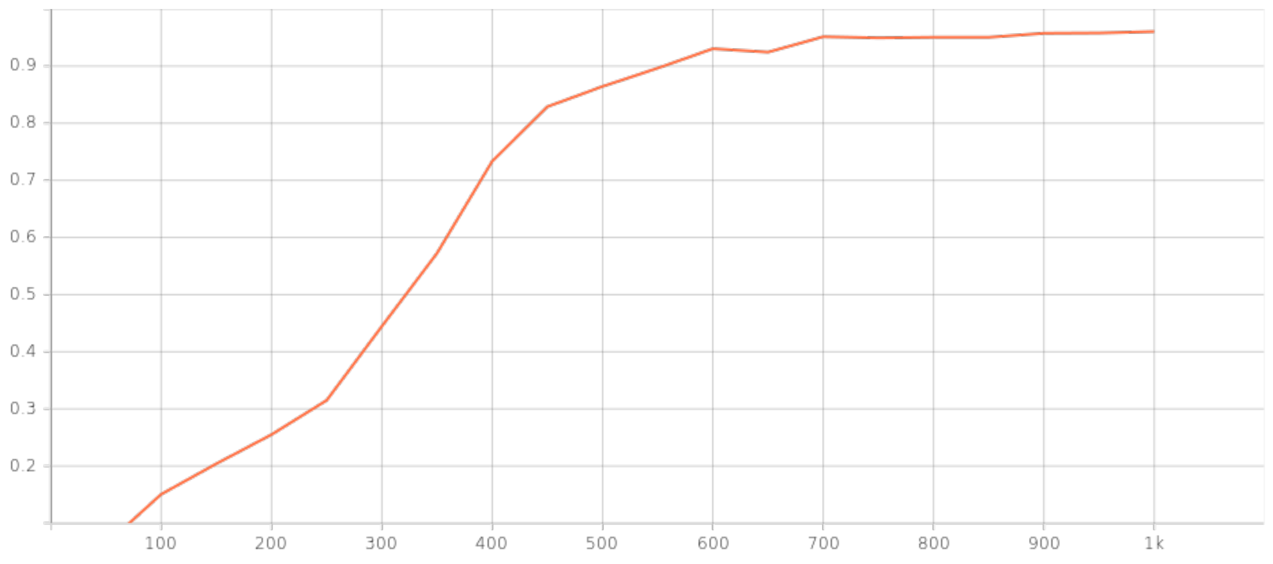

Figure 2: Steps vs test accuracy on two-digit addition

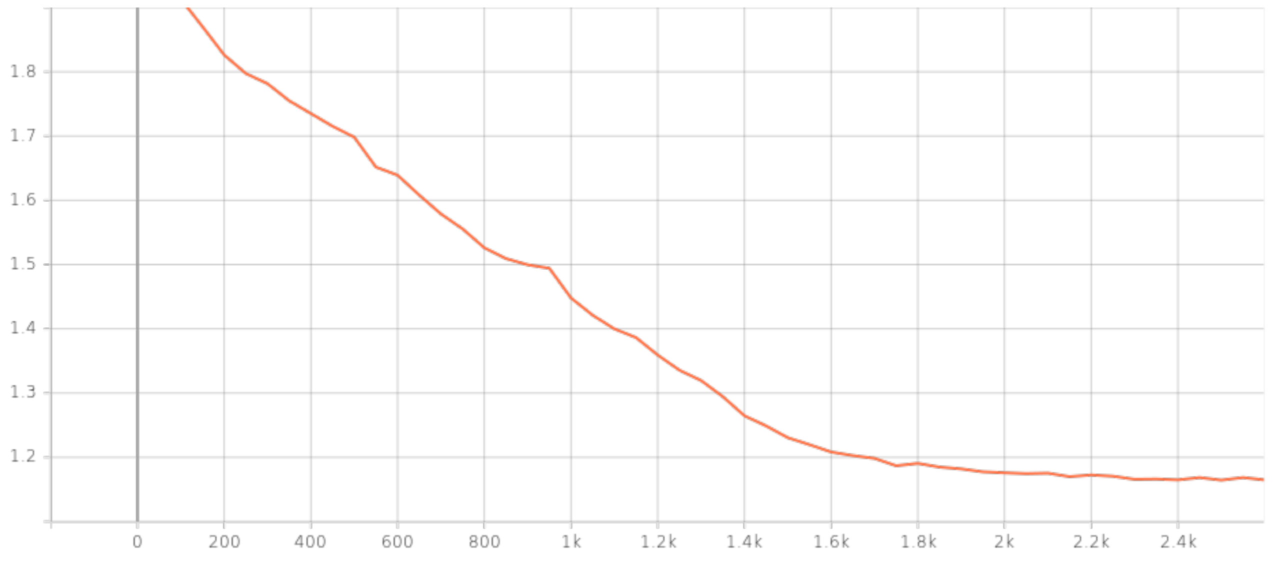

Figure 3: Steps vs test loss on three-digit addition

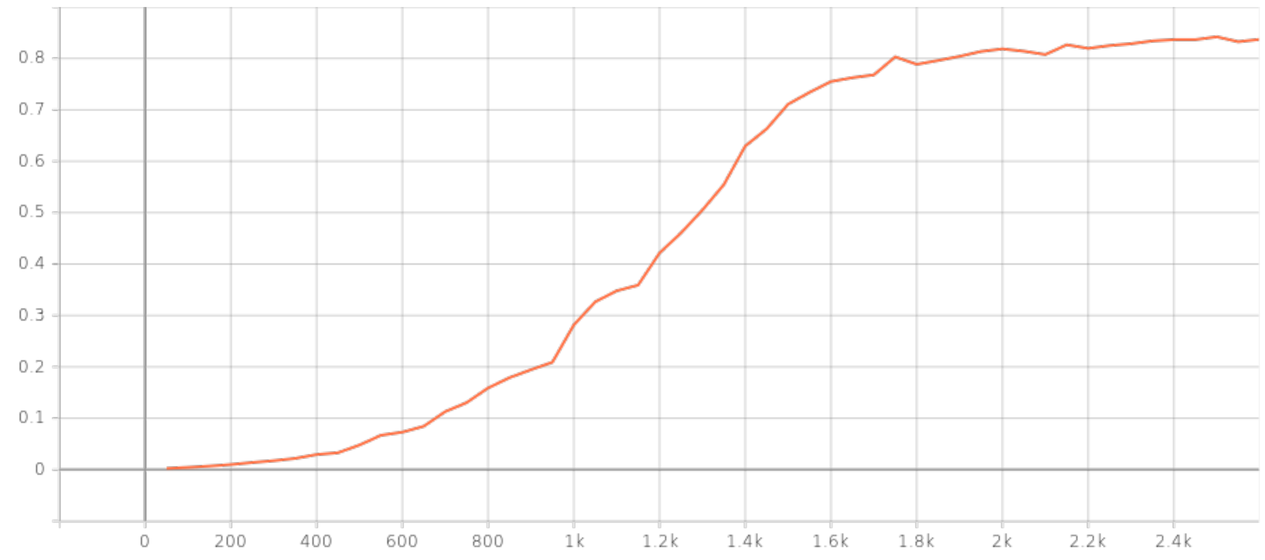

Figure 4: Steps vs test accuracy on three-digit addition

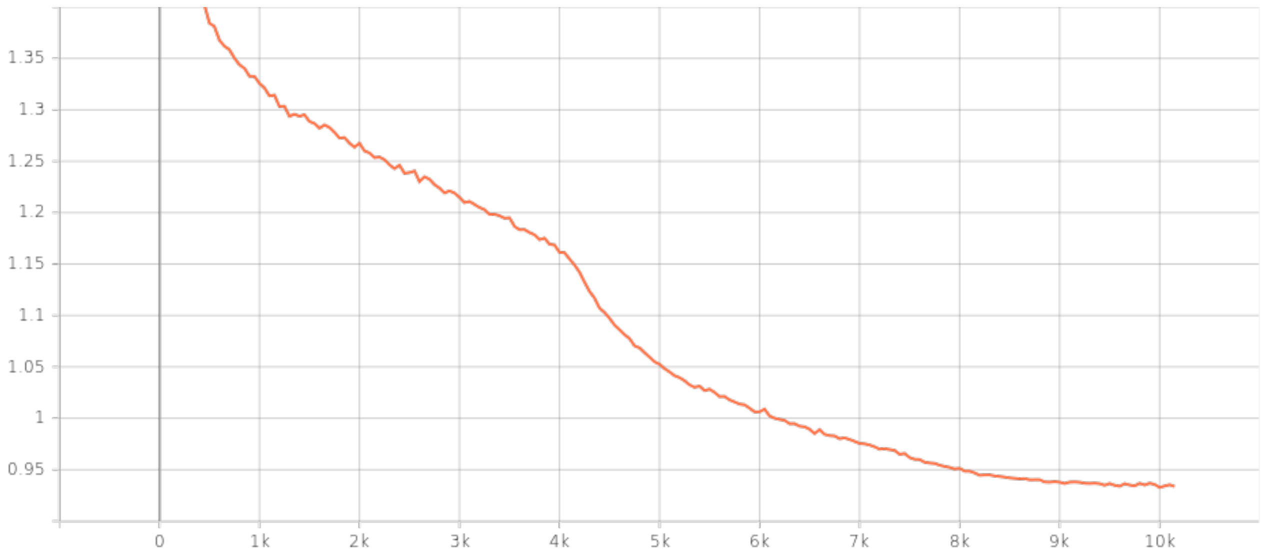

Figure 5: Steps vs test loss overall

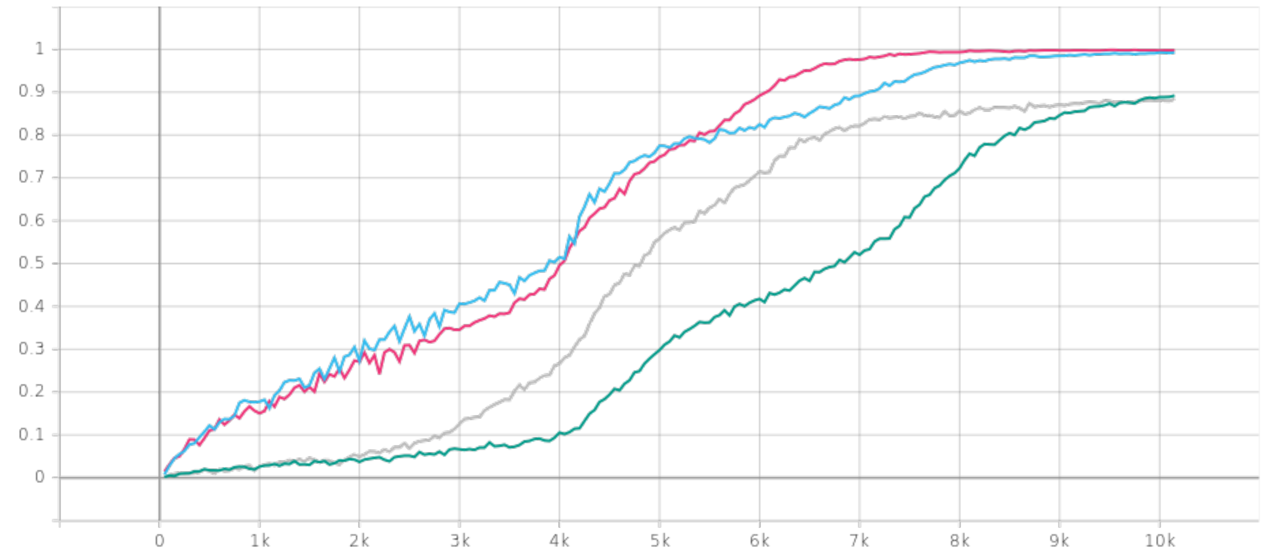

Figure 6: Steps vs test accuracy overall

was far behind for a significant portion of time.

However, around step 3,000, the accuracy on three-digit problems started to pick up slightly. At step 4,000, the accuracy on all fronts suddenly spiked. This may be where the problem "clicked" for the model and it began to understand the underlying mechanism, even sharing knowledge from one type of problem to another.

Note that there are around 1,000,000 possible three-digit addition problems. We only provided the model with 30,000, yet it was able to generalize to the others with high accuracy. Neural networks are universal function approximators, so there is no reason in principle our model could not learn to implement addition. Indeed, it should not be surprising if genuinely learning to implement addition is the most compact way for the network to achieve high accuracy.

On the other hand, it is often difficult for noisy, fuzzy systems like a neural network to implement precise computations, so it is hard to be certain what is going on. GPT-3 itself struggles beyond three digits, which makes it appear as though it has not fully grasped the concept. But on the third hand, many humans can do two-digit arithmetic in their head, but struggle with five digits. Our own biological neural networks have trouble implementing arithmetic precisely, yet we still have a concept of addition. Could GPT-2 and GPT-3 not be the same?

I suspect that GPT-3 really is implementing something like addition, but ultimately, neural networks are still too opaque to be sure. There is recent research in "circuits", which tries to reverse-engineer the algorithms networks implement by analyzing their weights and activations (Cammarata et al., 2020).

Up to now, most of this work has been in computer vision systems, finding things like Gabor filters in early layers and high-level concepts in later layers. For future research, it would be interesting to see if these methods could be extended to language models. If we could find the part of the transformer that is answering these arithmetic questions and pick it apart, it could be very illuminating.

## References

Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel Ziegler, Jeffrey Wu, Clemens Winter, Chris Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. Language models are few-shot learners. In *Advances in Neural Information Processing Systems*, volume 33, pages 1877–1901. Curran Associates, Inc.

Nick Cammarata, Shan Carter, Gabriel Goh, Chris Olah, Michael Petrov, Ludwig Schubert, Chelsea Voss, Ben Egan, and Swee Kiat Lim. 2020. Thread: Circuits. *Distill*. Https://distill.pub/2020/circuits.

Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, and Jamie Brew. 2019. Huggingface's transformers: State-of-the-art natural language processing. *CoRR*, abs/1910.03771.