Sourceformer: Tool Integration into Transformer Large Language Models Through Source Code

Eli Richmond
https://bloge.li

Nic Herndon
herndonn19@ecu.edu

Abstract *— LLMs have been shown to use tools well. By allowing specific tools to increase the capabilities that LLMs struggle with, these models can become much more useful. Previous works use handcrafted examples of simple tool use during self-training. This type of data generation and training is a great compliment to the self-supervised nature of LLMs because most of the generation effort is placed on the LLM. But as the tools become more complex it will become harder to handwrite thorough examples that allow this generation. Today we introduce Sourceformer, which attempts to use a tool in the form of raw source code for self-training and benchmarking during evaluation. We propose a potentially viable method that allows tools to easily grow in complexity and size as the input token sequence to our LLMs inevitably grows. We focus on one tool in particular, a calculator, as a proof of concept for this idea; although, our results are sub par. Across three math benchmarks SVAMP, MAWPS, and ASDiv our model accuracy increases slightly, for some versions, compared to our base model before finetuning. Our code is available here https://github.com/erichmond33/sourceformer*

1 Introduction

Large language models (LLMs) like GPT-3 [Brown et al. 2020] have many emergent capabilities that make them very useful in a wide variety of situations. For the first time in history our computers are beginning to understand human language, or at least mimic understanding extremely well. A very interesting next step is using this understanding to interface and control tools in ways that increase human efficiency and creativity. And it is this large overarching idea that motivates us to pursue smaller steps forward within the world of tool use.

In this paper, we focus on one tool, a calculator. In previous work, Toolformer [Schick et al. 2023] was shown to take advantage of calculators, and other tools, by using the self-training process. This process takes handwritten examples of these API tools, and generates a training dataset based on this guidance. However, as we try to use more complex tools, it will become much more strenuous to create thorough handwritten examples for complex tools that can do many different things. ***Today we introduce Sourceformer, which learns to use a calculator by understanding the tool in the form of raw source code. This technique provides a potentially viable method that allows tools to easily grow in complexity and size.*** Based on the performance of Toolformer's use on other tools and the similarity of our model to theirs, we believe this approach could work well with many other tools beyond the calculator despite our results. Another important aspect of our work is by using source code, we are leaning into the future of transformer models based on two assumptions.

#1 The input sequence of transformers will get much larger

Today we can only squeeze modest amounts of code into the input sequence of modern transformers. It feels like the early days of digital memory when we had kilobytes worth of storage to work with. But in the future we can imagine having a much larger input sequence. This increase in length could potentially store entire apps or websites worth of source code providing a modern day AI assistant that can interface and control our computers.

#2 Future data will be collected in real time

Our hunger for data is always increasing. Eventually our need will cap out at what is being produced in real time. Sensors and computer logs will collect everything that is happening in the present. A key source of data will be the interactions between users and computers. This self-supervised app/website data will be a great source of training data for larger versions of what we are

proposing here because these interactions are with the raw source code that these apps and websites are built with.

2 Approach

We take the self-training [Schick et al. 2023; He et al. 2020] approach and generate our entire dataset. The idea is to augment a textual dataset by allowing a pre-trained transformer LLM to insert calculator function calls into places where the information is helpful later in the text. With this augmented dataset, we then finetune the same model with the augmented dataset that it generated. Thus, the model self-trains itself to use a tool, the calculator. An example of a single generated sample is shown in figure 1.

There are 2000 students and only 120 teachers, resulting in a [divide(2000, 120) -> 16.67] 16.67 student to teacher ratio.

Figure 1: An ideal generated data example. The inserted function call is highlighted.

In this example, we can see that the beginning of the function call is denoted with `" ["` and the end is denoted with a `"] "`. We call these `<start_tool_use>` and `<stop_tool_use>` tokens in this paper; although, to avoid adding two new tokens to the model's vocabulary, in our code we use `" ["` and `"] "` to represent these. In our calculator source code we only include add, subtract, multiply, and divide as available functions to call. Any number of function calls can happen in-between the `<start_tool_use>` and `<stop_tool_use>` tokens including nested function calls.

2.1 Defining the data and model

Our dataset C is the c4 - colossal, cleaned version of common crawl - dataset [Raffel et al. 2020] which includes a large amount of text data. From this point forward, a *Sentence*, with a capital 'S', indicates a single example of text data from our dataset rather than a single English sentence such as the one being read right now. Given C, through the data generation process, only an augmented subset of the dataset will be kept. We denote this augmented data-subset as C^*. We use GPT-Neo 1.3 billion parameters [Black et al. 2021] as our large language model, M, that will be used to generate and finetune on C^*. [1]

2.2 Preprocessing

Before evaluating a *Sentence* any further, we put it through some heuristics to help speed up quality data generation. Only one of the three heuristics must be met, otherwise we skip to the next *Sentence* in C.

(i) contain at least three numbers within a window of 100 tokens, where one of these numbers is the result of applying a mathematical operation to the other two.

No explicit operators (+,-,*,/) have to be included in the text, and we try every possible combination of all the numbers found within the 100 token window to satisfy this constraint.

(ii) contain one of the sequences "=", "equals", "equal to", "total of", "average of" followed by a number.

(iii) contain at least three numbers; for texts that only match the last criterion, we only keep a random subset of 1%.

To account for a maximum input token sequence of *1024* in M, we break each *Sentence* up into 128 token sequences before proceeding with the following steps. Let these 128 token *Sentence* chunks be denoted as *Sentence-128*.

2.3 Sentence position filtration

With our model M, at every position i within *Sentence-128*, we calculate the probability of each predicted token $z = z_1, \ldots, z_n$. This means each z is a vector with some non-negative prediction between 0 and 1 for the `<tool_use_start>` token, as well as every other token in our model's vocabulary. We filter z by taking the top k positions based on the predicted probability of the `<tool_use_start>` token where k=*20*. See figure 2 for a visual representation of this process.

---

1 We actually use GPT-J 6B for data generation as well. We had originally planned to train on GPT-J; however, we could not find the computational resources for this. This is discussed in more depth in section 3.1 Data generation.

Sentence	There	are	2000	students	...	a	16.67	student	to	teacher	ratio.
Probability	0.01	0.2	0.02	0.04	...	0.3	0.01	0.00	0.03	0.02	0.1
Index	0	1	2	3	...	10	11	12	13	14	15

Figure 2: This is a visual representation of sentence position filtration. We start with a single training *Sentence*. At each index we calculate the predicted probability for the <tool_use_start> token.* For this example only, let $k = 2$. We then grab the index of the top k probabilities. Here those is indexes are 1 and 10.

* Note these values are only for demonstration.

2.4 Generation

For each z after top k filtration, we are left with 20 positions where the next token prediction for `<tool_use_start>` is the most probable. Let $x = x_1, \ldots, x_n$ be a slice of Sentence-128 from the start, up to position z. With our model M we put some instructions, the source code, some examples, x, and `<tool_use_start>` as an input token prompt before doing generation. We include our entire input prompt in Appendix A.

We generate m continuations for each z where $m = 10$. The `<stop_tool_use>` token is treated as an end of sequence token, otherwise generation is stopped after a maximum of 28 new tokens have been generated.

2.5 Calling the functions

First we check for the `<stop_tool_use>` token. Then we check to ensure that either the strings add, subtract, multiply, or divide are found in between the start and stop tokens. If found, we execute the generated function call (or nested function calls) as Python code. We format the entire function call and response as `"[functionName(num1, num2) -> response]"` and a function call with no response as `"[functionName(num1, num2)]"`.

2.6 Loss threshold filtration

Let *f->r* represent the function call and response and *f* represent just the function call with no response. We start by creating three different versions of *Sentence-128*.

f->r + Sentence-128 (version 1): Here we prefix *Sentence-128* with *f->r*.

f + Sentence-128 (version 2): Here we prefix *Sentence-128* with *f*.

***Sentence-128* (version 3):** This is the normal *Sentence-128* with no changes.

Let each version be represented as *v1, v2, v3*. Note that because our model doesn't know how to use these function calls yet, we prefix them instead of inserting them in the position they were generated at.

Let i be the position of the function call that was generated in section 2.3 Sentence position filtration. Let $x = x_1, \ldots, x_n$ be a version of *Sentence-128*, in our case either *v1, v2, v3*. Also, let w be a sequence of weights where $w_t = \max(0, 1 - 0.2 \cdot t)$. Using weighted cross entropy loss

$$L_i(\mathbf{z}) = -\sum_{j=i}^{n} \left(\frac{w_{j-i}}{\sum_{k=i}^{n} w_{k-i}}\right) \cdot \log p_M\left(x_j \mid \mathbf{z}, x_{1:j-1}\right)$$

we calculate the loss for all three variations *v1, v2, v3* of our *Sentence-128*.

Next, given a filter threshold $T_f = 0.3$ we only keep examples where

$$\min\left(L_i(v2), L_i(v3)\right) - L_i(v1) \geq T_f$$

the loss of *v1* is at least T_f smaller than the minimum of *v2* or *v3*.[2] In other words, the function call and response must reduce the loss by T_f else it will be cut from our final training dataset.

3 Results

In order to determine how well our fine-tuned model uses the calculator, we evaluate it zero-shot on question and answer math benchmarks including SVAMP, MAWPS, and ASDiv [Patel, Bhattamishra, and Goyal 2021; Koncel-Kedziorski et al. 2016; Miao, C.-C. Liang, and Su 2020]. Figure 3 shows an example of an SVAMP sample.

Julia played tag with 18 kids on Monday. She played tag with 10 kids on Tuesday. How many more kids did she play with on Monday than on Tuesday?

Answer: 8

Figure 3: A single example from the SVAMP dataset. Both MAWPS and ASDiv questions use a similar question and answer format.

3.1 Data generation

We use 1 Nvidia A100 32gb, 1 Nvidia Tesla t4 24gb, and 1 Nvidia t4 16gb to generate data using the methods we described in section 2 Approach. On the A100, we generate data with GPT-J 6 billion parameters [B. Wang and Komatsuzaki 2021]; however, we only use GPT-Neo 1.3 billion parameters on the latter two GPUs. Originally we had planned to train and evaluate using GPT-J 6B; however, due to compute constraints, we had to scale back to GPT-Neo 1.3B.

Using a filter threshold T_f of *0.3*, our generated dataset *C** includes 3697 calculator function calls across 1577 training examples. 33% of our dataset comes from GPT-J while the rest comes from GPT-Neo.

3.2 Training

For finetuning we use floating point 16, a batch size of 8, a learning rate of 1 * 10^-5 with linear warmup the first 10% of training. Our hardware includes 1 Nvidia A100 32gb and 1 Nvidia Tesla t4 24gb. Our results are based on training for 30 epochs; however, they are almost the same when using any lower number of epochs. We train using Deepspeed ZeRO-3.

3.3 Models & testing input prompts

During testing we check the performance of two models and three different zero shot input prompt variations.

GPT-Neo: Nothing special here, just vanilla GPT-Neo. Our zero shot testing prompt for this model is a question from one of our math benchmarks with the text " The answer is" added to the end.

Sourceformer: This is GPT-Neo finetuned on our calculator dataset. If the `<stop_tool_use>` token is generated, we pause generation, execute the function call, then resume generation. Our zero shot setup is the same as GPT-Neo above.

Sourceformer (source-code on): This is GPT-Neo finetuned on our calculator dataset. Here any function calls will be executed and thus have a response. "source-code on" means the four source code functions add, subtract, multiply, and divide will be included at the front of the zero shot testing prompt used by GPT-Neo above.

Sourceformer (cheats on): This is GPT-Neo finetuned on our calculator dataset. However, at the end of our zero shot testing prompt we add the `<start_tool_use>` token to force the model into using the calculator tool. Other than this addition, the zero shot prompt is the same as GPT-Neo above.

Examples of the zero shot testing input prompts can be found in Appendix B.

3.4 Benchmarks

We evaluate our performance on the SVAMP, ASDiv, and MAWPS datasets. The correct answers

---

[2] The sequence of weights will always be $w = [1/3, .8/3, .6/3, .4/3, .2/3, 0, \ldots, 0, n]$ where the first element, 1/3, is at the function call position *i*.

for these are always a single number, so to determine whether the answer generated by our model is correct or not, we check the first number generated by the model. However, if the model's output contains a "=" we count that question as wrong. Originally we planned on flagging these and manually checking if an equation such as "The answer is 5+5=10" existed, thus checking the number after the equals sign. The number of questions where an equal sign was generated is on average around 50 per benchmark, and the number of correct answers within those 50 are miniscule.

Table 1 shows our results. And as we can see, Sourceformer performs almost identically to the base GPT-Neo thus it didn't learn how and when to use the tool as well as we would have liked. There are 1% fluctuations across a few different models, but these are negligible; our model performs roughly the same as base GPT-Neo.

Across all benchmarks our **Sourceformer** model calls a calculator function 6.1% of the time, **Sourceformer source-code on** calls the tool 4.8% of the time, and lastly when we strongly suggest the model to call a function with **Sourceformer cheats on**, it calls a function 98% of the time. Clearly, our model has no idea how or when to use this tool as the accuracy does not move significantly in any case.

We are not entirely sure why Toolformer was able to get such good results, and we were not. Perhaps our implementation has an error somewhere, we didn't have enough data, we didn't have high enough quality data, we should have trained on more than one tool in order for our model to gain a general understanding of how to use tools, or some combination of these potential problems.

Model	SVAMP	MAWPS	ASDiv
GPT-Neo	2.2	1.5	1.6
Sourceformer	1.5	1.7	1.2
Sourceformer source-code on	3.0	2.1	1.8
Sourceformer cheats on	2.5	2.1	1.4

Table 1: The accuracy percentage from testing different versions of our model on the SVAMP, MAWPS, and ASDiv benchmarks.

4 Related works

This paper is largely based on Toolformer [Schick et al. 2023]. We are very thankful for their team's great work.

4.1 Tools in the form of APIs:

The most related paper overall is of course Toolformer. Here their most important contribution is the loss filtering method which we use in our paper as well. They also test their methods on many tools including the calculator whereas we only focus on the one. The key distinction between Toolformer's calculator implementation and our's is we teach our model to use raw source code vs a calculator API. Their methods yield x results, which are better/worse than ours. Our aim was to, at minimum, match Toolformer's performance on these math benchmarks, thus showing that raw source code is a viable path going into the future.

Quite a few papers use APIs as tools in some form or another. Here an API is used as a search tool for code generation [Zhang et al. 2023]. These take the approach of having access to many different tools in the form of APIs with particular workflows and processes to pick and use the correct one [Y. Liang et al. 2023], [Li et al. 2023]. For biomedical information, web APIs can be used to help query databases to gain more precise specialized knowledge [Jin et al. 2023].

4.2 Tools in the form of generated code:

PAL [Gao et al. 2023] is a model that uses chain of thought to break down a question and then generates Python code as a tool in between each thought to solve the problem. We focus on having the model use the source code tools we provide it, while PAL generates everything on its own. Similarly, ART [Paranjape et al. 2023] use Codex [M. Chen et al. 2021] to generate tools on the fly as well.

PAL and ART achieve 79.4 & 76.2 on SVAMP, and PAL achieves 79.6 on ASDiv.

4.3 Avoiding labeling:

Finding ways to avoid labeling data is very exciting and nice compliment to the self-supervised nature of transformers. In our paper, we try to copy Toolformer's self-training process as closely as possible. Similar to this, [Y. Wang et al. 2023]

uses textual query generation to self-train a model for document retrieval. We can also see the usefulness of self-training being adopted in other areas. For pre-trained image processing[H. Chen et al. 2021], they don't generate data, but they do create corrupted image pairs which also do not require any kind of labeling. DALLE2 [Ramesh et al. 2022], which uses diffusion, works in a similar way. Diffusion models learn to generate data by reversing a gradual noising process.

4.4 Transformer LLMs:

Another fundamental proponent in this paper is the transformer LLM. The model used in this paper, GPT-Neo is a smaller open source version of GPT-3 [Brown et al. 2020]. And a key proponent of transformer LLMs is the attention mechanism [Vaswani et al. 2017].

5 Conclusion

In this paper we introduce Sourceformer, a novel approach to teaching a transformer LLM how and when to use a calculator. Our method attempts to teach GPT-Neo how to use this tool in the form of raw source code in order to show that this is a viable path forward for larger and more complex tools. We use self-training to generate our dataset, which is then used to finetune our pre-trained transformer LLM. Experiments on the SVAMP, MAWPS, and ASDiv datasets show our model performs similarly to our base model before finetuning. These results indicate that our methods either have a large flaw or simply don't work; however, given Toolformer's success, we assume the former.

References

Black, Sid et al. (Mar. 2021). GPT-Neo: Large Scale Autoregressive Language Modeling with Mesh-Tensorflow. Version 1.0. doi: 10.5281/zenodo.5297715. url: https://doi.org/10.5281/zenodo.5297715.

Brown, Tom B. et al. (2020). Language Models are Few-Shot Learners. arXiv: 2005.14165 [cs.CL].

Chen, Hanting et al. (2021). Pre-Trained Image Processing Transformer. arXiv: 2012.00364 [cs.CV].

Chen, Mark et al. (2021). Evaluating Large Language Models Trained on Code. arXiv: 2107.03374 [cs.LG].

Gao, Luyu et al. (2023). PAL: Program-aided Language Models. arXiv: 2211.10435 [cs.CL].

He, Junxian et al. (2020). "Revisiting Self-Training for Neural Sequence Generation". In: International Conference on Learning Representations. url: https://openreview.net/forum?id=SJgdnAVKDH.

Jin, Qiao et al. (2023). GeneGPT: Augmenting Large Language Models with Domain Tools for Improved Access to Biomedical Information. arXiv: 2304.09667 [cs.CL].

Koncel-Kedziorski, Rik et al. (June 2016). "MAWPS: A Math Word Problem Repository". In: Proceedings of the 2016 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies. San Diego, California: Association for Computational Linguistics, pp. 1152–1157. doi: 10.18653/v1/N16-1136. url: https://aclanthology.org/N16-1136.

Li, Minghao et al. (2023). API-Bank: A Benchmark for Tool-Augmented LLMs. arXiv: 2304.08244 [cs.CL].

Liang, Yaobo et al. (2023). TaskMatrix.AI: Completing Tasks by Connecting Foundation Models with Millions of APIs. arXiv: 2303.16434 [cs.AI].

Miao, Shen-yun, Chao-Chun Liang, and Keh-Yih Su (July 2020). "A Diverse Corpus for Evaluating and Developing English Math Word Problem Solvers". In: Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics. Online: Association for Computational Linguistics, pp. 975–984. doi: 10.18653/v1/2020.acl-main.92. url: https://aclanthology.org/2020.acl-main.92.

Paranjape, Bhargavi et al. (2023). ART: Automatic multi-step reasoning and tool-use for large language models. arXiv: 2303.09014 [cs.CL].

Patel, Arkil, Satwik Bhattamishra, and Navin Goyal (June 2021). "Are NLP Models really able to Solve Simple Math Word Problems?" In: Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies. Online: Association for Computational Linguistics, pp. 2080–2094.

doi: 10.18653/v1/2021.naacl-main.168. url: https://aclanthology.org/2021.naacl-main.168.

Raffel, Colin et al. (2020). Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer. arXiv: 1910.10683 [cs.LG].

Ramesh, Aditya et al. (2022). Hierarchical Text-Conditional Image Generation with CLIP Latents. arXiv: 2204.06125 [cs.CV].

Schick, Timo et al. (2023). "Toolformer: Language models can teach themselves to use tools". In: arXiv preprint arXiv:2302.04761.

Vaswani, Ashish et al. (2017). Attention Is All You Need. arXiv: 1706.03762 [cs.CL].

Wang, Ben and Aran Komatsuzaki (May 2021). GPT-J-6B: A 6 Billion Parameter Autoregressive Language Model. https://github.com/kingoflolz/mesh-transformer-jax.

Wang, Yujing et al. (2023). A Neural Corpus Indexer for Document Retrieval. arXiv: 2206.02743 [cs.IR].

Zhang, Kechi et al. (2023). ToolCoder: Teach Code Generation Models to use API search tools. arXiv: 2305.04032 [cs.SE].

Appendix A

Your task is to add calculator function calls to a piece of text. The calls should help you get information required to complete the text. You can call a function by writing "[functionName(num1, num2)]". Here are the available calculator functions:

```
def add(num1, num2):
    return num1 + num2

def subtract(num1, num2):
    return num1 - num2

def divide(num1, num2):
    if num2 == 0:
        raise ValueError("Cannot divide by zero.")
    return num1 / num2

def multiply(num1, num2):
    return num1 * num2
```

Here are some examples of calculator function calls:
Input: The number in the next term is 18 + 12 x 3 = 54.
Output: The number in the next term is 18 + 12 x 3 = [add(18, multiply(12, 3))] 54.
Input: The population is 658,893 people. This is 11.4% of the national average of 5,763,868 people.
Output: The population is 658,893 people. This is 11.4% of the national average of [divide(658,893, 11.4%)] 5,763,868 people.
Input: A total of 252 qualifying matches were played, and 723 goals were scored (an average of 2.87 per match). This is three times less than the 2169 goals last year.
Output: A total of 252 qualifying matches were played, and 723 goals were scored (an average of [divide(723, 252)] 2.87 per match). This is twenty goals more than the [subtract(723, 20)] 703 goals last year.
Input: I went to Paris in 1994 and stayed there until 2011, so in total, it was 17 years.
Output: I went to Paris in 1994 and stayed there until 2011, so in total, it was [subtract(2011, 1994)] 17 years.
Input: From this, we have 4 * 30 minutes = 120 minutes.
Output: From this, we have 4 * 30 minutes = [multiply(4, 30)] 120 minutes.

Input: x
Output: There are 2000 students and only 120 teachers, resulting in a [** Model Generated Text **

Figure 4: This is the input prompt we use to generate our data. We append the entire Sentence up to the top k position. Then our model generates an example - hopefully a proper function call of course. Notice we prod the model to do a function call by adding the function start token " [". We repeat this step 20 times before taking the example with the best loss defined in section 2.6 Loss threshold filtration.

Appendix B

GPT-Neo & Sourceformer:

Julia played tag with 18 kids on Monday. She played tag with 10 kids on Tuesday. How many more kids did she play with on Monday than on Tuesday? The answer is

Sourceformer (source-code on):

```
def add(num1, num2):
    return num1 + num2

def subtract(num1, num2):
    return num1 - num2

def divide(num1, num2):
    if num2 == 0:
        raise ValueError("Cannot divide by zero.")
    return num1 / num2

def multiply(num1, num2):
    return num1 * num2
```

Julia played tag with 18 kids on Monday. She played tag with 10 kids on Tuesday. How many more kids did she play with on Monday than on Tuesday? The answer is

Sourceformer (cheats on):

Julia played tag with 18 kids on Monday. She played tag with 10 kids on Tuesday. How many more kids did she play with on Monday than on Tuesday? The answer is [

Figure 5: These are the zero shot input prompts we use for each model at test time. Here the sentence "Julia played tag..." is an example from the SVAMP benchmark meant to be replaced by whatever question we are currently evaluating. The text highlighted in red are the additions that classify the versions as either **source-code on** or **cheats on**.