

ToolFormer: Guiding AI Models To Use External Tools

Meta's LLM teaches itself to call External APIs



tds

[Nikos Kafritsas](#)

.

Published in

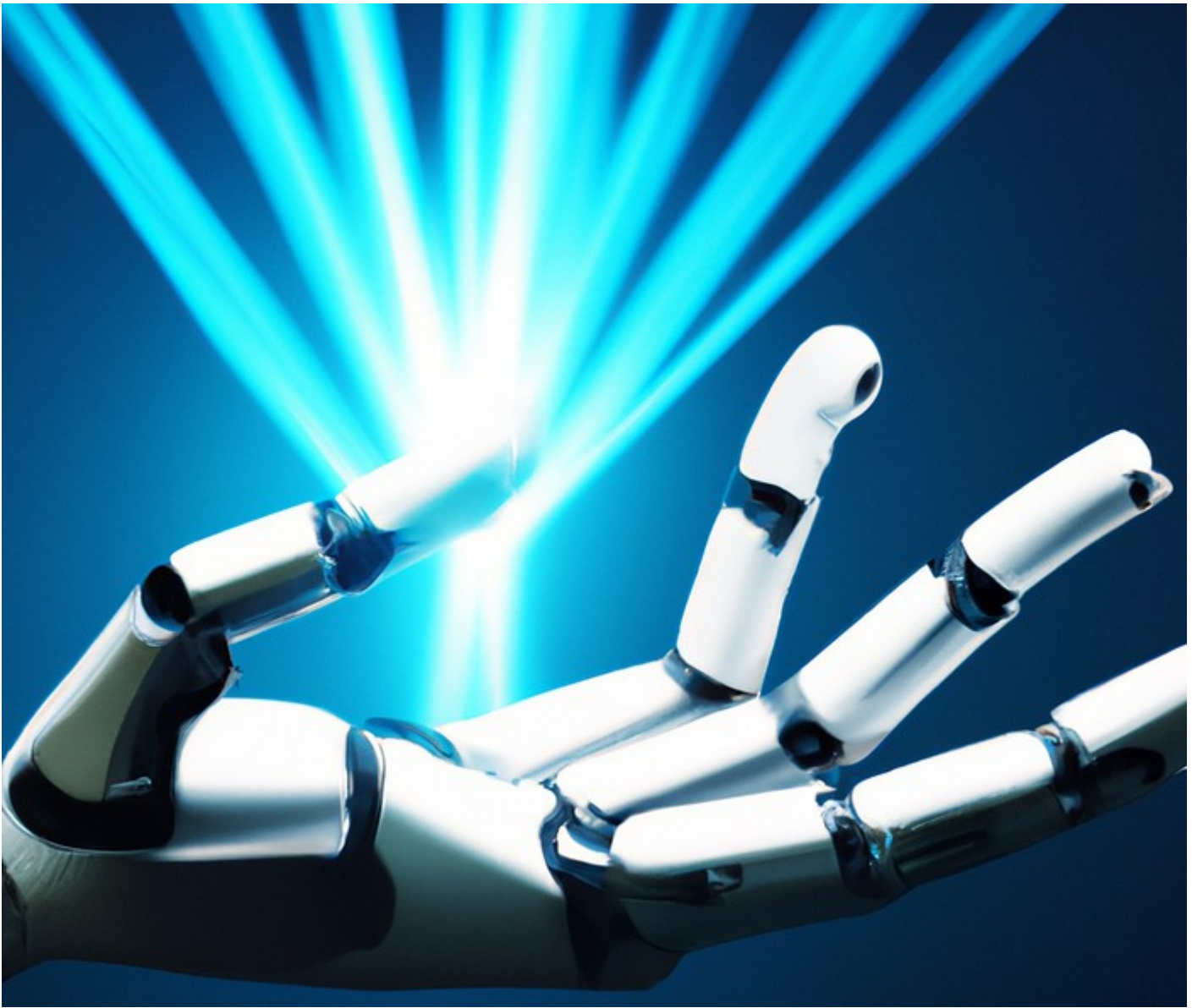
[Towards Data Science](#)

.

14 min read

.

Oct 23



Now that the dust has settled, the weaknesses of LLMs are known.

Even the powerful GPT-4 struggles with math operations.

Also, the training cut-off time is an inherent weakness of every LLM. They struggle to answer queries on new things.

A loose fix is to use external Plugins (e.g. ChatGPT plugins). Still, the user has to manually specify some actions, and these plugins are sometimes unreliable.

What if there was a model that knew its weaknesses — and was trained to **natively** call the optimal external tool when uncertain?

That's what Meta did, by creating **ToolFormer[1]**.

In this article, we discuss :

- What is ToolFormer and why is it a breakthrough?
- How the model works.
- How ToolFormer's methodology can be applied to any LLM.
- Why AI research heads towards ToolFormer's vision.

Let's dive in.

Weaknesses of Large Language Models

Before starting to describe ToolFormer, let's explore what issues the modern LLMs face:

- **Progression of Time:** Every LLM has a training cutoff date. Hence, they can't access up-to-date information and recent events.
- **Incorrect Facts:** LLMs are infamous for making up facts, places, events, products, and even research papers.
- **Arithmetic operations:** LLMs struggle with mathematical calculations.
- **Rare languages:** LLMs cannot handle low-resource languages, usually due to a lack of training data.

Obviously, these issues are irrelevant to language mechanics. An ideal solution would be to combine text generation with external tools.

Here comes ToolFormer.

What is ToolFormer?

ToolFormer is an LLM, trained to decide which APIs to call, when to call them, and what arguments to pass to call them.

ToolFormer is amazing because of:

- **Best of both worlds:** ToolFormer is an LLM, like GPT-3. But when uncertain, it learns to call external APIs — thus avoiding common mistakes.
- **Portability:** The methodology of training ToolFormer can be applied to any LLM.
- **Superior Performance:** ToolFormer is smaller, but outperforms much larger models like OPT and GPT-3.
- **Open Source:** While Meta has not released the original version yet, the community has created a [few great open-source implementations](#).

ToolFormer provides the following tools. These are shown in **Figure 1**:

The New England Journal of Medicine is a registered trademark of [QA("Who is the publisher of The New England Journal of Medicine?") → Massachusetts Medical Society] the MMS.

Out of 1400 participants, 400 (or [Calculator(400 / 1400) → 0.29] 29%) passed the test.

The name derives from "la tortuga", the Spanish word for [MT("tortuga") → turtle] turtle.

The Brown Act is California's law [WikiSearch("Brown Act") → The Ralph M. Brown Act is an act of the California State Legislature that guarantees the public's right to attend and participate in meetings of local legislative bodies.] that requires legislative bodies, like city councils, to hold their meetings open to the public.

Figure 1: ToolFormer autonomously calls external APIs to obtain accurate information and complete the output text (highlighted). From top to bottom, the APIs are: a question-answering system, a calculator, a machine translation system, and a Wikipedia search engine. ([Source](#))

According to **Figure 1**, ToolFormer provides:

- **QA:** A question-answering system
- **Calculator**
- **MT:** a machine translation system
- **WikiSearch:** a Wikipedia search engine API
- **Calendar:** a calendar API that returns the current date (not shown in **Figure 1**)

How ToolFormer generates text

In each case, the model decides which API to call and what arguments to use. The tool names like **QA** and **Calculator** are special tokens.

Out of 1400 participants, 400 (or `Calculator(400 / 1400)`
→ 0.29] 29%) passed the test.

If the model generates `Calculator(400/1400)`, then the model is ready to call the `Calculator API` with `(400/1400)` as argument.

The `->` token signifies that the model next expects the response for the API call.

When that happens, decoding(inference) is interrupted, and the model places the answer from the corresponding API. Decoding then continues, if necessary, until the answer is complete (...*passed the test.*).

How ToolFormer is Built

It's time to delve into technical stuff.

The key innovation of ToolFormer isn't the base pretrained model — it's the dataset used for training and particularly the **unique way** the authors augmented it.

Fundamentally, ToolFormer is a **GPT-J** pretrained model:

ToolFormer, a small pretrained **GPT-J** 6.7B model, beats the much larger GPT-3 and OPT on numerous tasks.

The authors used a subset of the CCNet training dataset (abbreviated as **C**). Then, they augmented that dataset with API calls — and called it **C***

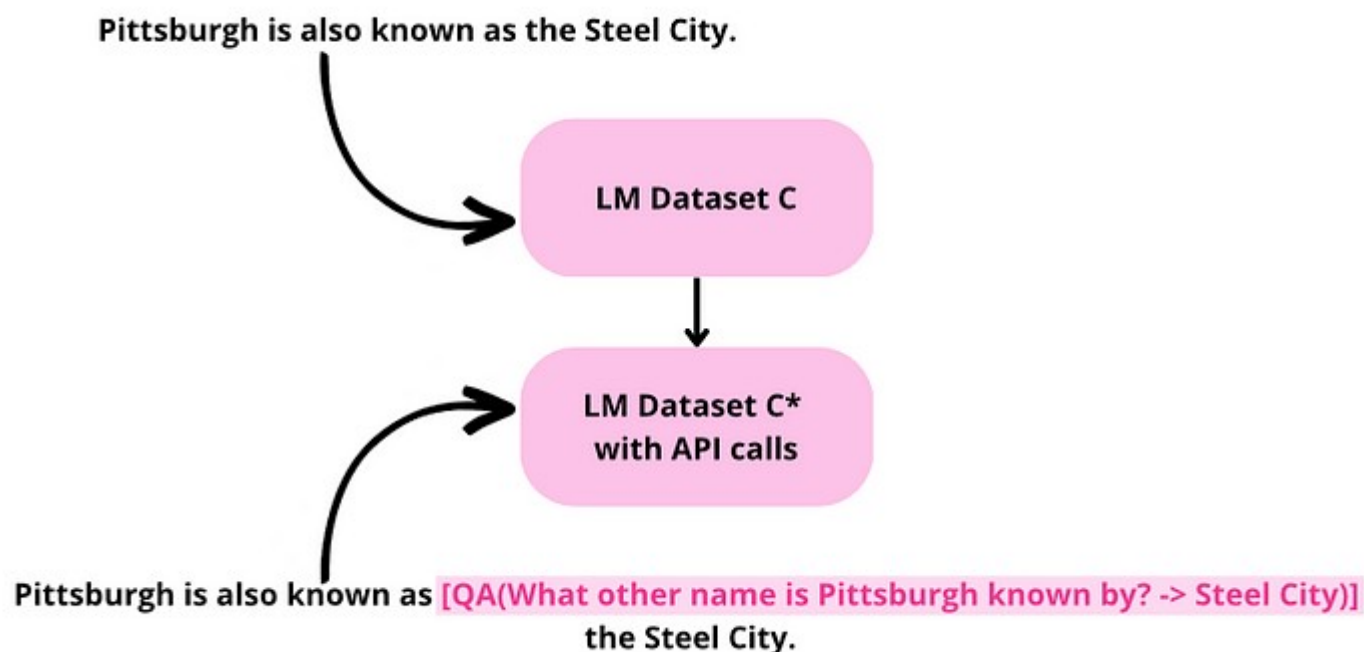


Figure 2: Augmentation of a training example for the QA tool (Image by author)

The process of augmenting C to C* is the real novelty of ToolFormer; This dataset can be used to teach any model how to effectively use API calls.

However, augmenting the training dataset is not an easy feat. We discuss this next.

Augmenting the Training Dataset

This is the most crucial part. The authors have 3 goals here:

1. **No human intervention.** We don't expect that a human will manually perform the process shown in **Figure 2**.
2. **Top-quality data:** The augmentation should be meaningful and helpful for the model. For example, the augmentation: **Pittsburgh is also known as the [QA: (What type of material characterizes Pittsburgh? -> Steel)] the Steel City** is wrong and not meaningful.
3. **No loss of generality:** With the new dataset, ToolFormer will still function as an LLM (able to optimally predict the next word).

Now, it's time to zoom in on **Figure 2** and reveal the intermediate steps between Dataset **C** and Dataset **C***.

A more detailed view is shown in **Figure 3**:

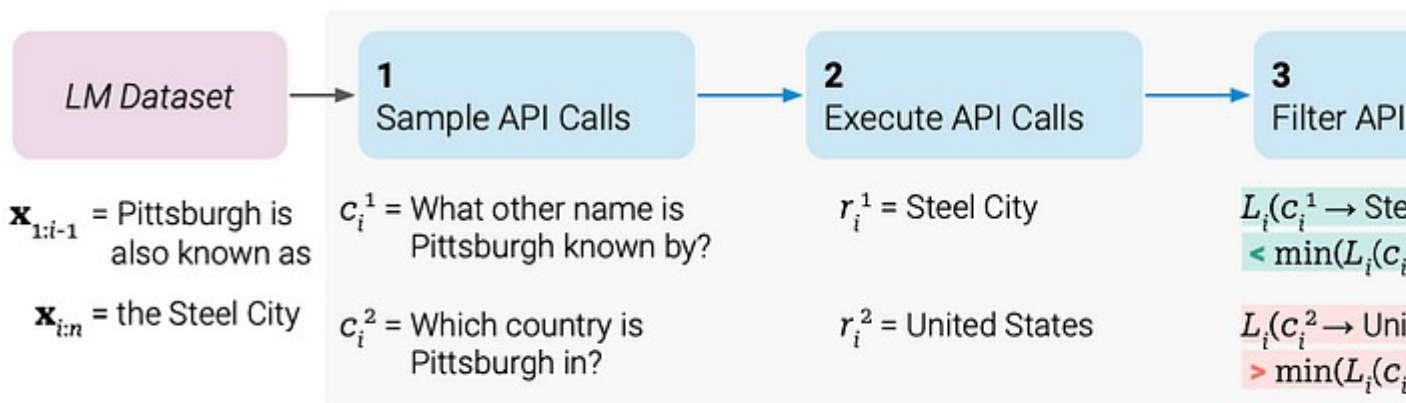


Figure 3: All 3 steps of augmenting a training example \mathbf{x} for the QA tool. ([Source](#))

There are 3 steps — let's decompose them. Given a sentence \mathbf{x} , we have:

- **Sample API Calls:** Sample a position \mathbf{i} in sentence \mathbf{x} that is likely to be used for an API call. Then, generate sample candidate API calls $[\mathbf{c1}, \mathbf{c2}.. \mathbf{ck}]$
- **Execute API Calls:** Execute those API calls, and take the responses $[\mathbf{r1}, \mathbf{r2}.. \mathbf{rk}]$
- **Filter API Calls:** Not all pairs $(c_i \rightarrow r_i)$ are useful or correct. We filter the API calls that don't reduce the loss function L over the next tokens.

Don't worry if you don't fully understand the steps. In the next section, we will delve deeper into each step.

Step 1: Sample API Calls

In this step, we generate possible API calls — from the dataset **C**.

The prerequisites are i) no manual intervention and ii) the API calls should be as meaningful as possible.

The best way to automate this task is to ask the GPT-J model to make the annotations itself!

Specifically, we will write a prompt $P(x)$ with instructions and a few examples — and encourage the model to annotate a sentence x with API calls. For the QA tool, the authors use the following prompt:

Your task is to add calls to a Question Answering API to a piece of text. The questions should help you get information required to complete the text. You can call the API by writing "[QA(question)]" where "question" is the question you want to ask. Here are some examples of API calls:

Input: Joe Biden was born in Scranton, Pennsylvania.

Output: Joe Biden was born in [QA("Where was Joe Biden born?")] Scranton, [QA("In which state is Scranton?")] Pennsylvania.

Input: Coca-Cola, or Coke, is a carbonated soft drink manufactured by the Coca-Cola Company.

Output: Coca-Cola, or [QA("What other name is Coca-Cola known by?")] Coke, is a carbonated soft drink manufactured by [QA("Who manufactures Coca-Cola?")] the Coca-Cola Company.

Input: x

Output:

Figure 4: Example of prompt $P(x)$ that generates API calls for the QA tool. ([Source](#))

Note: The process of including a few examples in the prompt to help the model better understand the given task is called *in-context learning*.

But remember, language models tend to hallucinate or produce errors — that's why we need the filtering process in **Step 3**.

Next, let's examine how **Step 1** works in practice:

- We will use the prompt $P(x)$ from **Figure 4** to annotate a sentence x with some candidate API Calls.
- Let's try the sentence $x =$ **Pittsburgh is also known as the Steel City.**

Prompt: Your task is to [...]

Input: **Pittsburgh is also known as the Steel City**

Output:

Pittsburgh is also known as [QA("In which state is Pittsburgh?")] the Steel City.

Pittsburgh is also known as [QA("What other name is Pittsburgh known by?")] the Steel City.

Pittsburgh is also known as [QA("Which is the second city in Pennsylvania?")] the Steel City.

Figure 5: Example of candidate API calls using as input the sentence x =**Pittsburgh is also known as the Steel City**. (Image by author)

We got 3 annotated sentences as output. Obviously, only the 2nd candidate API call is meaningful here.

The purpose of this step is to generate multiple annotated sentences without human effort. We will address how to filter out the incorrect ones later.

Note: The authors also impose a minimal filtering process here, to save costs. For more info, check the **Appendix** at the end of the article.

Step 2: Execute API Calls

This is straightforward — given the candidate calls from **Step 1**, we ask APIs for responses:

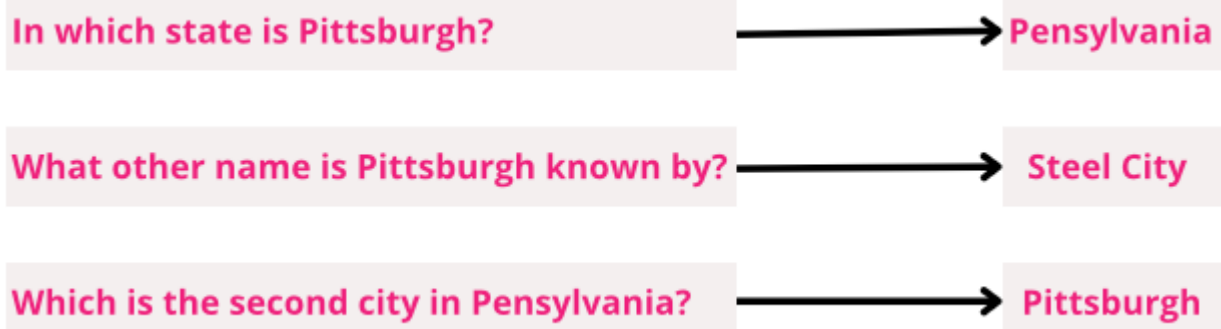


Figure 6: In Step 2, we get responses for each candidate call, regarding the QA tool (Image by author)

In reality, there are no actual APIs for all cases — except for the **Calculator** and the **Calendar** cases, which are simple scripts.

For the other cases, the authors use specialized LMs. For example, for the QA tool, they use Atlas, (Izacard et al., 2022), a retrieval-augmented LM finetuned on Natural Questions.

Feel free to check the paper for further details on each tool.

Step 3: Filter API Calls

While **Step 1** generated numerous API Calls, **Step 3** keeps only the meaningful ones.

Meaningful API calls: **Those which improve the model’s capability to call external APIS.**

This improvement is measured with a loss function and a formula — called **helpfulness ratio**.

The whole process is displayed in **Figure 7**:

The goal is to check if the following example is meaningful enough to be included in the augmented C* dataset:

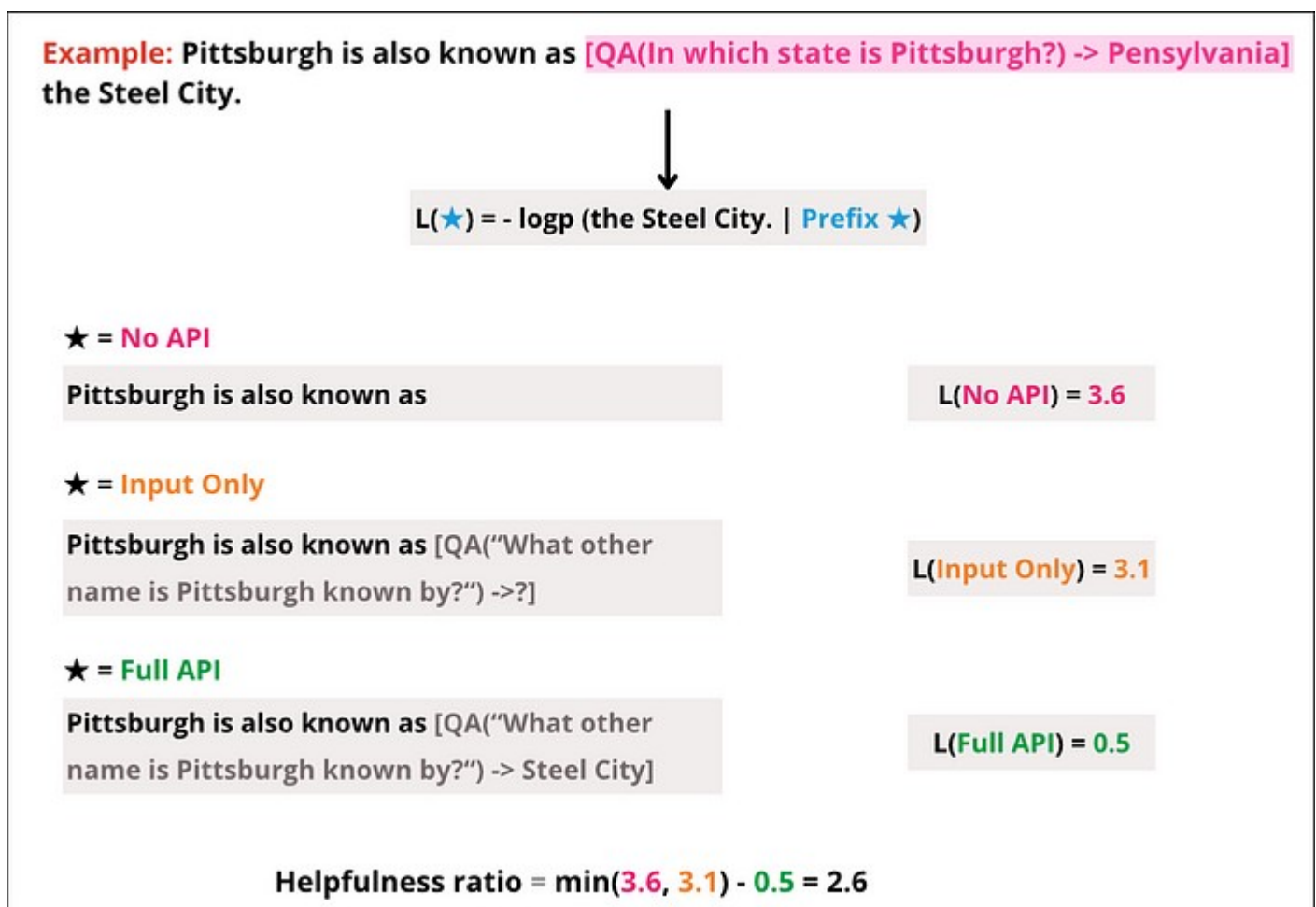


Figure 7: The helpfulness ratio determines whether an augmented candidate sentence is useful for the model (Image by author)

Let’s break down what happens here:

We calculate the *negative log-likelihood* of each case as the prefix to the phrase “the Steel City”:

- **No API:** We calculate $p(\text{the Steel City} \mid \text{Pittsburgh is also known as})$. Here, we don’t help the model much — that’s why the loss is high.
- **Input Only:** We calculate $p(\text{the Steel City} \mid \text{Pittsburgh is also known as [QA("What other name is Pittsburgh known by?") ->?]})$

- **Full API:** We calculate $p(\text{the Steel City} \mid \text{Pittsburgh is also known as [QA("what other name is Pittsburgh known by?") ->Steel City]})$

The full API case is the most helpful. Besides, the prefix contains the correct answer ‘Steel City’. That’s why we obtain the lowest loss here!

However, we have to quantify how helpful an annotation is. Here comes the *helpfulness ratio*. The higher the ratio, the more helpful the annotation is for training ToolFormer.

In **Figure 7**, we achieve a high helpfulness ratio, so our annotated example [**Pittsburgh is also known as [QA ... city]**] goes into the augmented dataset **C***.

The role of the helpfulness ratio

Now, consider this:

A more powerful model like GPT-4 probably knows that Pittsburgh is also known as the “Steel City”. In that case, the loss of the **No API** case would be low and almost similar to the **Full API** case. That leads to a helpfulness ratio close to 0.

But GPT-J, being a smaller model, doesn’t know the answer.

Hence, the GPT-J model benefits from being finetuned on the annotated example **Pittsburgh is . . city**, while **GPT-4** doesn't. Probably, GPT-4 would require more complex examples.

Thus, the process of training ToolFormer can be applied to any LM — thanks to the 3-step pipeline and the helpfulness ratio formula.

A rejection example

Remember, for our case, we sampled 3 API calls (**Figure 5**). Only the 2nd one is meaningful — the other 2 should be rejected.

Let’s see a rejection example. We will use as an example the following API call (the 1st in **Figure 5**):

Pittsburgh is also known as [QA(**In which state is Pittsburgh?**) -> **Pensylvania**] the Steel City.



Figure 8: Here, the annotated example is not helpful for our model (Image by author)

Here, we make an API call for a completely irrelevant question — “*which state does Pittsburgh belong*”. This does not help our model answer “*how else is Pittsburgh known*”.

Hence, we get a high loss, which means a negative helpfulness ratio. Thus, that annotated API call is not inserted in the C^* dataset.

The threshold τ_f for the helpfulness ratio

So far so good, but how high should the helpfulness ratio of an example be to be considered meaningful— and eligible to enter C^* ?

The authors found this threshold τ_f experimentally — by considering the number of training examples per category on C^* for different values of the helpfulness ratio. The results are shown in **Figure 9:**

| API | Number of Examples | | |
|---------------------|--------------------|----------------|----------------|
| | $\tau_f = 0.5$ | $\tau_f = 1.0$ | $\tau_f = 2.0$ |
| Question Answering | 51,987 | 18,526 | 5,135 |
| Wikipedia Search | 207,241 | 60,974 | 13,944 |
| Calculator | 3,680 | 994 | 138 |
| Calendar | 61,811 | 20,587 | 3,007 |
| Machine Translation | 3,156 | 1,034 | 229 |

Figure 9: Number of examples with API calls in C^* for different values of our filtering threshold τ_f .

Obviously, by increasing the threshold τ_f , fewer examples are going into C^* .

However, **Machine Translation** and **Calculator** examples were fewer than in the other categories. Hence, to avoid a serious imbalance, the authors set:

- For **QA**, **Wiki Search**, and **Calendar**, $\tau_f = 1$
- For **Machine Translation** and **Calculator**, $\tau_f = 0.5$

Final Step: Finetuning ToolFormer

The augmented dataset C^* is now ready.

We finetune GPT-J on C^* — and voila, we get **ToolFormer!**

Finetuning is trivial. The authors use *perplexity* for the finetuning objective.

Perplexity is a standard metric to evaluate how uncertain a language model is. For instance, a perplexity of 32 means the model is as sure for predicting the next word as throwing a 32-side die. Hence, lower is better.

Evaluation

Next, the authors evaluate ToolFormer.

In total, the authors use the following models:

- **GPT-J:** The original GPT-J pretrained model.
- **GPT-J on C:** Here, GPT-J fine-tuned on the C dataset (the one without API calls).
- **ToolFormer:** GPT-J model fine-tuned on C^* dataset.
- **ToolFormer (disabled):** ToolFormer, but using API calls is disabled. (This is done by setting the probability of generating the [token during inference to zero)

The goal is to evaluate the model for each separate task: QA, Calculator, Calendar, and so on. Let's start:

QA Evaluation (LAMA Benchmark)

The authors evaluate ToolFormer on 3 subsets of the [LAMA Benchmark](#): SQuAD, Google-RE, and T-REx.

For each of these subsets, the task is to complete a short statement with a missing fact — e.g. The theory of relativity is developed by ___ and the model should fill in the correct fact.

The results of this benchmark are shown in **Figure 10**.

Note: The performance scores below represent metrics that are evaluated differently for each dataset. To avoid getting into details, consider that **higher is better**. This is true for the other benchmarks throughout this paper.

| Model | SQuAD | Google-RE | T-REx |
|-----------------------|-------------|-------------|-------------|
| GPT-J | 17.8 | 4.9 | 31.9 |
| GPT-J + CC | 19.2 | 5.6 | 33.2 |
| Toolformer (disabled) | 22.1 | 6.3 | 34.9 |
| Toolformer | 33.8 | 11.5 | 53.5 |
| OPT (66B) | 21.6 | 2.9 | 30.1 |
| GPT-3 (175B) | 26.8 | 7.0 | 39.8 |

Figure 10: Performance of ToolFormer on the LAMA benchmark. ([Source](#))

The results are particularly interesting: ToolFormer **outperforms** the much larger OPT and GPT-3 on all benchmarks.

The power of ToolFormer comes from its ability to call external APIs in challenging situations.

Specifically, the model decided to call the QA API in **98.1%** of all cases. For only very few examples, it uses a different tool (**0.7%**) or no tool at all (**1.2%**).

Calculator Evaluation (Math Benchmark)

| Model | ASDiv | SVAMP | MAWPS |
|-----------------------|-------------|-------------|-------------|
| GPT-J | 7.5 | 5.2 | 9.9 |
| GPT-J + CC | 9.6 | 5.0 | 9.3 |
| Toolformer (disabled) | 14.8 | 6.3 | 15.0 |
| Toolformer | 40.4 | 29.4 | 44.0 |
| OPT (66B) | 6.0 | 4.9 | 7.9 |
| GPT-3 (175B) | 14.0 | 10.0 | 19.8 |

Figure 11: Performance of ToolFormer on the math benchmarks. ([Source](#))

- ToolFormer again outperforms OPT and GPT-3 by a large margin.
- The model decides to call the Calculator API in 97.9% of all cases.

Wiki Search Evaluation (Search Benchmark)

Here, ToolFormer is **not the best model**:

| Model | WebQS | NQ | TriviaQA |
|-----------------------|-------------|-------------|-------------|
| GPT-J | 18.5 | 12.8 | 43.9 |
| GPT-J + CC | 18.4 | 12.2 | 45.6 |
| Toolformer (disabled) | 18.9 | 12.6 | 46.7 |
| Toolformer | 26.3 | 17.7 | 48.8 |
| OPT (66B) | 18.6 | 11.4 | 45.7 |
| GPT-3 (175B) | <u>29.0</u> | <u>22.6</u> | <u>65.9</u> |

Figure 12: Performance of ToolFormer on Search benchmarks. ([Source](#))

ToolFormer outperforms OPT but loses to GPT-3. The authors provide the following reasons:

- ToolFormer Wiki Tool searches on **Wikipedia** only, instead of the whole Web (GPT-3 was trained on a huge portion of online content).
- ToolFormer would have outperformed GPT-3 if it was able to call the QA Tool as well. The authors disabled the QA Tool on purpose — because the datasets used to train the QA system have some potential overlap with the data in these benchmarks.
- An additional layer on top of the **Wiki Tool** is necessary — to reformulate the return results and provide clearer answers.

Translation Evaluation (MLQA Benchmark)

Here, the results are very interesting:

| Model | Es | De | Hi | Vi | Zh | Ar |
|-----------------------|-------------|-------------|------------|-------------|-------------|------------|
| GPT-J | 15.2 | 16.5 | 1.3 | 8.2 | 18.2 | 8.2 |
| GPT-J + CC | 15.7 | 14.9 | 0.5 | 8.3 | 13.7 | 4.6 |
| Toolformer (disabled) | 19.8 | 11.9 | 1.2 | 10.1 | 15.0 | 3.1 |
| Toolformer | 20.6 | 13.5 | 1.4 | 10.6 | 16.8 | 3.7 |
| OPT (66B) | 0.3 | 0.1 | 1.1 | 0.2 | 0.7 | 0.1 |
| GPT-3 (175B) | 3.4 | 1.1 | 0.1 | 1.7 | 17.7 | 0.1 |
| GPT-J (All En) | 24.3 | 27.0 | 23.9 | 23.3 | 23.1 | 23.6 |
| GPT-3 (All En) | 24.7 | 27.2 | 26.1 | 24.9 | 23.6 | 24.0 |

Figure 13: Performance of ToolFormer on Translation benchmarks. ([Source](#))

ToolFormer easily outperformed OPT and GPT-3 in all languages (except Zh, with GPT-3).

However, ToolFormer was surpassed by the original GPT-J. The authors explained this was because GPT-J was also pretrained on multilingual data — while C had very few multilingual examples.

Calendar Evaluation (Temporal Datasets)

Finally, we evaluate ToolFormer’s ability to extract dates and recent information.

The authors used 2 datasets:

- **TempLAMA:** Contains masked data that change with time (e.g., “Kylian Mbappé plays for _____”)
- **DATESET:** Contains random queries about dates (e.g. “What day of the week was it 30 days ago?”)

Figure 14 displays the results:

| Model | TEMPLAMA | DATESET |
|-----------------------|--------------------|--------------------|
| GPT-J | 13.7 | 3.9 |
| GPT-J + CC | 12.9 | 2.9 |
| Toolformer (disabled) | 12.7 | 5.9 |
| Toolformer | <u>16.3</u> | <u>27.3</u> |
| OPT (66B) | 14.5 | 1.3 |
| GPT-3 (175B) | 15.5 | 0.8 |

Figure 14: Performance of ToolFormer on Temporal benchmarks. ([Source](#))

Again, ToolFormer outperforms the much larger models. The difference is huge in **DATESET** — this is expected since finding dates is an inherent weakness of LLMs.

Scaling Laws

An integral part of training LLMs is whether the training model obeys the scaling laws.

Scaling laws are empirical rules that describe the relationship between a LM’s parameter size, tokens(dataset size), training time and performance.

Scaling laws were first introduced in [2], but were later re-examined in [3], where Deepmind researchers discovered that many LMs were significantly undertrained.

Here, the authors explored ToolFormer’s ability to scale, compared to the other models of the benchmark. The results are shown in **Figure 15:**

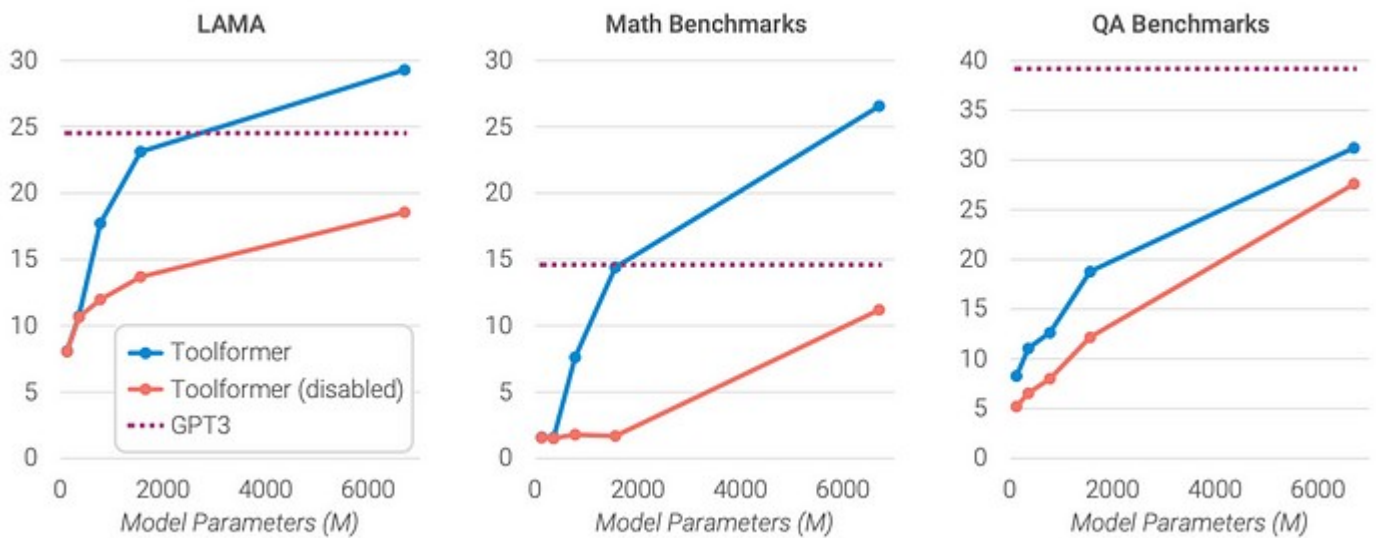


Figure 15: Performance of ToolFormer vs GPT3 on LAMA, math, and QA benchmarks in terms of model size. While API calls are not helpful to the smallest models, larger models learn how to make good use of them (Source)

Evidently, ToolFormer displays excellent signs of scalability — following scaling laws.

Smaller models (less than 775M parameters) achieve similar performance — they don't gain an advantage by calling external APIs. After 775M parameter size, ToolFormer starts to scale dramatically.

Decoding Strategy

An interesting part about ToolFormer is how the authors implemented the decoding strategy.

In truth, `[` is a special token — and signifies the start of an API call.

During word generation, an LM model calculates the probabilities of every token in the vocabulary, and generates **the one with the highest probability** (I explain it roughly). This is known as greedy decoding.

The authors found experimentally that ToolFormer performs better if `[` is generated when it is in the top $k=10$ most probable tokens, instead of getting generated when it's the most likely token ($k=1$).

The parameter $k=10$ is found experimentally. The results are shown in **Figure 16**:

| k | T-REx | | | | WebQS | | | |
|-----|-------------|------|------|------|-------------|------|------|-------|
| | All | AC | NC | % | All | AC | NC | % |
| 0 | 34.9 | – | 34.9 | 0.0 | 18.9 | – | 18.9 | 0.0 |
| 1 | 47.8 | 53.0 | 44.3 | 40.3 | 19.3 | 17.1 | 19.9 | 8.5 |
| 3 | 52.9 | 58.0 | 29.0 | 82.8 | 26.3 | 26.5 | 6.6 | 99.3 |
| 10 | 53.5 | 54.0 | 22.5 | 98.1 | 26.3 | 26.4 | – | 100.0 |

Figure 16: ToolFormer results on the T-REx subset of LAMA and WebQS for different values of k used during decoding. ([Source](#))

Clearly, the model performs best, on average when $k=10$. Figure 16 displays only 2 datasets. However, the pattern holds for the other benchmarks as well.

The Tools LLMs ecosystem

ToolFormer uses external APIs to solve maths and reasoning problems. But these problems can also be addressed by other approaches.

The most popular approach is called ‘**chain-of-thoughts**’ [4]:

In ‘chain of thoughts’, the LLM learns to break a prompt into intermediate steps — solving each step individually before giving the final answer.

Here, we don’t call external APIs. Instead, we teach the model to decompose a prompt into smaller parts — which helps the model with arithmetic tasks. An example is shown in **Figure 17**:

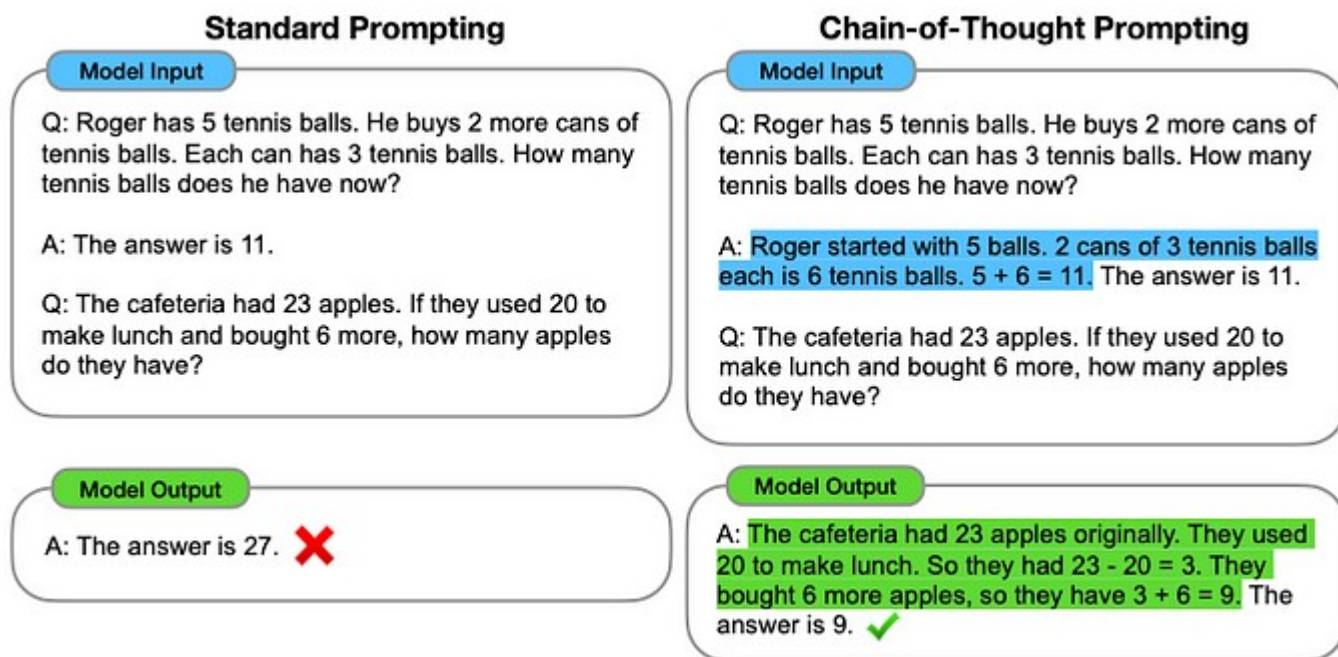


Figure 17: Using chain-of-thought prompting (right) the model can figure out the correct answer. Chain-of-thought reasoning processes are highlighted in green [[Wei et al.](#)]

The ‘**chain-of-thoughts**’ paradigm has been improved in the latest research.

Program-aided Language models (PAL) [5] achieve even better results by breaking down the prompts into both textual intermediate steps and Python code (**Figure 18**):

Chain-of-Thought (Wei et al., 2022)

Input


Q: Roger has 5 tennis balls. He buys 2 more cans of tennis balls. Each can has 3 tennis balls. How many tennis balls does he have now?

A: Roger started with 5 tennis balls. 2 cans of 3 tennis balls each is 6 tennis balls. $5 + 6 = 11$. The answer is 11.

Q: The bakers at the Beverly Hills Bakery baked 200 loaves of bread on Monday morning. They sold 93 loaves in the morning and 39 loaves in the afternoon. A grocery store returned 6 unsold loaves. How many loaves of bread did they have left?

Model Output

A: The bakers started with 200 loaves. They sold 93 in the morning and 39 in the afternoon. So they sold $93 + 39 = 132$ loaves. The grocery store returned 6 loaves. So they had $200 - 132 - 6 = 62$ loaves left. The answer is 62.



Program-aided Language models (this work)

Input

Q: Roger has 5 tennis balls. He buys 2 more cans of tennis balls. Each can has 3 tennis balls. How many tennis balls does he have now?

A: Roger started with 5 tennis balls.
`tennis_balls = 5`
2 cans of 3 tennis balls each is
`bought_balls = 2 * 3`
tennis balls. The answer is
`answer = tennis_balls + bought_balls`

Q: The bakers at the Beverly Hills Bakery baked 200 loaves of bread on Monday morning. They sold 93 loaves in the morning and 39 loaves in the afternoon. A grocery store returned 6 unsold loaves. How many loaves of bread did they have left?

Model Output

A: The bakers started with 200 loaves
`loaves_baked = 200`
They sold 93 in the morning and 39 in the afternoon
`loaves_sold_morning = 93`
`loaves_sold_afternoon = 39`
The grocery store returned 6 loaves.
`loaves_returned = 6`
The answer is
`answer = loaves_baked - loaves_sold_morning - loaves_sold_afternoon + loaves_returned`

```
>>> print(answer)
74
```




Figure 18: Chain-of-thought (left) gives a wrong answer, while PAL (right) is correct. PAL combines Chain-of-thought reasoning (highlighted in blue) with programming annotations (highlighted in pink) [Luyu Gao et al.]

Lastly, we can use LangChain, an application framework for LLMs. Langchain uses agents to integrate with various search APIs, capable of searching the web. **Figure 19** shows the SerpAPI tool:

```
params = {
    "engine": "bing",
    "gl": "us",
    "hl": "en",
}
search = SerpAPIWrapper(params=params)

search.run("Obama's first name?")

'Barack Hussein Obama II is an American politician who served as the 44th president of the United States'
```

Figure 19: We can instruct a language model to use an API for searching the web, and integrating the search results into the prompt so we get the correct answer. (Source)

What is the difference between ToolFormer and Langchain agents?

- Langchain agents have to first use the appropriate API (which a human specifies) and then combine the results with the prompt to get a correct answer.
- In contrast, ToolFormer was **explicitly trained** to call and integrate API tools (no manual intervention).

Closing Remarks

This article explored ToolFormer, a model capable of calling external Tools.

Essentially, ToolFormer is a process that can teach any LLM to call external APIs.

With the adoption of LLMS, the necessity to call external resources will become apparent. Even ChatGPT now allows the user to enrich his prompt with search results from the web.

Thank you for reading!

I write an in-depth analysis of an impactful AI paper once a month.

Stay connected!

- Follow me on [LinkedIn!](#)
- Subscribe to my newsletter, [AI Horizon Forecast!](#)

[AI Horizon Forecast | Nikos Kafritsas | Substack](#)

[Explaining complex AI models as clear as daylight. Focusing on time series and latest AI research. Click to read AI...](#)

aihorizonforecast.substack.com

Appendix

The authors also impose a minimal filtering process on the 1st step of the data augmentation process, to save costs. For example, the sentences that are not annotated at all with special tokens `[QA. .` etc are rejected from the next steps.

Also, the authors calculate the most probable positions in the sentence that are likely to initiate an API call. The symbols `[,]` are also special tokens and signify the start and the end of an API call.

So, the authors calculate the position i where the token `[` has the highest probability of appearing. Hence, only the sentences where the start of the API call(the token `[`) is generated on the most probable position i , are passed to the next step.

References

- [1] Timo Schick et al. [Toolformer: Language Models Can Teach Themselves to Use Tools](#)
- [2] Jared Kaplan et al. [Scaling Laws for Neural Language Models](#)

[3] Jordan Hoffmann et al. [Training Compute-Optimal Large Language Models](#)

[4] Jason Wei et al. [Chain-of-Thought Prompting Elicits Reasoning in Large Language Models](#) (January 2023)

[5] Luyu Gao et al. [PAL: Program-aided Language Models](#) (January 2023)

[Artificial Intelligence](#)

[Deep Learning](#)

[Data Science](#)

[Large Language Models](#)

[Machine Learning](#)



tds

[Written by Nikos Kafritsas](#)

[3.3K Followers](#)

·Writer for

[Towards Data Science](#)

Data Scientist @ Persado || Top Writer in Artificial Intelligence and Time Series