Toolformer: Language Models Can Teach Themselves to Use Tools

Timo Schick **Jane Dwivedi-Yu** **Roberto Dessì**[†] **Roberta Raileanu**
Maria Lomeli **Luke Zettlemoyer** **Nicola Cancedda** **Thomas Scialom**

Meta AI Research [†]Universitat Pompeu Fabra

Abstract

Language models (LMs) exhibit remarkable abilities to solve new tasks from just a few examples or textual instructions, especially at scale. They also, paradoxically, struggle with basic functionality, such as arithmetic or factual lookup, where much simpler and smaller models excel. In this paper, we show that LMs can teach themselves to *use external tools* via simple APIs and achieve the best of both worlds. We introduce *Toolformer*, a model trained to decide which APIs to call, when to call them, what arguments to pass, and how to best incorporate the results into future token prediction. This is done in a self-supervised way, requiring nothing more than a handful of demonstrations for each API. We incorporate a range of tools, including a calculator, a Q&A system, a search engine, a translation system, and a calendar. Toolformer achieves substantially improved zero-shot performance across a variety of downstream tasks, often competitive with much larger models, without sacrificing its core language modeling abilities.

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: Exemplary predictions of Toolformer. The model autonomously decides to call different APIs (from top to bottom: a question answering system, a calculator, a machine translation system, and a Wikipedia search engine) to obtain information that is useful for completing a piece of text.

1 Introduction

Large language models achieve impressive zero- and few-shot results on a variety of natural language processing tasks (Brown et al., 2020; Chowdhery et al., 2022, i.a.) and show several emergent capabilities (Wei et al., 2022). However, all of these models have several inherent limitations that can at best be partially addressed by further scaling. These limitations include an inability to access up-to-date information on recent events (Komeili et al., 2022) and the related tendency to hallucinate facts (Maynez et al., 2020; Ji et al., 2022), difficulties in understanding low-resource languages (Lin et al., 2021), a lack of mathematical skills to perform precise calculations (Patel et al., 2021) and an unawareness of the progression of time (Dhingra et al., 2022).

A simple way to overcome these limitations of today's language models is to give them the ability to *use external tools* such as search engines, calculators, or calendars. However, existing approaches either rely on large amounts of human annotations (Komeili et al., 2022; Thoppilan et al., 2022) or limit tool use to task-specific settings only (e.g., Gao et al., 2022; Parisi et al., 2022), hindering a more widespread adoption of tool use in LMs. Therefore, we propose *Toolformer*, a model that learns to use tools in a novel way, which fulfills the following desiderata:

- The use of tools should be learned in a self-supervised way without requiring large amounts of *human annotations*. This is impor-

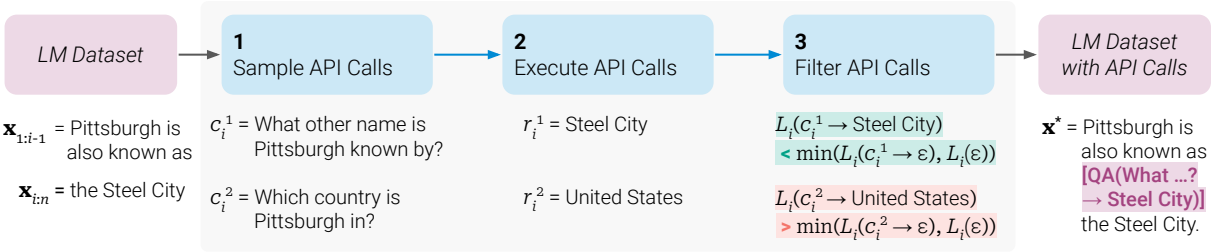

Figure 2: Key steps in our approach, illustrated for a *question answering* tool: Given an input text $\mathbf{x}$, we first sample a position i and corresponding API call candidates $c_i^1, c_i^2, \ldots, c_i^k$. We then execute these API calls and filter out all calls which do not reduce the loss L_i over the next tokens. All remaining API calls are interleaved with the original text, resulting in a new text $\mathbf{x}^*$.

tant not only because of the costs associated with such annotations, but also because what humans find useful may be different from what a model finds useful.

- The LM should not lose any of its *generality* and should be able to decide for itself *when* and *how* to use which tool. In contrast to existing approaches, this enables a much more comprehensive use of tools that is not tied to specific tasks.

Our approach for achieving these goals is based on the recent idea of using large LMs with *in-context learning* (Brown et al., 2020) to generate entire datasets from scratch (Schick and Schütze, 2021b; Honovich et al., 2022; Wang et al., 2022): Given just a handful of human-written examples of how an API can be used, we let a LM annotate a huge language modeling dataset with potential API calls. We then use a self-supervised loss to determine which of these API calls actually help the model in predicting future tokens. Finally, we finetune the LM itself on the API calls that it considers useful. As illustrated in Figure 1, through this simple approach, LMs can learn to control a variety of tools, and to choose for themselves which tool to use when and how.

As our approach is agnostic of the dataset being used, we can apply it to the exact same dataset that was used to pretrain a model in the first place. This ensures that the model does not lose any of its generality and language modeling abilities. We conduct experiments on a variety of different downstream tasks, demonstrating that after learning to use tools, Toolformer, which is based on a pretrained GPT-J model (Wang and Komatsuzaki, 2021) with 6.7B parameters, achieves much stronger zero-shot results, clearly outperforming a much larger GPT-3 model (Brown et al., 2020) and several other baselines on various tasks.

2 Approach

Our aim is to equip a language model M with the ability to use different tools by means of API calls. We require that inputs and outputs for each API can be represented as text sequences. This allows seamless insertion of API calls into any given text, using special tokens to mark the start and end of each such call.

We represent each API call as a tuple $c = (a_c, i_c)$ where a_c is the name of the API and i_c is the corresponding input. Given an API call c with a corresponding result r, we denote the linearized sequences of the API call not including and including its result, respectively, as:

$$\text{e}(c) = \texttt{<API>}\, a_c\,(i_c)\, \texttt{</API>}$$
$$\text{e}(c, r) = \texttt{<API>}\, a_c\,(i_c)\, \rightarrow r\, \texttt{</API>}$$

where "<API>", "</API>" and "→" are special tokens.[1] Some examples of linearized API calls inserted into text sequences are shown in Figure 1.

Given a dataset $\mathcal{C} = \{\mathbf{x}^1, \ldots, \mathbf{x}^{|\mathcal{C}|}\}$ of plain texts, we first convert this dataset into a dataset $\mathcal{C}^*$ augmented with API calls. This is done in three steps, illustrated in Figure 2: First, we exploit the in-context learning ability of M to sample a large number of potential API calls. We then execute these API calls and finally check whether the obtained responses are helpful for predicting future tokens; this is used as a filtering criterion. After filtering, we merge API calls for different tools, resulting in the augmented dataset $\mathcal{C}^*$, and finetune

[1] In practice, we use the token sequences " [", "]" and "->" to represent "<API>", "</API>" and "→", respectively. This enables our approach to work without modifying the existing LM's vocabulary. For reasons of readability, we still refer to them as "<API>", "</API>" and "→" throughout this section.

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 3: An exemplary prompt $P(\mathbf{x})$ used to generate API calls for the question answering tool.

M itself on this dataset. Each of these steps is described in more detail below.

Sampling API Calls For each API, we write a prompt $P(\mathbf{x})$ that encourages the LM to annotate an example $\mathbf{x} = x_1, \ldots, x_n$ with API calls. An example of such a prompt for a question answering tool is shown in Figure 3; all prompts used are shown in Appendix A.2. Let $p_M(z_{n+1} \mid z_1, \ldots, z_n)$ be the probability that M assigns to token z_{n+1} as a continuation for the sequence $z_1, \ldots, z_n$. We first sample up to k candidate *positions* for doing API calls by computing, for each $i \in \{1, \ldots, n\}$, the probability

$$p_i = p_M(\texttt{<API>} \mid P(\mathbf{x}), x_{1:i-1})$$

that M assigns to starting an API call at position i. Given a sampling threshold τ_s, we keep all positions $I = \{i \mid p_i > \tau_s\}$; if there are more than k such positions, we only keep the top k.

For each position $i \in I$, we then obtain up to m API calls $c_i^1, \ldots, c_i^m$ by sampling from M given the sequence $[P(\mathbf{x}), x_1, \ldots, x_{i-1}, \texttt{<API>}]$ as a prefix and `</API>` as an end-of-sequence token.[2]

[2]We discard all examples where M does not generate the `</API>` token.

Executing API Calls As a next step, we execute all API calls generated by M to obtain the corresponding results. How this is done depends entirely on the API itself – for example, it can involve calling another neural network, executing a Python script or using a retrieval system to perform search over a large corpus. The response for each API call c_i needs to be a single text sequence r_i.

Filtering API Calls Let i be the position of the API call c_i in the sequence $\mathbf{x} = x_1, \ldots, x_n$, and let r_i be the response from the API. Further, given a sequence $(w_i \mid i \in \mathbb{N})$ of *weights*, let

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

be the weighted cross entropy loss for M over the tokens $x_i, \ldots, x_n$ if the model is prefixed with $\mathbf{z}$. We compare two different instantiations of this loss:

$$L_i^+ = L_i(\mathrm{e}(c_i, r_i))$$
$$L_i^- = \min\left(L_i(\varepsilon), L_i(\mathrm{e}(c_i, \varepsilon))\right)$$

where ε denotes an empty sequence. The former is the weighted loss over all tokens $x_i, \ldots, x_n$ if the API call and its result are given to M as a prefix;[3] the latter is the minimum of the losses obtained from (i) doing no API call at all and (ii) doing an API call, but not providing the response. Intuitively, an API call is helpful to M if providing it with both the input *and* the output of this call makes it easier for the model to predict future tokens, compared to not receiving the API call at all, or receiving only its input. Given a filtering threshold τ_f, we thus only keep API calls for which

$$L_i^- - L_i^+ \geq \tau_f$$

holds, i.e., adding the API call and its result *reduces* the loss by at least τ_f, compared to not doing any API call or obtaining no result from it.

Model Finetuning After sampling and filtering calls for all APIs, we finally merge the remaining API calls and interleave them with the original inputs. That is, for an input text $\mathbf{x} = x_1, \ldots, x_n$ with a corresponding API call and result (c_i, r_i) at position i, we construct the new sequence $\mathbf{x}^* =$

[3]We provide $\mathrm{e}(c_i, r_i)$ as a prefix instead of inserting it at position i because M is not yet finetuned on any examples containing API calls, so inserting it in the middle of $\mathbf{x}$ would interrupt the flow and not align with patterns in the pretraining corpus, thus hurting perplexity.