Fine-Tuning Codex Toward Optimal Solutions for Sorting and Searching Problems

Sourya Kakarla, Derek Pham, Gabriele Tombesi
Department of Computer Science
Columbia University
New York, NY 10027

1 Problem, Formulation, and Goal

OpenAI Codex [7] is a language model obtained by training OpenAI's base GPT-3 [4]–which is pre-trained on web and text resources–on a vast set of public code repositories retrieved from GitHub. As a result of this additional training process, Codex has become reasonably proficient generating code in several programming languages by simply parsing the input natural language instructions. Typical of the third generation of generative transformers, Codex inherits few-shot, one-shot and zero-shot learning capabilities, intuiting a wide variety of downstream tasks just from the input settings, without need of additional instructions and with few to no examples.

The research problem that we formulate in this work is based on the observation that, while Codex is capable of achieving state-of-the-art results for basic Python program synthesis tasks, performance significantly drops when the complexity of the prompt's problem and consequently the corresponding programming solution increases. More specifically, Codex can produce outputs with time complexities which are several orders of magnitude higher than the corresponding optimal solutions.

Given this fact, we are interested in exploring solutions to improve the quality of results (QoR) of Codex-generated Python code. Given the complexity of the models leveraged by OpenAI and the limited computational resources at our disposal, we decide to break down this task into manageable pieces and set one of those as our research goal: we will focus on improving the computational efficiency of generated code for a specific subset of tasks, namely sorting and searching algorithms.

We choose to focus on improving the computational efficiency of generated code specifically for sorting and searching algorithms, because we are targeting tasks that are of medium-level difficulty, for which Codex is able to generate functionally correct solutions, but not optimal ones. For this reason, it would not make sense to focus on trivial tasks where Codex is already proficient, or on complex tasks where Codex is not able to handle yet, namely those where Codex cannot generate functionally correct solutions at this point in time. Nevertheless, even though some of our preliminary experiments have shown that this task set is promising for fine-tuning Codex, we want to stress that we are open to including harder prompts beyond search and sorting in the case too many problems based on those algorithms turn out to be already optimally solved by Codex. For example, we might include harder coding questions involving dynamic programming.

2 Method, Dataset, and Evaluation Criteria

2.1 Method and Dataset

We intend to leverage the fine-tuning API [6] offered by OpenAI. The basic idea of our approach is the following: given a set of prompts in our domain of interest, we will need to collect several functionally correct Python implementations with varying time complexity for each of those tasks. Given this initial raw version of the dataset–still not compliant to what Codex expects for fine-tuning–we intend to build a fine-tuning dataset composed of *[prompt:completions]* example pairs such that, for each of the original prompts in the target tasks set, we generate as many examples as the number of output implementations that we have for that task. It is crucial though, in order for the examples to be distinct and for the fine-tuning to improve the model, that the original prompt is concatenated with an additional token that embeds the optimality, in terms of complexity, of the corresponding completion (i.e. Python solution). A summary of this second stage of the dataset preparation is illustrated in Figure 1.

```
{"prompt_1": "solution_1_C1", "solution_1_C2", ... , "solution_1_Ck"}
{"prompt_2": "solution_2_C1", "solution_2_C2", ... , "solution_2_Ck"}

...

{"prompt_100": "solution_100_C1", "solution_100_C2", ... , "solution_100_Ck"}
```

```
{"prompt_1_1"="prompt_1"+"C_1", "completion": "solution_1_C1"}
{"prompt_1_2"="prompt_1"+"C_2", "completion": "solution_1_C2"}
..
{"prompt_1_k"="prompt_1"+"C_k", "completion": "solution_1_Ck"}

{"prompt_2_1"="prompt_2"+"C_1", "completion": "solution_2_C1"}
{"prompt_2_2"="prompt_2"+"C_2", "completion": "solution_2_C2"}
..
{"prompt_1_k"="prompt_2"+"C_k", "completion": "solution_2_Ck"}

...

{"prompt_100_1"="prompt_100"+"C_1", "completion": "solution_100_C1"}
{"prompt_100_2"="prompt_100"+"C_2", "completion": "solution_100_C2"}
..
{"prompt_100_k"="prompt_100"+"C_k", "completion": "solution_100_Ck"}
```

Figure 1: From raw dataset to Codex-compliant dataset

At this point, having the compliant data-set saved in the *fine_tune.json* file and setting code-davinci-001 [8] as the pretrained model that we intend to customize, we will proceed by passing the dataset to the fine-tuning API call. These steps, together with the openAI API call at evaluation phase, are summarized in figure 2.

```
openai tools fine_tunes.prepare_data -f fine_tune.json

openai api fine_tunes.create -t fine_tune.json -m code-davinci-001
```

```python
import openai

for test_prompt in test_set:
    openai.Completion.create(
    model=FINE_TUNED_MODEL_NAME,
    prompt=test_prompt)
```

Figure 2: Fine-Tuning call and Evaluation

The novelty of this approach is that we intend to perform a complexity-biased fine-tuning based on an additional input tokens indicating complexity, which are meant to teach the model what it means to generate an optimal solution for this specific subset of tasks that we target.

Finally, we note that a substantial amount of time for this research project will be dedicated toward collecting a suitable dataset with solutions of different efficiency levels for each task. We realize that this process may not be straightforward, and so under the circumstance in which we face difficulty in reaching our minimum goal of 100 examples, we are considering to integrate our search for existing solutions using the following method: We can use Codex–or another accessible code-generating model–to generate different implementations of a task with different complexity by leveraging parameters such as Temperature, which would allow us to generate more variance in the output. We could then feed the outputs into a benchmark tool to first discard the incorrect solutions, and then rank the remaining solutions based on time complexity. This way, we could create new prompt-solution pairs across different complexity levels to use for fine-tuning.

2.2 Evaluation Criteria

We will use two evaluation metrics for our evaluation criteria to see if there is any performance improvement with respect to the pre-fine-tuned model. The first evaluation metric is the percentage of prompts for which the fine-tuned model outputs correct programs. It is of interest to compare the percentage with the original engine of Codex and observe if there is any significant deviation due to our fine tuning.

Secondly, the metric with which we are mainly concerned about is the one that evaluates how the generated code performs when compared to the known gold solution. We plan to have a gold solution for every prompt in the test-set that has the best known complexity and high performing implementation. These solutions are human generated and evaluated. For all the valid codes generated by the engine for a particular prompt, we would evaluate how optimal they are by using distance measures with the gold solution's practical benchmarks and theoretical complexity.

We will use a benchmark module to measure the average time taken by the generated code for different sufficiently large inputs. Multiple runs are done to mitigate noise from the OS environment. This time measure is compared to the gold solutions' time measure, which is obtained in the same fashion, and then a suitable distance measure is calculated. The baseline for this metric can be computed by measuring average time taken through the same method for the codes generated by the original engine that is not fine-tuned. By evaluating the difference between the computed benchmark of the code generated from the fine tuned model and that of the baseline, we aim to survey the success of our fine-tuned model.

3 Previous Work

Henraks et al. (2021) have created APPS - Automated Programming Progress Standard, a benchmark for code generation [5]. This benchmark measures the ability of models to take an arbitrary natural language specification and generate satisfactory Python code. This benchmark includes 10,000 problems, which range from having simple one-line solutions to being substantial algorithmic challenges.

We take inspiration from Chen et al. (2021) who introduce Codex and evaluate its capabilities in generating Python code [2]. Moreover, Codex is fine-tuned from GPT-3, which is an autoregressive language model that achieves strong performance across a variety of natural-language-generating tasks, including translation and question-answering [1]. A significant portion of the existing language models for code-related tasks developed from the GPT architecture. Among them, IntelliCode Compose leverages the autoregressive approach of GPT, applying it to the field of source code understanding, namely multilingual code completion tasks [9].

In parallel to this line of work based on the most advanced auto-regressive models implemented so far, there are several examples of high-performance code-generation applications based on auto-encoding models. Among them, it is worth mentioning CodeBERT: an NL-PL pretrained model customized for natural language code search and code documentation generation [3].

Finally, the Code Transformer is a multilingual code summarization model that leverages the encoder-decoder architecture of sequence-to-sequence transformers, learning from both the source code and parsed abstract syntax trees [10].

References

[1] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared 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 M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners. *CoRR*, abs/2005.14165, 2020.

[2] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harrison Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Joshua Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. Evaluating large language models trained on code. *CoRR*, abs/2107.03374, 2021.

[3] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. Codebert: A pre-trained model for programming and natural languages. *CoRR*, abs/2002.08155, 2020.

[4] Luciano Floridi and Massimo Chiriatti. Gpt-3: Its nature, scope, limits, and consequences. *Minds and Machines*, 30(4):681–694, 2020.

[5] Dan Hendrycks, Steven Basart, Saurav Kadavath, Mantas Mazeika, Akul Arora, Ethan Guo, Collin Burns, Samir Puranik, Horace He, Dawn Song, et al. Measuring coding challenge competence with apps. *arXiv preprint arXiv:2105.09938*, 2021.

[6] OpenAI. Fine-tuning. `https://beta.openai.com/docs/guides/fine-tuning`, 2022. [Online; accessed 10-March-2022].

[7] OpenAI. Openai codex. `https://openai.com/blog/openai-codex/`, 2022. [Online; accessed 10-March-2022].

[8] OpenAI. Openai codex engines. `https://beta.openai.com/docs/engines/codex-series-private-beta`, 2022. [Online; accessed 10-March-2022].

[9] Alexey Svyatkovskiy, Shao Kun Deng, Shengyu Fu, and Neel Sundaresan. Intellicode compose: Code generation using transformer. *CoRR*, abs/2002.08155, 2020.

[10] Daniel Zügner, Tobias Kirschstein, Michele Catasta, Jure Leskovec, and Stephan Günnemann. Language-agnostic representation learning of source code from structure and context. *CoRR*, abs/2103.11318, 2021.