DOI: 10.15514/ISPRAS-2022-34(4)-6

Big Transformers for Code Generation

G.A. Arutyunov, ORCID: 00000-0003-4537-4332 <gaarutyunov@edu.hse.ru>
S.M. Avdoshin, ORCID: 0000-0001-8473-8077 <savdoshin@hse.ru>
HSE University,
20, Myasnitskaya st., Moscow, 101000 Russia

Abstract. IT industry has been thriving over the past decades. Numerous new programming languages have emerged, new architectural patterns and software development techniques. Tools involved in the process ought to evolve as well. One of the key principles of new generation of instruments for software development would be the ability of the tools to learn using neural networks. First of all, it is necessary for the tools to learn how to write code. In this work we study the ability of Transformers to generate competition level code. The main goal is to discover whether open-source Big Transformers are "naturally" good coders**.**

Keywords: neural networks; code generation; Transformers; GPT

For citation: Arutyunov G.A., Avdoshin S.M. Big Transformers for Code Generation. Trudy ISP RAN/Proc. ISP RAS, vol. 34, issue 4, 2022. pp. 79-88. DOI: 10.15514/ISPRAS-2022-34(4)-6

Acknowledgments: This research was supported in part through computational resources of HPC facilities at HSE University [1].

Большие трансформеры для генерации кода

Г.А. Арутюнов, ORCID: 0000-0003-4537-4332 <gaarutyunov@edu.hse.ru>
С.М. Авдошин, ORCID: 0000-0001-8473-8077 <savdoshin@hse.ru>
Национальный исследовательский университет «Высшая школа экономики» (НИУ ВШЭ),
101000, Россия, г. Москва, ул. Мясницкая, д. 20

Аннотация. Индустрия разработки программного обеспечения развивается стремительными темпами. Непрерывно появляются новые языки программирования, архитектурные паттерны и подходы к разработке. Развиваться должны и инструменты, используемые программистами. Среди необходимых условий появления нового семейства инструментария следует выделить способность обучаться за счет использования моделей машинного и глубинного обучения. В данной статье будут рассмотрены достижения последних лет в области применения авторегрессионых моделей для генерации программного кода из естественного языка. Основной целью исследования является оценка того, можно ли назвать Большие трансформеры с открытым исходным кодом программистами «от природы».

Ключевые слова: нейронные сети; генерация кода; Трансформеры; GPT

Для цитирования: Арутюнов Г.А., Авдошин С.М. Большие трансформеры для генерации кода. Труды ИСП РАН, том 34, вып. 4, 2022 г., стр. 79-88. 10.15514/ISPRAS-2022-34(4)-6

Благодарности. Исследование выполнено с использованием суперкомпьютерного комплекса НИУ ВШЭ [1].

1. Introduction

In the recent years, there has been solid advancement in code generation using neural networks. GitHub and OpenAI, for example, launched their tool called Copilot that can generate code from

prompts written in natural language [2]. The code assistant is based on OpenAI Codex [3], a proprietary GPT-3 [4] variant fine-tuned on code from GitHub.

The success of OpenAI's GPT-3 model has encouraged open-source communities to develop large pre-trained language models. For example, EleutherAI has developed several good performing autoregressive language models: GPT-Neo [5], GPT-J [6] and finally GPT-NeoX [7]. The last of these three has 20 billion parameters and performs remarkably well on several benchmarks as shown in the original paper [7].

The goal of this work is to test the ability of the two biggest EleutherAI models, GPT-J and GPT-NeoX, to generate Python competition-level code. We will analyse its performance and compare its results with preceding works using APPS benchmark [8].

2. Problem Statement

Although the task of text-to-code generation may appear similar to text-to-text generation, it has some major differences. First, the generated code must be syntactically correct in order to be executed. Moreover, it should solve the exact task that was expected from it. Therefore, testing code generation and using same quality metrics as with text generation is not useful [8].

2.1 The APPS Dataset

Due to these issues we will use a benchmark specifically designed to validate code generation – the APPS dataset [8]. It is composed by 10000 programming tasks from platforms such as Codewars AtCoder, Kattis and Codeforces. Tasks are separated into 3 difficulty levels: introductory, interview, competition. Furthermore, each task is accompanied by 20 test-cases on average that are used for validation.

2.2 Performance Metrics

As for model quality metrics, the same as in original work is used – average test cases. In addition, we review the runtime and syntax errors rate of generated code. Strict accuracy is only used in this work to compare previous research since they used it in their articles (see Table 1). However, it is not used for our models since Transformers studied in this work are not yet good coders for such a strict metric.

Average test cases metric shows the average number of test cases that generated code has passed. It is defined as follows:

$$\frac{1}{P}\sum_{p=1}^{P}\frac{1}{C_p}\sum_{c=1}^{C_p}1\{\text{eval}(\langle \text{code}_p\rangle, x_{p,c}) = y_{p,c}\},$$

P – number of tasks,
C_p– number of test cases per task,
$\langle \text{code}_p\rangle$ – code generated by the model
$\{(x_{pc}, y_{p,c})\}$ – set of inputs-outputs for the task.

Strict accuracy demands that a task passes all test cases. It is defined as follows:

$$\frac{1}{P}\sum_{p=1}^{P}\prod_{c=1}^{C_p}1\{\text{eval}(\langle \text{code}_p\rangle, x_{p,c}) = y_{p,c}\}$$

3. Previous Works and Motivation

After Hendrycks et al. showed in their research [8] that Transformers are able to generate code that can solve competition-level code several companies have tried to use models of similar architecture for the same task.

The first one was Codex [3] which is a model by OpenAI based on GPT-3. As we can see from the Table 1 the model almost doubled the results of previous GPT-Neo model at introductory problems.

Table 1. Comparison of GPT-Neo, GPT-J, GPT-NeoX, Codex and AlphaCode Performance ON APPS dataset. Strict accuracy is calculated from 5 solutions. For Codex and AlphaCode we also present results filtered from 1000 generated solutions

Model name	**Strict accuracy @ 5, %**		
	Introductory	***Interview***	***Competition***
GPT-Neo 2.7B	5.50	0.80	0.00
GPT-J 6B	0.00	0.00	0.00
GPT-NeoX 20B	0.15	0.20	0.00
Codex 12B	9.65	0.51	0.09
Codex 12B filtered from 1000	24.52	3.23	3.08
AlphaCode 1B filtered from 1000	14.36	5.63	4.58

The next is AlphaCode by Google [9]. Authors trained 5 models: 300M, 1B, 3B, 9B, 41B. However only the one with 1B parameters was tested on APPS dataset. As shown in Table 1 AlphaCode didn't beat Codex in introduction problems, however it did in the interview and competition tasks.

Sadly, both models are proprietary and it is impossible to research them and perform more experiments. Open-source models with comparable sizes are GPT-Neo 2.7B [5], GPT-J 6B [6] and GPT-NeoX 20B [7] – all developed by EleutherAI.

The first model has already been tested in the Hendrycks et al. work and showed promising results (see Table 1). The motivation behind this work is to determine whether by just fine-tuning Big Transformers like GPT-J and GPT-NeoX with billions of parameters, as it was done in the mentioned work, directly results in high performance in a code generation task such as APPS.

4. Proposed Solution

The primary model in this research is the GPT-NeoX. We used GPT-NeoX model since it showed results comparable with GPT-3 and other Big Transformers in many NLP tasks [7]. Moreover, it is one of the biggest available open-source models with 20 billion parameters. Other reasons for choosing GPT-NeoX model is its configurability suitable for extensive research, the tokenizer and the architecture of the model that allows parallelizing the training process.

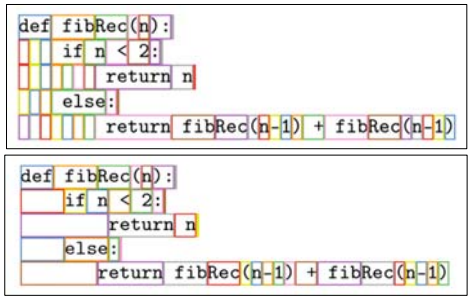

Fig. 1. Tokenization result for GPT-2 (above) and GPT-NeoX (below)

The tokenizer used in the model can process new lines spaces and tabs grouping them in one token which is very useful for code processing. For example, the code displayed on fig. 1 processed be the GPT-NeoX tokenizer has 39 tokens against 50 produced by the GPT-2 tokenizer.

In addition, the model supports model parallelism [10] and pipeline parallelism [11] with Deepspeed and PyTorch distributed. Therefore, even billions of parameters may fit in the GPUs' memory. Table 2 depicts the GPU requirements for models training. Smaller models with low sequence length can be trained with less requirements, however we doubled the number of GPUs to increase the speed of training process.

Table. 2. Comparison of GPU requirements for models training. "M" in model name stands for millions of tokens, "B" – for billions, while the number in the brackets "(2k)" shows the sequence length in thousands.

Model	**GPUs characteristics**		
	Number of GPUs	***GPU model***	***GPU memory***
165M (2k)	2	V100	32GB
165M (4k)	2	V100	32GB
165M (8k)	2	A100	80GB
6B (2)	8	A100	80GB
20B (2k)	8	A100	80GB

All the GPT-NeoX variants were trained using Adam optimizer with batch size of 64 for 100 epochs. Then the same batch size and optimizer were used for fine-tuning process during 10 epochs. The GPT-J model was fine-tuned with batch size of 256 for 10 epochs – same as GPT-Neo in the original APPS paper [8].

The development of the text-to-code generation model based on GPT-NeoX was separated into 4 different stages that are discussed below. The results and conclusions of these stages are detailed in the next section.

4.1 Fine-tuning on APPS

We started by fine-tuning both GPT-J 6B and GPT-NeoX 20B models on APPS dataset. In the original paper [8] GPT-Neo showed good results after such fine-tuning, however, neither the GPT-J nor GPT-NeoX did. As we can see in Table 1 both models yielded 0% strict accuracy in all the levels of difficulty.

4.2 Fine-tuning on English-to-Python datasets

Due to the failure of APPS-only fine-tuning we gathered 5 additional datasets. Two datasets that contain question-answer pairs scraped from Stackoverflow:

1) StaQC [12, 13] that contains around 270 thousand examples. We use the 148 thousand that are written in python.
2) CoNaLa [14] also scraped from Stackoverflow containing around 600 thousand examples.
3) Moreover, we used 2 additional datasets comprised of programming competition tasks:
4) Description2Code [15] comprised from 8 thousand tasks from CodeChef, Codeforces and Hackerearth. Codeforces data was excluded to avoid data leakage since it also appears in APPS dataset.
5) CodeNet [16, 17] that includes tasks from AIZU Online Judge and AtCoder platforms. AtCoder data was also excluded to avoid overlap with APPS dataset.
6) Finally, we add a dataset with parsed docstrings: CodeDocstringCorpus [18, 19] consisting of almost 150 thousand docstring-to-code pairs with functions and methods written in Python.

All the code in the above-mentioned datasets was processed by interpreting and transforming it into Python3 code using lib2to3. It was done to diminish syntax errors in generated code. All the code was then tokenized using a vocabulary of 52 thousand tokens.

4.3 Pre-training on Python code from GitHub

As demonstrated in other researches models pre-trained with code samples and then fine-tuned on text-to-code datasets yield promising results [9, 20]. Therefore, 150 thousand python scripts from around 5 thousand public GitHub repositories were collected. They were used to pre-train GPT-NeoX model on code.

4.4 Other models and context windows

Researchers have demonstrated that not only the size of the transformer matters but rather a good combination of size and context window [20].

In this work, we also pre-trained and fine-tuned smaller models with 165 million parameters. Furthermore, we increased the context windows from 2048 to 8192 tokens. The models were pre-trained with GitHub Python dataset and fine-tuned with English-to-Python dataset.

5. Experiment Results

In this section we will discuss the conclusions we made based on the results of all 4 stages.

5.1 Fine-tuning dataset matters

When creating a fine-tuning dataset for text-to-code generation, there are several things that need to be considered.

First, the dataset should not be too narrow for a concrete task. Otherwise, the model will be limited to the constructs it learned from the dataset. Programming is a very wide field, therefore the more examples you get, more programming skills the model will acquire.

Second, it is important to filter syntactically incorrect or inappropriate code, for example code written on another version of the programming language. Otherwise, the model will learn from bad examples which will result in crushed tests.

When conducting the experiments, the construction of the fine-tuning dataset allowed us to increase results from slightly above 0% average test cases to around 3% for the 20 billion model.

5.2 Text vs Code pre-training

Although some previous works demonstrate that code is similar to natural language [21], text pre-trained models are not expected to yield the same results as models pre-trained on code.

In our research we do not emphasize on comparing text vs code pre-trained models. However, as we can see in Table 3 GPT-NeoX models with less parameters that were pre-trained on code yield similar results as the 20 billion parameter model trained on the Pile [22].

5.3 Model size and context window

The performance of models was analysed in two manners: based on the average test cases passed and on the ratio of runtime and syntax errors.

Fig. 2 demonstrates the results of testing the code generated with the GPT-NeoX models with different sizes and context lengths. It shows very interesting results. As we can see, the smallest 165 million parameter model with sequence length of 8 thousand tokens acts almost as well as the 20 billion parameter model.

As for the errors rate, we noticed that the best models had a syntax errors rate around 5-15% and a runtime errors rate between 60 and 70%. For worst models the situation was the opposite: around 60% of the code was syntactically wrong.

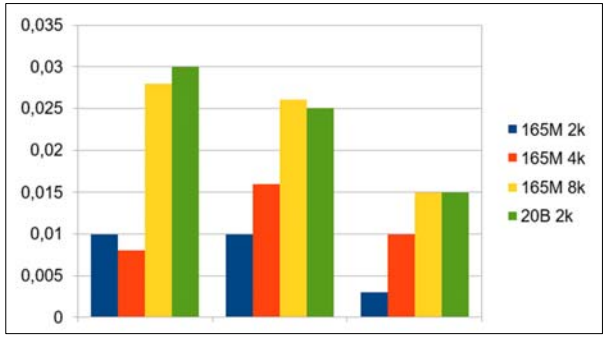

Fig. 2. Comparison of average passed test cases for different models

5.4 Best models

Table 3 shows the comparison of different models trained on code. In the case of GPT-NeoX the context window with best results for each model size are shown. As we can see, the number of parameters doesn't directly result in the best performance. Nor does the size of the pre-training dataset. The GPT-NeoX 20B model trained on the Pile performed similar to the 165M model that was trained on a comparably smaller dataset. However, the length of the context window does affect the model performance positively.

Table. 2. Comparison of EleutherAI models Performance on APPS dataset

Model name	**Avg. test cases passed, %**		
	Introductory	***Interview***	***Competition***
GPT-NeoX 165M 8k	2.80	2.60	1.50
GPT-Neo 2.7B 2k	14.68	9.85	6.54
GPT-J 6B 2k	0.09	0.01	0.00
GPT-NeoX 20B 2k	3.00	2.50	1.60

5.5 Quality of generated code

Apart from the metrics it is necessary to compare the generated code. In this section we will present some examples of code generated by both GPT-J and GPT-NeoX models. The task is formulated below. And the generated code is displayed on Figures 3 and 4.

Task: You are given a positive integer n, and an array a of positive integers. The task is to calculate the number of such pairs (i, j) that $i < j$ and $a_i \,\&\, a_j \geq a_i \oplus a_j$, where & denotes the bitwise AND operation, and $\oplus$ denotes the bitwise XOR operation.

```
t=int(input())
for i in range(t):
   n=int(input())

a=list(map(int,input().spl
it()))
   for j in range(n):
      a[j]=a[j]%2
   print(a.count(1))
```

Fig. 3. Code generated by GPT-NeoX model

On fig. 3 you can see the code generated by GPT-NeoX with 165 million parameters and context length of 8 thousand tokens. This code passes all the test cases and seems easily understandable and concise.

```
n = int(input())
a = list(map(int,
input().split()))
b = list(set(a))
b.sort()
for i in range(n-1):

print(b[b.index(a[i])+1])
```

Fig. 4. Code generated by GPT-J model

On fig. 4 you can see the code generated by GPT-J. Although this code doesn't contain syntax errors and seems pleasant overall it results in runtime errors.

As we can see, the models learnt to generate good code for simple tasks, nevertheless they could solve too few problems.

6. Future Work

This research demonstrated several key results on text-to-code generation.

First of all, the quality of a dataset matters a lot. To increase the metrics of a Transformer model it is necessary to construct a good dataset first. Future work could elaborate further on other languages – both programming and natural. For example, using datasets such as MCoNaLa [23] which contains examples on Russian, Spanish and Japanese.

Secondly, the research discovered that even smaller models pre-trained on code show similar results to bigger transformers pre-trained on text. However, analysing code as sequence may lead to information loss and increasing the context window on bigger models requires lots of computation resources. It would be a good approach to use models that support other representations of code that captures most of the information. For example, graph representation using abstract syntax tree could be used with graph transformers [24–27].

6. Conclusion

As for conclusion, in this work we demonstrated that Big Transformers are not naturally good coders. There are several points that must be taken into account to use them for code generation task.

First, it is important to collect extensive and high-quality dataset with minima incorrect code. Datasets must be verified by compiling code or running static analysis tool to exclude code that contains errors from the dataset.

Secondly, a model with appropriate context length and number of parameters must be chosen to generate functionally correct code. As shown in our results, context length is positively correlated with model performance.

References

[1] Kostenetskiy P.S., Chulkevich R.A., Kozyrev V.I. HPC Resources of the Higher School of Economics. Journal of Physics: Conference Series, vol. 1740, no. 1, 2021, article no. 012050, 11 p.

[2] Introducing GitHub Copilot: your AI pair programmer, The GitHub Blog, 2021. Available at: https://github.blog/2021-06-29-introducing-github-copilot-ai-pair-programmer/, accessed: 27.06.2022.

[3] Chen M., Tworek J. et al., Evaluating large language models trained on code. arXiv:2107.03374, 2021, 35 p.

[4] Brown T., Mann B. et al., Language models are few-shot learners. Advances in Neural Information Processing Systems, vol. 33, 2020, pp. 1877-1901.

[5] Black S., Gao L. et al. GPT-Neo: Large scale autoregressive language modeling with meshtensorflow, 2021. Available at: https://zenodo.org/record/5551208, accessed: 17.03.2022.

[6] Wang B., Komatsuzaki A. GPT-J-6B: A 6 Billion Parameter Autoregressive Language Model, 2021. Available at: https://github.com/kingoflolz/mesh-transformer-jax, accessed: 27.06.2022.

[7] Black S., Biderman S. et al. GPT-NeoX-20B: An Open-Source Autoregressive Language Model. arXiv:2204.06745, 2022, 42 p.

[8] Hendrycks D., Basart S. et al., Measuring coding challenge competence with APPS. arXiv:2105.09938, 2021, 22 p.

[9] Li Y., Choi D. et al., Competition-Level Code Generation with AlphaCode. arXiv:2203.07814, 2022, 74 p.

[10] Shoeybi M., Patwary M. et al. Megatron-LM: Training Multi-Billion Parameter Language Models Using Model Parallelism. arXiv:1909.08053, 2020, 15 p.

[11] Harlap A., Narayanan D. et al. PipeDream: Fast and Efficient Pipeline Parallel DNN Training. arXiv:1806.03377, 2018, 14 p.

[12] Yao Z., Weld D.S. et al. StaQC: A Systematically Mined Question-Code Dataset from Stack Overflow, In Proc. of the 2018 Conference on World Wide Web, 2018, pp. 1693-1703.

[13] Yao Z., Weld D.S. et al. StackOverflow-Question-Code-Dataset, 2022. Available at: https://github.com/LittleYUYU/StackOverflow-Question-Code-Dataset, accessed: 17.06.2022.

[14] Yin P., Deng B. et al., Learning to mine aligned code and natural language pairs from stack overflow, In Proc. of the 15th International Conference on Mining Software Repositories, 2018, pp. 476-486.

[15] Caballero E., Sutskever I. Description2Code Dataset, 2016. Available at: https://github.com/ethancaballero/description2code, accessed: 17.06.2022.

[16] Puri R., Kung D.S. et al. CodeNet: A Large-Scale AI for Code Dataset for Learning a Diversity of Coding Tasks. c, 2021, 22 p.

[17] IBM, Project CodeNet, 2022. Available at: https://github.com/IBM/Project_CodeNet, accessed: 17.06.2022.

[18] code-docstring-corpus, 2022. Available at: https://github.com/EdinburghNLP/code-docstring-corpus, accessed: 17.06.2022.

[19] Barone A.V.M., Sennrich R. A parallel corpus of python functions and documentation strings for automated code documentation and code generation. arXiv: 1707.02275, 2017, 5 p.

[20] Nijkamp E.б Pang B. et al. A Conversational Paradigm for Program Synthesis. arXiv:2203.13474, 2022, 22 p.

[21] Hindle A., Barr E.T. et al. On the naturalness of software. Communications of the ACM, vol. 59, issue 5, 2016, pp. 122-131.

[22] Gao L., Biderman S. et al. The Pile: An 800GB Dataset of Diverse Text for Language Modeling. arXiv:2101.00027, 2020, 39 p.

[23] Wang Z., Cuenca G. et al., MCoNaLa: A Benchmark for Code Generation from Multiple Natural Languages. arXiv:2203.08388, 2022, 11 p.

[24] Kim J., Nguyen T.D. et al. Pure Transformers are Powerful Graph Learners. arXiv:2207.02505. 2022, 28 p.

[25] Kreuzer D., Beaini D. et al., Rethinking Graph Transformers with Spectral Attention. arXiv:2106.03893, 2021, 18 p.

[26] Dwivedi V.P., Bresson X. A Generalization of Transformer Networks to Graphs. arXiv:2012.09699, 2021, 8 p.

[27] Dwivedi V.P., Bresson X. Graph Transformer Architecture, 2022. Available at: https://github.com/graphdeeplearning/graphtransformer, accessed: 17.06.2022.

Информация об авторах / Information about authors

German Arsenovich ARUTYUNOV – Master's student at the Faculty of Computer Science at HSE University. Research interests include programming language generation and programming language understanding using machine learning and deep neural networks.

Герман Аренович АРУТЮНОВ – студент магистратуры факультета компьютерных наук НИУ ВШЭ. Сфера научных интересов: генерация и анализ языков программирования посредством машинного обучения и глубоких нейронных сетей.

Sergey Mikchailovitch AVDOSHIN – Candidate of Technical Science, Professor of the School of Computer Engineering at Tikhonov Moscow Institute of Electronics and Mathematics HSE

University. Research interests include design and analysis of computer algorithms, simulation and modeling, parallel and distributed processing, machine learning.

Сергей Михайлович АВДОШИН – кандидат технических наук, профессор департамента компьютерной инженерии Московского института электроники и математики им. А.Н. Тихонова НИУ ВШЭ. Сфера научных интересов: разработка и анализ компьютерных алгоритмов, имитация и моделирование, параллельные и распределенные процессы, машинное обучение.