A Systematic Evaluation Of Large Language Models Of Code

摘要

近来，代码的大型语言模型 (LMs) 在代码补全和从自然语言描述中生成代码方面已经显示出了巨大的潜力。然而，目前最先进的代码大语言模型（如 Codex）并没有公开，这使得许多有关于它们的模型和数据决策问题尚未得到解答。该文章的目标是通过对跨各种编程语言的最大现有模型进行系统评估来填补一些空白，这些模型为 Codex、GPT-J、GPT-Neo、GPT-NeoX-20B 和 CodeParrot。尽管 Codex 本身不是开源的，但是作者发现现有的开源模型在某些编程语言中确实取得了接近的结果，虽然主要针对自然语言建模。作者进一步确定了该研究领域一个重要的缺失部分，即一个基于多语言代码语料库专门训练的大型开源模型。作者发布了 PolyCoder 这一新的模型，它基于 GPT-2 架构，具有 2.7B 个参数，在单台机器上针对 12 种编程语言的 249GB 代码进行了训练。其中在 C 语言编程中，PolyCoder 胜过了包括 Codex 的所有模型。

关键词：大型语言模型；代码生成

1 引言

近年来，随着人工智能和机器学习技术的迅猛发展，特别在在自然语言处理领域 (NLP)，LM 在代码补全或者自动文本生成等任务中显现出了前所未有的潜力，这种潜力体现在它们能够理解和生成复杂的代码结构，以及从自然语言描述中转化为有效的代码。这是通过预测下一个字符 (tokens) 或生成文本的一部分这一操作实现的。

当前 SOTA 大型语言模型 [1] 在基于 AI 的编程辅助领域中取得重要的进展。此外，OpenAI 推出的 Codex [4] 已经部署在了现实的生产工具 GitHub Copilot 中，作为 IDE 内置开发者助手，根据用户上下文自动生成代码。

尽管代码的大型语言模型取得了巨大的成功，但最强大的模型并不是公开可用的。这阻止了这些模型在运算资源充足的公司以外的应用，也限制了资源匮乏的组织在这个领域的研究。以 Codex 为例，它通过黑盒 API 调用提供了该模型输出的收费访问，但模型的权重和训练数据不可用。这阻止了研究人员微调模型，无法适应代码生成之外的领域和任务。无法访问模型的内部也组织了研究团体研究它们的其他关键方面，例如可解释性、用于实现更高效部署的模型蒸馏以及融合检索等额外组件。

与此同时，GPT-J、GPT-Neo [3]、GPT-NeoX [2] 等中等和大规模预训练语言模型是公开可用的。尽管这些模型是在包括新闻文章在内的多样化文本、在线论坛以及少量的 GitHub 软

件存储库的混合资源上训练的，但它们可以用于生成具有合理性能的源代码。此外，还有一些仅在源代码上进行训练的全新开源语言模型。例如 CodeParrot [6] 是基于 180GB 的 Python 代码训练的。

考虑到这些模型中涉及的模型大小和训练方案的多样性，以及彼此之间缺乏比较，许多建模和训练设计决策的影响仍不清楚。基于现有代码的大型语言模型的局限性和开源模型的需求，作者认为系统评估这些模型是必要的，通过比较和对比各种模型，希望为代码建模设计决策前景提供更多的信息。并且为了填补没有大型开源语言模型纯粹根据几种编程语言的代码进行训练这一空白，而引入了新的开源模型 PolyCoder。

2 相关工作

2.1 预训练模型

代码语言建模中有三种常用的预训练模型，图 1 显示了这些方法的示例。

从左到右的语言模型：该类模型即自回归式方法。自回归的从左到右的 LMs，是根据前面的 tokens 来预测下一个 token 的概率。在代码建模中，例如 CodeGPT(124M)、CodeParrot(1.5B)、GPT-Neo(2.7B)、GPT-J(6B)、Codex(12B)、GPT-NeoX(20B) 以及 Google(137B) 都属于这一类。这些模型从左到右的性质使得它们非常适合程序生成任务，比如代码补全。另一方面，由于代码通常不是一个单一的从左到右的编写过程，所以利用出现在生成位置“之后”的上下文并不容易。在这篇论文中，作者主要关注的就是这一类预训练方法所属的模型。

屏蔽语言模型：虽然自回归语言模型对于序列概率的建模非常强大，但它单向的性质使得它不太适合为分类等下游任务生成有效的全序列表示。在表示学习中广泛使用的一种流行的双向目标函数是屏蔽语言建模 [5]，它的目的是根据周围上下文来预测屏蔽的文本片段。CodeBERT(125M) 和 CuBERT(345M) 是这类代码模型的应用示例。在编程环境中，这些方法为下游任务，例如代码分类、克隆检测和缺陷检测，提供了有用的代码序列表示。

编码器-解码器模型：编码器-解码器模型首先使用编码器对输入序列进行编码。然后使用从左到右的语言模型，根据输入序列为条件，进行输出序列的解码。流行的预训练目标包括屏蔽跨度检测，其中输入序列被随机屏蔽，并用多个掩码进行标记，输出序列是按顺序的屏蔽内容。另外，去噪序列重构也是一种预训练目标，其中输入为一个损坏的序列，期望输出为原始序列。这些预训练模型在许多序列到序列的任务中非常有用。CodeT5(220M) 和 PLBART(406M) 分别实现了上述的两个目标，并且在例如代码注释或自然语言到代码生成这类条件生产的下游任务中表现良好。

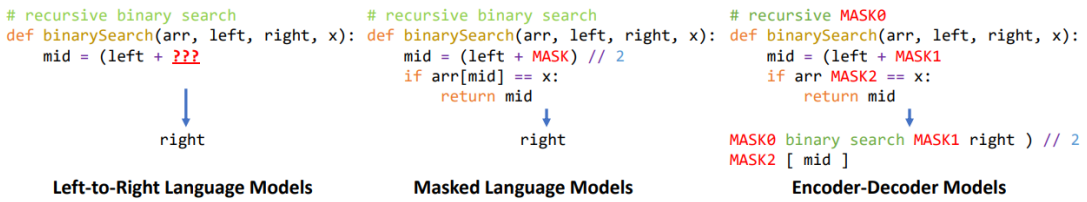


图 1. 三种预训练模型

2.2 HumanEval

HumanEval 是一种用于评估代码合成模型的方法，专门设计用于测试这些模型生成功能性编程解决方案的能力 [4]。

HumanEval 包括一套编程任务的基准集，每个任务都有描述、一组单元测试来验证功能的正确性，以及要实现的函数的签名。这个数据集是由 164 条样本构成的，其中数据格式及命名如下：task_id 表示任务的 ID，prompt 表示题目，这通常由直接请求大模型的途径来获取答案，entry_point 是唯一标记，canonica_solution 是参考答案，test 是测试单元。所有问题都是手写的，而不是通过编程从现有资源复制的。

```python
def incr_list(l: list):
    """Return list with elements incremented by 1.
    >>> incr_list([1, 2, 3])
    [2, 3, 4]
    >>> incr_list([5, 3, 5, 2, 3, 3, 9, 0, 123])
    [6, 4, 6, 3, 4, 4, 10, 1, 124]
    """
    return [i + 1 for i in l]
```

```python
def solution(lst):
    """Given a non-empty list of integers, return the sum of all of the odd elements
    that are in even positions.

    Examples
    solution([5, 8, 7, 1]) ==>12
    solution([3, 3, 3, 3, 3]) ==>9
    solution([30, 13, 24, 321]) ==>0
    """
    return sum(lst[i] for i in range(0,len(lst)) if i % 2 == 0 and lst[i] % 2 == 1)
```

图 2. HumalEval 数据集的两个示例问题

模型的任务是生成函数的主体。模型的性能基于通过所提供测试的生成函数的百分比来评估，反映了其在现实编码场景中的能力。

$pass@k$ 指标即每个问题生成 k 个代码样本。如果有任何一个样本通过测试，则认为问题已经解决，并报告总分数。因为一次实验随机性太大，需要多次实验求平均值。$pass@k$ 需要对每一个测试问题重复实验 t 次，并且每次都生成 k 个代码，最后计算平均通过率。例如重复实验 100 次来估计 $pass@100$ ，就需要生成 100*100=10000 个代码，计算量较大。而 t 越小，估计的 $pass@k$ 的方差就会越大。为了评估 $pass@k$ ，会为每个任务生成 $n \geq k$ 个样本例如使用 $n = 200$ ，$k \leq 100$ ，用以计算通过测试的正确样本的数量 $c \leq n$ ，并计算无偏估计值。

$$\text{pass@}k := \mathbb{E}_{\text{Problems}}\left[1 - \binom{n-c}{k} \Big/ \binom{n}{k}\right]$$

这种评估方法被广泛用于评估语言模型在编程相关任务中的有效性。

3 本文方法

3.1 本文方法概述

拟定部署开源的 Polycoder 模型。具体方法是通过作者已公开的 GPT-NeoX 分叉出的工具包及公开托管的预训练检查点部署经过预训练的模型到本地，并以此生成代码及复制论文的评估。

3.2 评估设置

作者使用外部和内在基准评估所有模型。

外部评估：对于代码建模，最受欢迎的下游任务之一是基于自然语言描述的代码生成。作者使用 HumanEval 数据集评估所有模型。为了在给定提示的情况下生成代码，本文作者沿用了与 HumanEval 示例测试相同的采样策略，即使用带有温度参数 $softmax(x/T)$ 的 softmax 函数。作者使用一系列不同的温度 $T = [0.2; 0.4; 0.6; 0.8]$ 来控制模型预测的置信度。同时使用 nucleus 采样并设置 $top - p = 0.95$ 。另外，作者从模型中采样 tokens，直到遇到以下表示方法结束的停止序列之一：’ \nclass’，’ \ndef’，’ \n#’，’ \nif’ 或’ \nprint’。评估规模为在评估数据集中的每个提示下随机采样 100 个示例。

内在评估：为了评估不同模型的内在性能，作者计算了不可见的 GitHub 存储库上每种语言的困惑度。为了防止 GPT-Neo 和 GPT-J 等模型出现从训练到测试数据的泄露，移除了评估数据集中出现在 Pile 训练数据集的部分。作者为评估数据集中的 12 种编程语言分别抽取了 100 个随机文件。为了使不同模型中使用的不同标记化方法之间的困惑度具有可比性，作者调用了 Pygments 库来等同地标准化每个模型的对数似然和，以计算困惑度。

3.3 模型比较

作者主要选取了自回归预训练语言模型，因为这类模型最适合代码补全任务。首先对 PolyCoder、开源大模型和两个非开源的大模型 Codex、Austin’21 这些现有的 LMs 进行大小和可用性的展示。

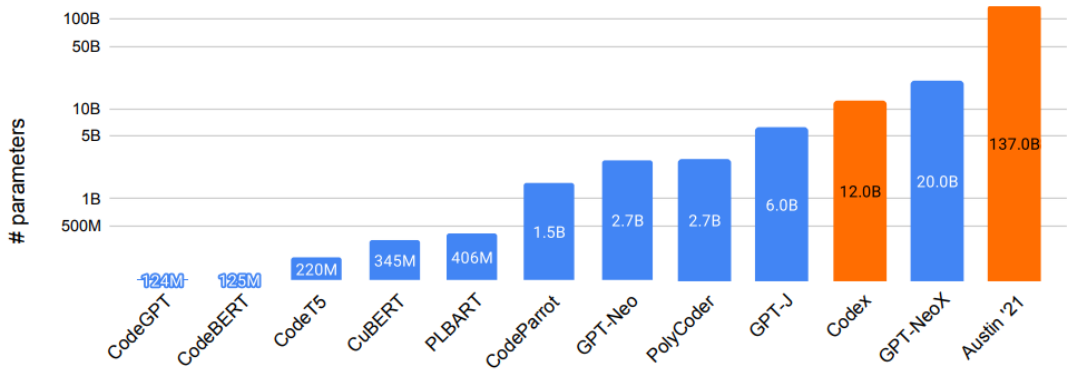


图 3. 现有代码模型大小及可用性

具体地，对于非开源模型，作者评估了 Codex 这一 OpenAI 开发的，并已部署广泛应用并展现卓越性能的模型。该模型是在包含了截止到 2020 年五月从 GitHub 中获得 5400 万个

公开的 Python 存储库的 179GB 的数据集中进行训练的。对于开源模型，作者选取了 GPT 的三种变体模型——GPT-Neo（27 亿参数）、GPT-J（60 亿参数）和 GPT-NeoX（200 亿参数）。其中 GPT-NeoX 是目前可用的最大规模的开源预训练语言模型。这些模型都在 Pile 数据集上进行训练。CodeParrot（15 亿参数）的训练数据集是从 Google BigQuery Github 数据库中获取了超过 2000 万个 Python 文件所形成的一个 180GB 的数据集。该数据集与 Codex 的 Python 训练数据集大小相当，但模型本身要小得多。因为没有专门针对多种编程语言的代码进行训练的大规模开源语言模型，因此作者在 GitHub 上 12 种不同编程语言的存储库数据集上训练了一个 27 亿参数的模型 PolyCoder。文章将 PolyCoder 与 CodeParrot 和 Codex 的数据预处理策略进行对比。具体来说，是过滤掉非常大和非常小的文件，并删除了重复数据。

	PolyCoder	CodeParrot	Codex
Dedup	Exact	Exact	Unclear, mentions “unique”
Filtering	Files > 1 MB, < 100 tokens	Files > 1MB, max line length > 1000, mean line length > 100, fraction of alphanumeric characters < 0.25, containing the word ”auto-generated” or similar in the first 5 lines	Files > 1MB, max line length > 1000, mean line length > 100, auto-generated (details unclear), contained small percentage of alphanumeric characters (details unclear)
Tokenization	Trained GPT-2 tokenizer on a random 5% subset (all languages)	Trained GPT-2 tokenizer on train split	GPT-3 tokenizer, add multi-whitespace tokens to reduce redundant whitespace tokens

图 4. 数据预处理策略对比

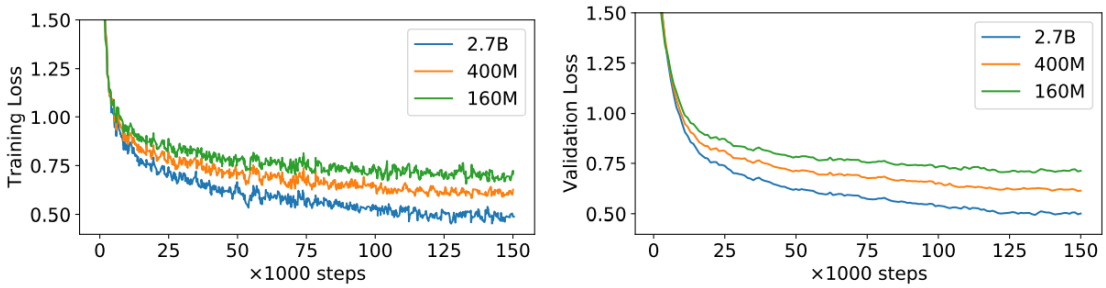
3.4 模型训练及拟得到结果

考虑到预算，作者选择将 GPT-2 作为 PolyCoder 的模型架构。为了探究模型大小缩放的影响，分别训练了参数量为 1.6 亿、4 亿和 27 亿的 PolyCoder 模型。同时，使用了 27 亿参数的模型与 CodeParrot 和 Codex 模型进行设计决策和超参数等方面的比较。

	PolyCoder (2.7B)	CodeParrot (1.5B)	Codex (12B)
Model Initialization	From scratch	From scratch	Initialized from GPT-3
NL Knowledge	Learned from comments in the code	Learned from comments in the code	Natural language knowledge from GPT-3
Learning Rate	1.6e-4	2.0e-4	1e-4
Optimizer	AdamW	AdamW	AdamW
Adam betas	0.9, 0.999	0.9, 0.999	0.9, 0.95
Adam eps	1e-8	1e-8	1e-8
Weight Decay	-	0.1	0.1
Warmup Steps	1600	750	175
Learning Rate Decay	Cosine	Cosine	Cosine
Batch Size (#tokens)	262K	524K	2M
Training Steps	150K steps, 39B tokens	50K steps, 26B tokens	100B tokens
Context Window	2048	1024	4096

图 5. 模型训练参数比较

复现中拟定得到有关 1.6 亿、4 亿和 27 亿参数量 PolyCoder 模型的训练和验证损失曲线。



(a) Training (b) Validation

图 6. PolyCoder 的训练和验证损失曲线

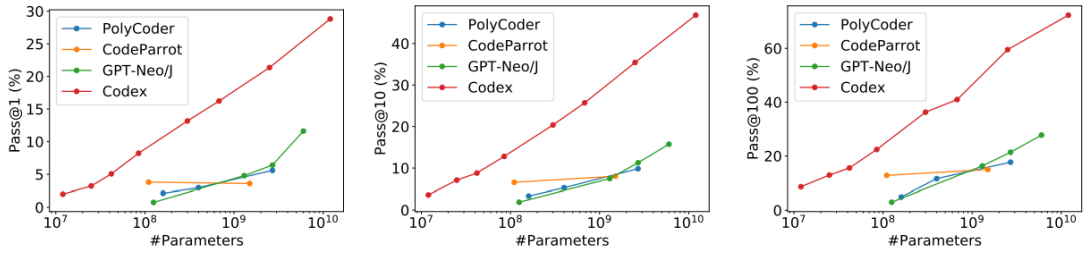
文章中选用 27 亿参数量 PolyCoder 有关外部评估的结果

Model	Pass@1	Pass@10	Pass@100	Tokens Trained	Code Tokens	Python Tokens
PolyCoder (160M)	2.13%	3.35%	4.88%	39B	39B	2.5B
PolyCoder (400M)	2.96%	5.29%	11.59%	39B	39B	2.5B
PolyCoder (2.7B)	5.59%	9.84%	17.68%	39B	39B	2.5B
CodeParrot (110M)	3.80%	6.57%	12.78%	26B	26B	26B
CodeParrot (1.5B)	3.58%	8.03%	14.96%	26B	26B	26B
GPT-Neo (125M)	0.75%	1.88%	2.97%	300B	22.8B	3.1B
GPT-Neo (1.3B)	4.79%	7.47%	16.30%	380B	28.8B	3.9B
GPT-Neo (2.7B)	6.41%	11.27%	21.37%	420B	31.9B	4.3B
GPT-J (6B)	11.62%	15.74%	27.74%	402B	30.5B	4.1B
Codex (300M)	13.17%	20.37%	36.27%	100B*	100B*	100B*
Codex (2.5B)	21.36%	35.42%	59.50%	100B*	100B*	100B*
Codex (12B)	28.81%	46.81%	72.31%	100B*	100B*	100B*

*Codex is initialized with another pretrained model, GPT-3.

图 7. 外部评估结果

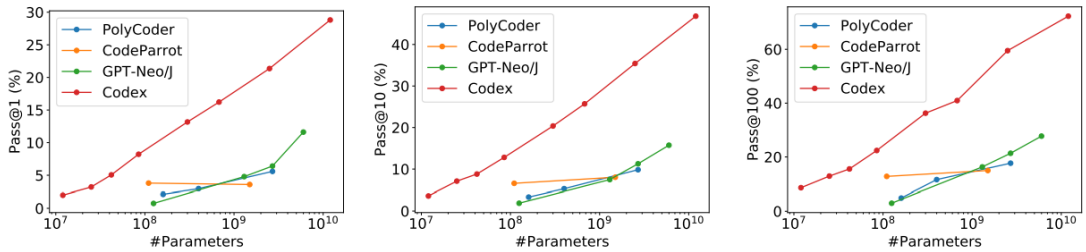
以及 27 亿参数 PolyCoder 选用不同评估标准时的表现



(a) Pass@1 (b) Pass@10 (c) Pass@100

图 8. 不同评估标准的表现

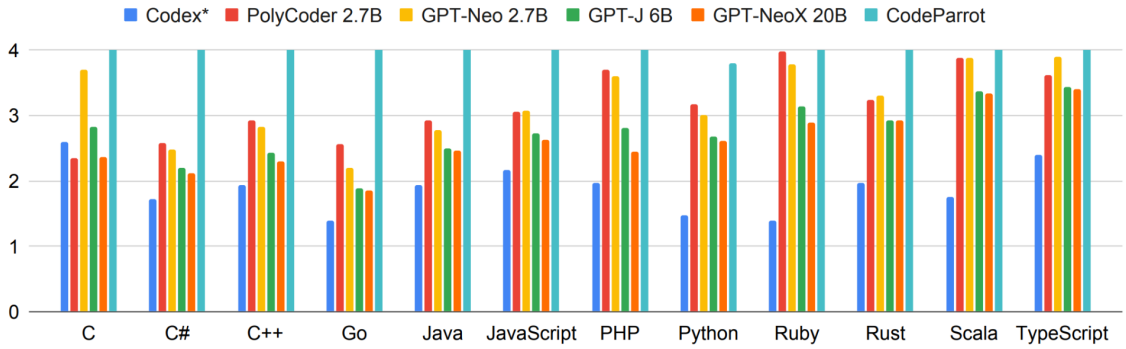
三个不同大小的模型有关温度的影响



(a) Pass@1 (b) Pass@10 (c) Pass@100

图 9. 不同温度的影响

有关内在评估的结果



* Since the exact training set of Codex is unknown, it may include files from these test sets rendering Codex's results overly-optimistic.

图 10. 内在评估结果

4 复现细节

4.1 与已有开源代码对比

由于之前对该方向没有了解，主要抱有在复现该选定的大模型中学习有关代码的大型语言模型相关知识的目的。希望实现文中提及的 PolyCoder 模型及其所比较的 CodeParrot 及 GPT-Neo 模型。但由于初次进行大模型的部署，导致配置过程中产生了不少问题。与已有开源代码对比，仅作了部分本地化部署的参数修改。

4.2 实验环境搭建

在 GitHub 上下载 PolyCoder 的 GPT-NeoX 分叉，根据源代码库尝试搭建 PolyCoder 模型。

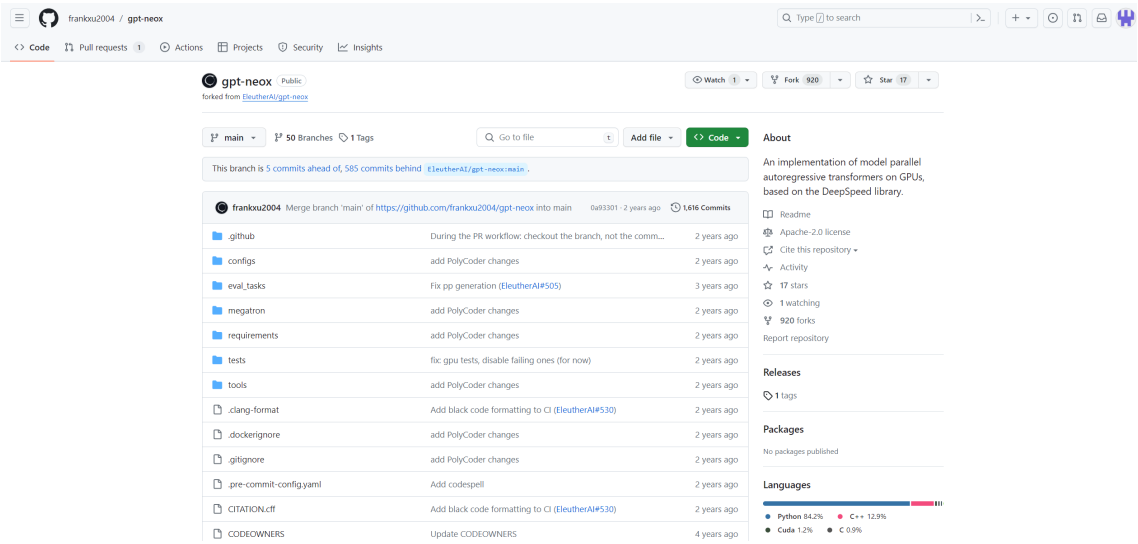


图 11. PolyCoder 的 GPT-NeoX 分叉

由于部署过程中需要安装 deepspeed 库，按照分叉要求，GPT-NeoX 为 1.0 版本。其中 PolyCoder 要求 deepspeed 版本为 0.3.15。但使用文档对应要求的 eb7f5cff 版本，以源码方式进行按照时提示找不到 op-builder 模块而报错。采用 pip install 方式直接安装 0.3.15 版本则出现兼容问题。若采用 GPT-NeoX2.0 版本，尽管可以成功部署 GPT-Neox 环境，但在部署 PolyCoder 环境时候出现了格式错误。因此最终复现仍停留在部署 PolyCoder 这一起始阶段。

5 实验结果分析

因为实验停留在部署阶段，并未能进行后续的评估测试。但使用 HuggingFace 封装好的 PolyCoder 模型，进行简单代码生成测试，可得如下生成代码。在生成简单代码情况下，三种不同参数的 PolyCoder 模型表现仍有一定差距。

```
def binarySearch(arr, left, right, x):
    mid = (left + right) / 2
    if arr[mid] < x:
        return mid
```

1.6亿参数

```
def binarySearch(arr, left, right, x):
    mid = (left + right) / 2
    if x < arr[mid]:
        return mid
```

4亿参数

```
def binarySearch(arr, left, right, x):
    mid = (left + right) // 2
    if arr[mid] == x:
        return mid
```

27亿参数

图 12. 简单测试生成代码

6 总结与展望

尽管未能完成复现的实验任务，但在论文信息收集及阅读阶段对代码的大型语言模型有了初步的认识。在部署过程中遇到了各种的问题，通过查阅文档及代码排除错误的阶段了解了大模型参数设置及修改的要点。期望能学习其他部署方式或排错方式，或接触其他开源模型的部署，在今后经验增长后重新查看如何部署 PolyCoder 这一模型。

参考文献

[1] Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, et al. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*, 2021.

[2] Sid Black, Stella Biderman, Eric Hallahan, Quentin Anthony, Leo Gao, Laurence Golding, Horace He, Connor Leahy, Kyle McDonell, Jason Phang, et al. Gpt-neox-20b: An open-source autoregressive language model. *arXiv preprint arXiv:2204.06745*, 2022.

[3] Sid Black, Leo Gao, Phil Wang, Connor Leahy, and Stella Biderman. Gpt-neo: Large scale autoregressive language modeling with mesh-tensorflow. *If you use this software, please cite it using these metadata*, 58, 2021.

[4] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

[5] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

[6] Lewis Tunstall, Leandro Von Werra, and Thomas Wolf. Natural language processing with transformers. 2022.