arXiv:2305.05383v1 [cs.PL] 8 May 2023

预训练语言模型的代码执行

Chenxiao Liu[1*], **Shuai Lu**[2], **Weizhu Chen**[2], **Daxin Jiang**[2],
Alexey Svyatkovskiy[2], **Shengyu Fu**[2], **NeelSundaresan**[2], **Nan Duan**[21]
北京大学 [2] 微软jadecxliu@gmail.com{shuailu,
wzchen,djiang}@microsoft.com{alsvyatk,
shengyfu,neels,nanduan}@microsoft.com

摘要

代码执行是编程语言语义的一个基本方面，反映了代码的确切行为。然而，大多数用于代码智能的预训练模型都忽略执行轨迹，仅依赖源代码和句法结构。本文研究预训练模型理解和执行代码的能力。我们开发了一种基于变异的数据增强技术，用于构建大规模且真实的 Python 代码执行数据集和任务，这对 Codex 等现有模型构成了挑战。随后，我们提出了 CodeExecutor，这是一种 Transformer模型，利用代码执行预训练和课程学习来增强语义理解。我们在代码执行任务上评估 CodeExecutor，展示了其令人鼓舞的性能和局限性。我们还展示了它在代码智能任务中的潜在优势，例如零样本代码到代码搜索和文本到代码生成。我们的分析揭示了预训练模型在代码执行方面的学习能力和泛化能力。

1 引言

预训练模型在自然语言（NL）任务中取得了显著成果（Radford 等人，2018年；Devlin 等人，2019年；Raffel 等人，2020年），推动了面向编程语言（PL）任务的预训练模型的发展（Kanade 等人，2020年；Feng 等人，2020年；Svyatkovskiy 等人，2020年；Wang 等人，2021年b；Guo 等人，2021年、2022年）。这些模型利用源代码和代码结构（例如抽象语法树（AST）（Wang 等人，2021年a；Guo 等人，2022年）以及数据流（Guo 等人，2021年）），来学习代码相关任务。这些结构虽然有用，却不足以表示代码在执行过程中的动态行为，而这种行为会反映在执行轨迹中。以图1为例，执行轨迹展示了代码在执行过程中的行为，反映了控制流以及变量的状态变化。另一方面，正如Casalnuovo 等人（2020年）所述，源代码包含两个信息通道：自然通道和形式通道。自然通道（Hindle 等人，2012年），例如标识符和注释，使语言模型能够用于理解代码相关任务。形式通道由解释器和编译器用于指定执行过程，并具有精确语义。形式通道是代码所独有的，也是代码能够执行的关键。执行轨迹属于第二类，因为它揭示了能够区分代码与自然语言的形式通道信息，同时也使代码能够被精确执行（Casalnuovo 等人，2020年；Chakraborty 等人，2022年）。

在这项工作中，我们旨在教会预训练模型真实世界的代码执行过程。我们提出了 CodeExecutor，这是一种基于 Transformer 的模型，能够学习执行任意程序并预测其执行轨迹。为支持大规模数据上的预训练，我们基于 CodeNet 中竞赛编程问题的提交构造了 Python Co- deNetMut 数据集，方法是对这些提交进行变换（Puri等人，2021年），并纳入单行 Python 转换以及根据 Python 官方教程改编的程序。我们设计了一项预训练任务，用于预测执行轨迹中的行顺序和中间状态，并采用课程学习逐步提高程序的难度。

我们在代码执行任务上评估了 CodeExecutor，结果表明它优于现有模型，并展现出令人期待的能力。我们还深入分析了该模型的性能，揭示了它的优势和不足。此外，我们证明，CodeExecutor 能够提升零样本代码到代码搜索和文本到代码生成等下游任务的表现，这表明利用执行轨迹来

*在微软实习期间完成的工作。ShuaiLu 和 Nan Duan 为通讯作者。

```
高 = 3
宽 = 7
n = 10
对于 i 在 range(min(h, w)) 中:
    n = n - max(h, w)
    如果 n <= 0:
        print(i + 1)
        break
```

（a）源代码

```
<行> 1 <状态> h:3
<行> 2 <状态> 高: 3; 宽: 7
<行> 3 <状态> 高: 3; 宽: 7; n: 10
<行> 4 <状态> h:3;w:7;n:10;i:0
<行> 5 <状态> h:3;w:7;n:3;i:0
<行> 6 <状态> h:3;w:7;n:3;i:0
<行> 4 <状态> h:3;w:7;n:3;i:1
<行> 5 <状态> h:3;w:7;n:-4;i:1
<行> 6 <状态> h:3;w:7;n:-4;i:1
<输出> 2
<行> 7 <状态> h: 3; w: 7; n: -4; i: 1
<行> 8 <状态> h:3;w:7;n:-4;i:1
```

（b）执行轨迹

图1：代码执行任务中的示例源代码及其执行轨迹。

增强代码智能。我们的模型和数据集已公开[1]。总而言之，本文的贡献如下：

- 我们提出了首次尝试，采用基于变异的数据增强方法，构建用于真实世界代码执行的大规模预训练数据集。
- 我们提出了一种名为 CodeExecutor 的新型预训练模型，它通过代码执行预训练任务和课程学习，学习预测执行轨迹。
- 我们对 CodeExecutor 在代码执行任务中的表现进行了全面评估，从而详细了解了模型的性能。
- CodeExecutor 显著提升了代码智能任务的表现，例如零样本代码到代码搜索和文本到代码生成。

2 相关工作

2.1 学习执行

以往的工作将学习执行任务构建为一个读取程序并计算其输出的问题。这些工作采用了循环神经网络（Zaremba 和 Sutskever，2014年）、图神经网络（Bieber 等人，2020年；Wang 等人，2020年）和 Transformer模型（Dehghani 等人，2019年；Yan 等人，2020年；Austin 等人，2021年；Nye 等人，2021年）等架构。另一个相关任务——算法归纳——是读取一个简短程序（例如整数加法或多项式求值），并计算其输出。算法归纳任务（Graves 等人，2014年；Kurach 等人，2016年；Kaiser 和 Sutskever，2016年；Graves 等人，2016年；Reed 和 de Freitas，2016年；Dehghani 等人，2019年；Velickovic 等人，2020a，b；Nye 等人，2021年）则针对特定算法提供直接的算法专属监督，与我们的代码执行任务中任意程序的情况形成对比。

一些新兴研究也采用预训练模型来解决这两项任务。Lu 等人(2022) 在非语言任务上对 GPT- 2（Radford 等人，2019年）的少部分权重进行微调，这些任务包括按位异或等简单的算法归纳任务。Austin 等人(2021年) 对预训练模型进行了评估，这些模型在网络文档和对话数据上预训练，参数量从 200 万到 1370 亿不等。结果表明，无论是少样本学习还是微调，最大的模型通常都无法预测程序的输出。Nye 等人(2021年) 使用“草稿区”存储中间计算步骤以执行多步计算，从而提升了模型在 Austin 等人(2021年) 中的能力。

与以往预测程序输出且主要处理特定算法的研究不同，我们预测程序的完整执行轨迹，并专注于模仿现实世界中任意程序的执行行为。此外，通过执行来捕捉代码语义，我们的工作有助于代码智能相关任务。

2.2 数学问题求解

数学问题求解是与代码执行相关的一个领域。近期研究表明，语言模型能够解决数学问题，而这需要学习执行一种软算法，从而得出确定的答案。Amini 等人(2019年);Ling 等人(2017) 将数学问题映射为运算程序，并专注于序列到程序生成。Saxton 等人(2019年) 引入了 DeepMind 数学数据集，其中包含加法、列表排序和函数求值等直接套用方法即可解决的问题。Henighan 等人(2020年)

[1] https://github.com/microsoft/CodeBERT/tree/master/CodeExecutor

	运算符	描述
CRP	常量替换	更改数值和字符串字面量。
AOD	算术运算符删除	删除一元算术运算符‘+’或‘-’。
AOR	算术运算符替换	将一个算术运算符替换为另一个。例如，$x * y$ 可以变异为x / y。
ASR	赋值运算符替换	将一个扩展赋值运算符替换为另一个。
BCR	break/continue 替换	在循环体中交换关键字 *break* 和*continue* 。
COD	条件运算符删除	删除一元否定运算符 *not* 或成员关系的否定运算符不属于。
LCR	逻辑连接符替换	交换逻辑运算符 *and* 与*or* ，反之亦然。
ROR	关系运算符替换	替换关系运算符。例如，$x <= y$ 可以变异为$x > y$。
SIR	切片索引移除	删除 *collection[start:end:step]* 的一个参数。
OIL	单次迭代循环	通过添加*break* 语句，使循环仅执行一次。
RIL	反向迭代循环	通过函数*reversed()*改变循环迭代的方向。
ZIL	零次迭代循环	在循环第一次迭代期间中断循环的执行。

表1：一组包含12个算子的变异算子，用于对代码示例进行变异。

表明，DeepMind 数学数据集中的大多数问题都可以通过大型 Transformer 模型轻松解决。Hendrycks等人(2021年) 推出了 MATH 数据集，其中包含竞赛数学题及以 LaTeX 和自然语言撰写的逐步解答。Cobbe等人(2021年) 发布了 GSM8K，其中包含小学数学题及其自然语言解答。近期，Zhou等人 (2022) 提出算法提示，以提升大语言模型解决数学问题的表现；该方法从学习加法、减法、乘法和奇偶性等技能入手。

代码执行涉及加法、减法、乘法、除法、指数运算和取模等计算，这些计算与解决数学问题相似。由于还要管理变量、数据结构、控制流和其他编程概念，学习代码执行所需的技能和知识与学习数学不同，尽管两者存在一定重叠。

3 基于变异的数据增强

代码代码执行 任务的目标是学习在不通过解释器运行程序的情况下模拟其执行过程。我们将该任务视为一项生成任务：给定源代码 c，需要生成执行轨迹 t 。执行轨迹由两个部分组成：一是计算机执行语句的顺序，二是从一个语句跳转到另一个语句时变量状态如何变化。通常，程序中的语句并非按顺序执行，尤其是在现实场景中，程序往往包含复杂逻辑和丰富语义。此外，变量涉及具有不同特征和操作的各种数据结构。鉴于这项任务的复杂性和难度，构建大规模数据集并探索大语言模型在代码执行方面的能力与局限至关重要。

3.1 修改源代码

构建用于真实世界代码执行的大规模 Python 数据集极具挑战性。从 GitHub [2] 等软件开发平台获取的程序大多无法大规模执行，因为它们依赖于难以获取的特定外部资源。外部资源包括程序输入、文件内容、外部模块和第三方软件包等。出于同样的原因，从 StackOverflow [3]等编程问答网站的帖子中收集程序也不切实际。

我们基于 CodeNet基准中的竞赛编程问题提交构建 Python代码执行数据集（Puri等人，2021年）。我们在沙盒环境中运行每份提交，以获取执行轨迹，并过滤掉超出时间或执行轨迹限制，或导致运行时错误的程序。

为构建大规模可执行程序数据集，我们提出了一种基于变异的数据增强方法。对于每个提交，该方法都会修改程序的部分内容，生成多种变异体，从而产生不同的执行轨迹。这些修改的规范称为变异算子。该方法的灵感来自变异测试 (Hamlet，1977年；Jia和Harman，

[2]https://github.com/[3] https://stackoverflow.com/

2011），这是软件工程中一种广受欢迎的技术，有助于设计高质量的程序测试套件。遵循Derezi´nska和Hałas(2014年)将变异测试技术应用于Python程序的做法，我们首先列出一组变异算子，如表1所示。其中大多数对应于强类型通用语言中使用的特定算子，并针对Python语言进行了调整。此外，还包含针对Python特性设计的算子，例如切片索引移除（SIR）和反向迭代循环（RIL）。接着，我们将程序转换为抽象语法树（AST），并提取其节点类型信息，以获取所有可变字面量、运算符和语句的候选列表。最后，我们生成变异体，并剔除不可执行的变异体。我们使用CodeNet Mutants（CodeNetMut）构建预训练数据集。关于数据集生成过程的更多细节，请参见附录A。

3.2 数据集构建

鉴于在真实世界的完整程序上训练模型难度较大，我们构建了两个更简单的数据集，并与 CodeNetMut 一起用于预训练。

第一个是 Fraser Greenlee 收集的 Python SingleLine数据集[4]，其中包含近九百万个单行转换示例。每个示例包含若干个以初始值指定的变量、一行 Python 代码，以及执行该行后得到的一组新变量及其值。我们将前两者合并作为输入代码，并将最后一项用作目标跟踪。我们不会重新执行该数据集。对 SingleLine 数据进行预训练时，我们只要求模型预测最后一行代码执行后的最终状态，而不提供逐行说明。图2（a）（b）展示了这些数据的示例。由于单独的代码行构成了真实世界复杂程序的一部分，该数据集为学习代码执行奠定了基础。

第二个是 Python Tutorial数据集。该数据集通过抓取并筛选 Python官方教程中出现的所有可执行代码示例而创建 [5]。官方教程介绍了 Python 语言的基本概念和最值得关注的特性。为了生成该数据集，我们应用常量替换算子（表 1 的第一行），将数值字面量替换为各种不同的值。采用这种方法后，最终得到 340 万个程序

[4]https://www.kaggle.com/frasergreenlee/python 状态变化[5]https://docs.python.org/3/tutorial

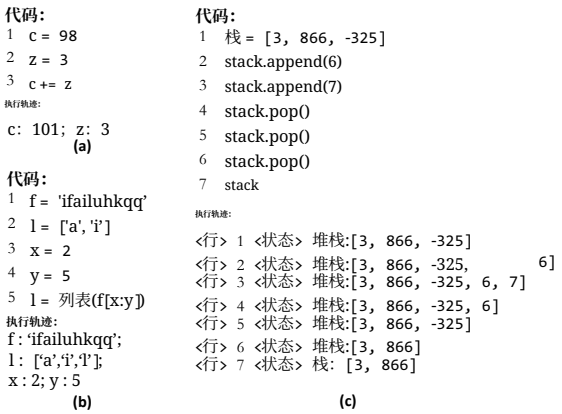


图2：(a) 和 (b) 是 SingleLine 数据集中的示例。(c) 是教程数据集中的示例。

。图2 (c) 展示了一个变异示例。尽管教程数据集并不全面，也未涵盖所有特性，但它很好地体现了 Python 的风格与特点，为模拟常用代码块的执行提供了宝贵的监督信息。

因此，Python 代码执行数据集是一系列遵循由简单到困难范式的数据集，包括 SingleLine 数据集、教程数据集和 CodeNetMut 数据集。

4 CodeExecutor

我们的 CodeExecutor 使用基于 Transformer 的框架，通过预训练来学习代码执行。我们将首先介绍模型架构 (§4.1)，然后介绍预训练任务 (§4.2)，最后介绍课程学习策略 (§4.3)。

4.1 模型架构

该模型基于 Transformer，并采用与 UniXcoder 相同的架构（Guo 等人，2022年）。UniXcoder 是一种面向编程语言的统一跨模态预训练模型，具有仅编码器、仅解码器和编码器-解码器三种模式。它利用带有前缀适配器的掩码注意力矩阵（Dong 等人，2019年）来控制模型行为。我们采用编码器-解码器方式，将特殊标记 $[E2D]$ 作为输入前面的前缀。CodeExecutor 由 12 层 Transformer 组成。每个 Transformer 层的架构完全相同，包含多头自注意力池化（Vaswani 等人，2017），后接一个前馈网络。

4.2 预训练任务

我们提出了一种名为代码执行的新预训练任务。提出这一任务的动机是提升模型理解和执行代码的能力。传统的预训练任务，例如语言建模或去噪目标，并不涉及代码执行，因此，经过这些任务训练的模型执行代码的能力有限。通过在代码执行任务上预训练模型，我们旨在使其从代码和执行轨迹的双模态数据中学习有用模式，从而提升这项能力。这样，模型便能生成更准确的执行轨迹，并理解代码的行为；这对于许多需要理解代码的代码智能应用至关重要。掌握代码的运行方式后，模型能够更好地理解代码的底层逻辑，并运用这种理解更好地完成这些任务。

我们继续在该任务上对 UniXcoder 进行预训练。在预训练阶段，模型接收代码作为输入，并学习生成执行轨迹。为了更好地理解代码，我们在代码中插入特殊标记 $[i]$ 表示行号和 $[INDENT]$ $[DETENT]$ 表示缩进。执行轨迹中的每一行都可以表示为 $[LINE]$，$[i]$，$[STATE]$，v_1，$:s_1$，$[DICTSEP]$，..... $[DICTSEP]$，v_k，$:s_k$，$[STATEEND]$，其中 k 表示变量的数量，而 k 个变量的状态 v_k 为 s_k。符号 $[DICTSEP]$ 用于分隔字典中的键值对，$[STATEEND]$ 表示状态的结束。这种表示方式使模型能够学习执行过程中每一步的变量状态，这对于理解代码的行为至关重要。

4.3 课程学习

为了提升泛化能力，我们在预训练过程中遵循课程学习策略。课程学习（Bengio 等人，2009）（CL）是一种从简单实例开始，再逐步处理更困难实例的学习策略，模仿了人类课程中有意义的学习顺序。在预训练过程中，我们按照从简单实例（即 SingleLine 数据）开始的课程，组织 Python 代码执行数据集的学习。首先，我们使用全部 900 万条 SingleLine 转换数据对 CodeExecutor 进行预训练，直至收敛。为了实现数据集平衡，我们随后保留了 Sin- 中 300万条实例，这些实例最难由我们的模型生成

难度等级	**SingleLine** Easy	**教程** 中等	**CodeNetMut** Hard
语言	Python	Python	Python
预训练 #	8,950,959	3,422,943	2,838,644
测试 #	7,968	13,744	19,541
平均代码长度	3.28	4.90	8.26
平均轨迹长度	1.00	11.89	22.80
平均状态数	2.44	1.34	3.67

表 2：预训练数据集的统计信息。“平均代码长度”和“平均轨迹长度”分别表示程序和执行轨迹中的平均行数。“平均状态数”表示执行轨迹中每行所达到的最大状态数的平均值。

gleLine 中最难由我们的模型生成的实例，并将 Tutorial 数据加入预训练语料库。我们进一步将 CodeNetMut 数据加入预训练语料库，并使用所有样例对模型进行预训练，直至收敛。为便于区分难度等级，我们在输入前添加前缀 $p \in \{[SINGLELINE], [TUTORIAL], [CODENETMUT]\}$ 用于指示数据类型，例如 $[SINGLELINE]$ 表示接收 SingleLine 数据。有关预训练设置和模型配置的更多详情，请参阅附录 B。

5 实验设置

5.1 数据集

我们按照第 3 节所述构建预训练数据集。表 2 展示了一些基本统计数据。CodeNetMut 测试集中的 19,541 个示例来自 CodeNet 中的 39 个未见过的编程问题，且未经过变异过程。此外，在预训练期间，我们从每个数据集中留出了 1 万个程序作为验证集划分。对于 Tutorial 和 CodeNetMut，真实轨迹是整个程序的执行结果。对于 SingleLine，由于其实例是由变量声明和单行转换组成的简单程序，因此模型只需预测变量的最终状态，其形式为单行执行轨迹。我们观察到，CodeNetMut 中代码和执行轨迹的平均长度约为 Tutorial 中的两倍。此外，执行 CodeNetMut 中的程序需要管理数量更多且状态各异的变量。

5.2 模型

我们在代码执行任务上评估了多个模型。**Codex** 模型 `code-cushman-001`是一种专门的 GPT 模型，经过 GitHub 代码微调（Chen 等人，2021年）。我们采用少样本学习

数据集	模型	通用		Line			标识符		
		输出准确率	轨迹准确率	精确率	召回率	F1	精确率	召回率	F1
SingeLine	Codex	-	36.87	36.87	36.87	36.87	71.87	69.34	70.58
	CEL-S1	-	93.32	93.32	93.32	93.32	96.94	96.86	96.90
	CodeExecutor	-	**94.03**	**94.03**	**94.03**	**94.03**	**97.28**	**97.18**	**97.23**
教程	Codex	13.07	-	-	-	-	-	-	-
	CEL-S2	**79.51**	**85.59**	**95.94**	**84.24**	**89.71**	**97.29**	**87.30**	**92.02**
	CEL-S3	7.89	8.35	26.58	21.33	23.67	26.36	19.47	22.40
	CodeExecutor	76.42	80.09	94.49	76.74	84.70	95.91	69.15	80.36
CodeNetMut	Codex	17.45	-	-	-	-	-	-	-
	CEL-S3	43.80	29.44	59.32	41.76	49.01	68.30	41.69	51.78
	CodeExecutor	**48.06**	**33.38**	58.70	**43.48**	**49.96**	67.81	**45.29**	**54.31**
	-不含 CL	45.93	30.98	**60.21**	42.45	49.79	**68.55**	41.58	51.76

表3：代码执行任务的结果。在 Tutorial 和 CodeNetMut 数据集中，Codex 无法以统一格式生成执行轨迹。因此，我们仅报告 Codex 在这些数据集上的输出准确率。

，为 Codex 提供三个代码及执行轨迹对，用于代码执行任务。CodeExecutor-Limited（**CEL**）是一个采用代码执行目标进行预训练的三阶段模型。CEL 在每个阶段只能访问有限的数据，而**CodeExecutor** 则可以同时利用所有数据集（详见附录C 中的详细比较）。该模型以公开可用的 UniXcoder 检查点为初始参数，并继续使用 SingleLine 数据进行训练，得到模型 CodeExecutorLimited-Stage1，我们将其称为**CEL-S1**。在第二阶段，我们以 CEL-S1 为初始模型，使用 Tutorial 数据进行预训练，从而得到模型**CEL-S2**。在第三阶段，我们继续对 CEL-S2 进行预训练，并使用 CodeNetMut 来提升执行真实世界程序的能力。**经过上述阶段后得到**CEL-S3。**不采用课程学习的 CodeExecutor**（CodeExecutor w/o CL）是一个将全部三个数据集一起用于训练的单阶段模型。

5.3 评估指标

我们在来自三个数据集的测试集上测试模型执行代码的能力，并从三个方面衡量采样执行轨迹的功能正确性。我们报告输出准确率和执行轨迹准确率，以评估整体表现。**输出准确率**用于检查模型输出的消息是否与代码执行结果相同，该指标仅针对包含标准输出的程序计算。**执行轨迹准确率**用于检查模型生成的执行轨迹是否与代码执行轨迹相同，不考虑轨迹中同一行内各状态的顺序。为了评估轨迹中每一行的正确性以及标识符的状态，我们还评估逐行得分和标识符得分。**行精确率**由正确识别的行数占模型生成的执行轨迹中所有行数的比例决定。**行召回率** 是模型正确预测的行数占真实轨迹中所有行数的比例。同样，我们也计算执行轨迹中标识符的得分。

为了深入了解模型行为和错误模式，我们还通过检查样本进行定性分析。

我们从测试集中随机抽取 50 对代码与执行轨迹，并请两位拥有至少 5 年经验的程序员从 7 个方面评估 CodeEx- ecutor 是否能正确执行程序。类别 基础知识 包括 Python 初学者需要掌握的基础知识，例如数学运算符、增强赋值运算符、比较运算符和变量。类别列表、元组等 包含典型的 Python数据结构，例如列表、元组、字典、集合及相关操作函数。如表 4 所示，我们构建了分类体系，并编写手册以指导分类。每位审阅者逐行检查生成的执行轨迹，并统计每个类别出现的频次。如果某一执行轨迹行涉及多个类别，他们会统计所有相关类别。出现错误时，他们会判断模型弄错了哪类知识。最后，他们共同讨论错误归因上的分歧，并达成一致。

6 结果与分析

在本节中，我们评估 CodeExecutor 在代码执行任务上的表现（§6.1），深入分析以了解模型行为和错误模式（§6.2），随后探讨两个下游任务（§6.3）。

代码：

```
rec 1 = [ 10 3 5 ] '', '', '' n, a, b = map(int, rec) 2nin 3 = [ ] a, b n max = min(nin) 4nmin = n - min(n, (n-nin[0])+ (n-nin[1])) 5print(str(nmax) + " " + str(nmin)) 6
```

预测：

<行> 1 <状态> 记录: [10, 3, 5]</行> 2 <状态> 记录: [10, 3, 5]; n:10; a:3; b:5</行> 3 <状态> 记录: [10, 3, 5]; n:10; a:3; b:5; nin:[3, 5]</行> 4 <状态> 记录: [10, 3, 5]; n:10; a:3; b:5; nin:[3, 5]; nmax:3</行> 5 <状态> 记录: [10, 3, 5]; n:10; a:3; b:5; nin:[3, 5]; nmax:3; nmin:2</输出> 3 2</行> 6 <状态> 记录: [10, 3, 5]; n:10; a:3; b:5; nin:[3, 5]; nmax:3; nmin:2

图3：来自 CodeNetMut 测试集的一个示例，其中 CodeExecutor 给出了不完美的预测，错误部分以<u>下划线</u>标出。

6.1 总体结果

我们评估模型在单行、教程和 CodeNetMut 数据集上的性能。

我们在**SingleLine** 的结果见表3（上）。CodeExecutor 能够正确执行约 94% 的单行转换，而 Codex 在大多数情况下都无法做到。与 CEL-S1 相比，CodeExecutor 的表现还提升了 0.7%，这表明在预训练期间学习困难程序有助于更好地解决简单示例。由于每个 SingleLine 程序始终会生成一行执行轨迹，且没有标准输出，因此我们不报告输出准确率；行精确率/召回率与执行轨迹准确率相同。

对于表**3**中的教程实验（中等），CodeExecutor 的输出准确率显著优于 Codex（76.42% 对 13.07%）。CodeExecutor 的得分低于 CEL-S2，说明教程中的代码示例与 CodeNet 中的代码示例存在差异，因为教程数据集仅由教程网站上少数几个程序的变异体构成，多样性有限。CEL-S3 难以生成执行轨迹，这表明它在最后一个训练阶段忘记了在教程数据中获取的大部分知识。

CodeNetMut 的结果远低于 SingleLine 和 Tutorial 数据集，这表明在真实场景中生成执行轨迹更具挑战性。CodeExecutor 对近一半的示例生成了正确输出（48.06%），约三分之一的执行轨迹与真实值完全匹配（33.38%）。通过在代码执行任务上进行预训练，CodeExecutor 的输出性能比 Codex 高出 30.6 个百分点。此外，CodeExecutor 的输出准确率得分和执行轨迹准确率得分分别比 CEL-S3 高 4.3% 和 3.9%，这表明第 4.3 节所述训练策略的有效性。移除课程学习后，输出准确率得分从 48.06% 降至 45.93%，执行轨迹准确率得分从 33.38% 降至 30.98%，这体现了其贡献

类别	总数	正确数	准确率
基础	204	183	89.71
内置函数	42	35	83.33
列表、元组等	44	34	77.27
字符串	19	10	52.63
条件语句	60	57	95.00
循环	25	21	84.00
函数调用	5	5	100.00

表4：人工评估结果。为评估 CodeExecutor 的能力，我们将 Python 编程知识分为七类，并手动分析在处理这些类别时生成的执行轨迹是否正确。第三类包括 Python 数据结构，例如列表、元组、字典和集合。

课程学习的

这些结果表明，对于 Codex 等源代码预训练模型而言，代码执行任务颇具挑战性。然而，我们的 CodeExecutor 模型能够高性能地执行简单程序，并且能够预测真实世界程序的复杂执行轨迹。

6.2 模型性能深入研究

我们通过检查样本（表 4）对模型性能进行了定性分析，得出以下发现。更多示例请参见附录 D。

模型通常对控制流有基本的理解条件语句、循环和函数调用揭示了程序的控制流。控制流反映了程序代码的执行顺序。理解程序离不开控制流，而控制流通常较为复杂，因为它通过特定决策控制代码，并监控哪些语句需要执行、哪些应当跳过。从表4中可以看出，CodeExecutor 对高级多行控制流有初步理解，尤其擅长处理条件语句和函数调用。在 60 个条件语句中，有 57 个被正确预测，对用户定义函数的 5 次调用也全部预测

模型	MAP
GraphCodeBERT	23.08
+ CodeExecutor	**55.94**
UniXcoder	71.86
+ CodeExecutor	**79.13**

表5：零样本设置下代码到代码搜索任务的 MAP 分数（%）。

模型	Pass@1	Pass@10
Codex	12.48	45.59
+ CodeExecutor	**17.87**	**49.69**

表 6：文本到代码生成任务在 HumanEval 基准测试上的结果。两种设置下每个问题均评估 50 个解决方案。

正确。循环的准确率为 84%，而预测错误的循环则经历了错误的迭代次数。以图1 （a）为例。CodeExecutor 预测出的执行轨迹与（b）中的真实值完全相同。我们的模型识别出，第 4 行出现的 for循环将执行若干次。在第二次迭代中，*“n”* 满足*“n <=0”*条件，从而执行*“break”* 语句并终止循环。模型在 for循环中的代码块上表现良好，展现了其理解控制流的能力。

模型难以处理操作的复杂细节，尤其是涉及数据结构的操作 复杂程序通常涉及多类编程知识。图3 展示了一个使用列表和字符串的示例。已知有*“n”*人同时订阅《报纸I》和《报纸II》，其中*“a”*人订阅《报纸I》，*“b”* 人订阅《报纸II》，该示例用于确定同时订阅两份报纸的人数可能的最大值和最小值。CodeExecutor 在第 5 行错误地计算了*“nmin”*，预期结果为 0，实际却得到 2。这项计算需要从列表中提取值、执行加法和减法运算，并使用“min”函数。这些操作的组合使模型难以完全理解代码并生成准确的状态。此外，如表4所示，模型在“列表、元组等”（77.27%）和“字符串”（52.63%）上的准确率相对较低，这表明模型在理解列表和字符串等数据结构方面存在不足。理解数据结构要求模型掌握对象创建、修改、添加或删除后的行为。这些操作可能变化多端，难以为模型所掌握。这表明，面对涉及多种操作和数据结构的复杂程序时，模型可能难以应对。

6.3 下游任务

为验证 CodeExecutor 在表示代码语义方面的有效性，我们将其应用于两项代码智能任务——零样本代码到代码搜索任务和文本到代码生成任务。

零样本代码到代码搜索 该任务由Guo 等人(2022年为避免关联数据集与我们的预训练语料库之间出现重复，我们通过从 CodeNet（Puri 等人，2021年）中收集 9,987 个 Python 函数，构建了一个新数据集。每个函数解决 48 个问题中的一个。给定一个函数，我们检索所有解决相同问题的函数。

我们首先使用基线模型最后一层隐藏状态的均值向量，计算两个函数之间的相似度。为了探究代码执行如何促进代码到代码的搜索，我们通过提供一个测试用例来执行每个函数。随后，我们利用从 CodeExecutor 生成的执行轨迹中提取的程序输出，并根据这些输出与查询程序输出之间的编辑相似度对候选项进行排序。

从表5中可以看出，与 GraphCodeBERT 相比，CodeExecutor 提升了 32.8 分（Guo 等人，2021年），与 UniXcoder 相比则提升了约 7.2 分。这表明，代码执行可以显著增强对代码语义的理解。

文本到代码生成 我们使用 HumanEval 基准（Chen 等人，2021年），其中包含 164 道由人工编写的编程问题。

我们首先利用 Codex（code-cushman-001）为每个问题生成 200 个解答。然后，将问题描述中的示例测试用例输入 CodeExecutor，以预测每个解答的输出。我们根据这些输出与预期输出之间的编辑相似度，为 200 个解答排序。最后，我们评估每个问题排名前 50 个解答的正确性。需要注意的是，与其他筛选策略不同，我们的方法不需要真实世界的代码执行器，只使用模型来预测执行结果。

表6 展示了将 CodeExecutor 用作解答筛选器能够提升文本到代码生成的性能，这表明 CodeExecutor

有益于其他代码智能任务 .

7 结论

我们提出了一种基于变异的数据增强方法，用于创建大规模且真实的 Python代码执行数据集和任务，这对当前模型（如 Codex）构成了重大挑战。我们开发了 CodeExecutor，这是一种将代码执行作为预训练目标并采用课程学习策略的 Transformer模型。
CodeExecutor 不仅在代码执行方面优于现有模型，还展现出在代码到代码搜索和文本到代码生成等下游任务中的通用性。我们的工作为代码执行及其他代码智能任务提供了一种新颖且有效的解决方案。

局限性

CodeExecutor 存在若干局限性，例如仅应用于 Python、生成结果缺乏忠实性，以及轨迹生成的最大长度限制，这些问题为未来工作指明了有趣的方向。

编程语言 我们当前模型的一项局限性是目前仅应用于 Python，这限制了其在执行其他编程语言编写的程序时的使用范围和有效性。这凸显了未来工作扩展模型对其他语言适用性的必要性。

忠实性在处理困难示例时，结果可能不够忠实，例如包含复杂逻辑、长循环或多个分支的示例。比如，我们发现，在两个都包含赋值 *“alpha =list('abcdefg')”*的复杂程序中，我们的模型在一个程序中正确预测了*“alpha”* 的值，却在另一个程序中预测错误。忠实性不足的问题仍需进一步研究，以推进代码执行研究。

生成窗口大小 我们将生成的执行轨迹长度限制为 1024 个 token。对于执行轨迹较长的程序，尤其是包含循环的程序，这可能会构成限制。提升 Transformer模型处理更长序列的能力（Tay 等人，2021年，2022）可能有助于代码执行任务。

伦理声明

这项工作遵循伦理原则。本文介绍的数据集仅使用了公开可用的数据。人工评估中的标注由论文的两位作者完成，因此不存在相关顾虑，例如报酬方面的顾虑。因此，这项研究不存在潜在风险。

参考文献

Aida Amini、Saadia Gabriel、Shanchuan Lin、Rik Koncel-Kedziorski、Yejin Choi 和 Hannaneh Hajishirzi。2019年。Mathqa：通过基于操作的形式体系实现可解释的数学应用题求解。载于计算语言学协会北美分会 *2019* 年年会论文集：人类语言技术（*NAACL-HLT 2019*），美国明尼苏达州明尼阿波利斯，*2019*年*6*月*2*日至*7*日，第 *1* 卷（长篇和短篇论文），第2357–2367 页。计算语言学协会。

Jacob Austin、Augustus Odena、Maxwell I. Nye、Maarten Bosma、Henryk Michalewski、David Do- han、Ellen Jiang、Carrie J. Cai、Michael Terry、Quoc V. Le 和 Charles Sutton。2021年。使用大语言模型进行程序合成。*CoRR*，abs/2108.07732。

Yoshua Bengio、Jérôme Louradour、Ronan Collobert 和 Jason Weston。2009年。课程学习。收录于第*26*届国际机器学习年会论文集，*ICML 2009*，*2009*年*6*月*14*日至*18*日，加拿大魁北克省蒙特利尔，第382卷，*ACM*国际会议论文集系列，第41–48页。ACM。

David Bieber、Charles Sutton、Hugo Larochelle 和 Daniel Tarlow。2020 年。使用指令指针注意力图神经网络学习执行程序。收录于《神经信息处理系统进展》第*33*卷：神经信息处理系统年会 *2020*，*NeurIPS 2020*，*2020*年*12*月*6*日至*12*日，虚拟会议。

CaseyCasalnuovo、Earl T.Barr、Santanu KumarDash、Prem Devanbu 和 Emily Morgan。2020年。双通道约束理论。载于*ICSE-NIER 2020*：第 *42* 届软件工程国际会议：新想法与新兴成果，韩国首尔，*2020* 年 *6* 月 *27* 日至 *7* 月 *19* 日，第 25–28 页。ACM。

Saikat Chakraborty、ToufiqueAhmed、Yangruibo Ding、Premkumar T. Devanbu 和 Baishakhi Ray。2022年。Natgen：通过“自然化”源代码进行生成式预训练。收录于第*30*届 *ACM* 欧洲软件工程会议与软件工程基础研讨会联合会议论文集，*ESEC/FSE 2022*，*2022*年*11*月*14*日至*18*日，新加坡，第18–30页。ACM。

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 Win- ter、Philippe Tillet、Felipe Petroski Such、Dave Cummings、Matthias Plappert、Fotios Chantzis、Eliza- beth 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、JoshuaAchiam、Vedant Misra、Evan Morikawa、Alec Radford、Matthew Knight、Miles Brundage、Mira Murati、Katie Mayer、Peter Welin- der、Bob McGrew、Dario Amodei、Sam McCandlish、Ilya Sutskever 和 Wojciech Zaremba。2021年。评估在代码上训练的大语言模型。*CoRR*，abs/2107.03374。

Karl Cobbe、VineetKosaraju、MohammadBavarian、J a- cob Hilton、Reiichiro Nakano、Christopher Hesse 和 John Schulman。2021年。训练验证器解决数学文字题。*CoRR*，abs/2110.14168。

Mostafa Dehghani、Stephan Gouws、Oriol Vinyals、Jakob Uszkoreit 和 Lukasz Kaiser。2019年。通用 Transformer。载于第七届国际学习表征会议（*ICLR 2019*），美国路易斯安那州新奥尔良，*2019*年*5*月*6*日至*9*日。OpenReview.net。

Anna Derezi´nska 和 Konrad Hałas。2014年。用于 Python 程序变异测试的算子。研究报告。

Jacob Devlin、Ming-Wei Chang、Kenton Lee 和 Kristina Toutanova。2019年。BERT：用于语言理解的深层双向 Transformer 预训练。载于*2019*年计算语言学协会北美分会会议论文集：人类语言技术，*NAACL-HLT 2019*，美国明尼苏达州明尼阿波利斯，*2019*年*6*月*2*日至*7*日，第*1*卷（长篇和短篇论文），第4171–4186页。计算语言学协会。

Li Dong、Nan Yang、Wenhui Wang、Furu Wei、Xi- aodong Liu、Yu Wang、Jianfeng Gao、Ming Zhou 和 Hsiao-Wuen Hon。2019年。面向自然语言理解与生成的统一语言模型预训练。载于《神经信息处理系统进展》第 *32* 卷：*2019*年神经信息处理系统年度会议，*NeurIPS 2019*，*2019*年*12*月*8*日至*14*日，加拿大不列颠哥伦比亚省温哥华，第 13042–13054 页。

Zhangyin Feng、Daya Guo、Duyu Tang、Nan Duan、Xi- aocheng Feng、Ming Gong、Linjun Shou、Bing Qin、Ting Liu、Daxin Jiang 和 Ming Zhou。2020年。Code- bert：一种用于编程语言和自然语言的预训练模型。载于计算语言学协会研究成果：*EMNLP 2020*，线上活动，*2020*年*11*月*16*日至*20*日，第EMNLP 2020卷，载于*ACL*研究成果，第1536–1547页。计算语言学协会。

Alex Graves、Greg Wayne 和 Ivo Danihelka。2014年。神经图灵机。*CoRR*，abs/1410.5401。

Alex Graves、Greg Wayne、Malcolm Reynolds、Tim Harley、Ivo Danihelka、Agnieszka Grabska- Barwinska、Sergio Gomez Colmenarejo、Edward Grefenstette、Tiago Ramalho、John P. Agapiou、Adrià Puigdomènech Badia、Karl Moritz Hermann、Yori Zwols、Georg Ostrovski、Adam Cain、Helen King、Christopher Summerfield、Phil Blunsom、Ko- ray Kavukcuoglu 和 Demis Hassabis。2016年。使用具有动态外部存储器的神经网络进行混合计算。《自然》，538(7626):471–476。

Daya Guo、Shuai Lu、Nan Duan、Yanlin Wang、Ming Zhou 和 JianYin。2022年。UniXcoder：用于代码表示的统一跨模态预训练。载于第*60*届计算语言学协会年会论文集（第*1*卷：长篇论文），*ACL 2022*，爱尔兰都柏林，*2022*年*5*月*22*日至*27*日，第7212–7225页。计算语言学协会。

Daya Guo、Shuo Ren、Shuai Lu、Zhangyin Feng、Duyu Tang、Shujie Liu、Long Zhou、Nan Duan、Alexey Svyatkovskiy、Shengyu Fu、Michele Tufano、Shao Kun Deng、Colin B. Clement、Dawn Drain、Neel Sundaresan、JianYin、Daxin Jiang 和 Ming Zhou。2021年。GraphCodeBERT：结合数据流的代码表示预训练。载于第*9*届国际学习表征会议，*ICLR 2021*，虚拟活动，奥地利，*2021*年*5*月*3*日至*7*日。OpenReview.net。

Richard G.Hamlet. 1977年。借助编译器测试程序。*IEEETrans. SoftwareEng.*，3(4):279–290。

Dan Hendrycks、Collin Burns、Saurav Kadavath、Akul Arora、Steven Basart、Eric Tang、Dawn Song 和 Jacob Steinhardt。2021年。使用 MATH数据集 衡量数学问题求解能力。载于 *NeurIPS*数据集与基准论文集*1*，*NeurIPS 2021*数据集与基准，*2021*年*12*月，虚拟。

Tom Henighan、Jared Kaplan、Mor Katz、Mark Chen、Christopher Hesse、Jacob Jackson、Heewoo Jun、Tom B. Brown、Prafulla Dhariwal、Scott Gray、Chris Hallacy、Benjamin Mann、Alec Radford、Aditya Ramesh、Nick Ryder、Daniel M.Ziegler、John Schul-man、Dario Amodei 和 Sam McCandlish。2020年。自回归生成建模的缩放定律。*CoRR*，abs/2010.14701。

Abram Hindle、Earl T.Barr、Zhendong Su、Mark Gabel 和 Premkumar T. Devanbu。2012年。关于软件的自然性。载于第*34*届国际软件工程会议（*ICSE 2012*），*2012*年*6*月*2*日至*9*日，瑞士苏黎世，第837–847页。IEEE计算机学会。

Yue Jia 和 Mark Harman。2011年。变异测试的发展分析与综述。*IEEE* 软件工程汇刊，37(5):649–678。

Lukasz Kaiser 和 Ilya Sutskever。2016年。神经图形处理器学习算法。收录于第四届国际学习表征会议，*ICLR 2016*，波多黎各圣胡安，*2016*年*5*月*2*日至*4*日，会议论文集。

Aditya Kanade、Petros Maniatis、Gogul Balakrishnan 和 Kensen Shi。2020年。学习和评估源代码的上下文嵌入。收录于第*37*届国际机器学习会议论文集，*ICML 2020*，*2020*年*7*月*13*日至*18*日，线上会议，第119卷机器学习研究论文集，第5110–5121页。PMLR。

Karol Kurach、Marcin Andrychowicz 和 Ilya Sutskever。2016年。神经随机存取机。收录于第四届国际学习表征会议，*ICLR 2016*，波多黎各圣胡安，*2016*年*5*月*2*日至*4*日，会议论文集。

Wang Ling、Dani Yogatama、Chris Dyer 和 Phil Blunsom。2017。通过理由生成进行程序归纳：学习解决并解释代数文字题。载于 第*55*届计算语言学协会年会论文集，*ACL 2017*，温哥华，加拿大，*7*月*30*日至*8*月*4*日，第*1*卷：长篇论文，第158–167页。计算语言学协会。

Kevin Lu、Aditya Grover、Pieter Abbeel 和 Igor Mor- datch。2022年。将预训练的 Transformer 冻结为通用计算引擎。载于第三十六届*AAAI*人工智能会议（*AAAI 2022*）、第三十四届人工智能创新应用会议（*IAAI 2022*）、第十二届人工智能教育进展研讨会（*EAAI 2022*）虚拟活动，*2022*年*2*月*22*日至*3*月*1*日，第7628–7636页。AAAI出版社。

Maxwell I. Nye、Anders Johan Andreassen、Guy Gur- Ari、Henryk Michalewski、Jacob Austin、David Bieber、David Dohan、Aitor Lewkowycz、Maarten Bosma、David Luan、Charles Sutton 和 Augus- tus Odena。2021年。展示你的工作：使用语言模型进行中间计算的草稿区。*CoRR*，abs/2112.00114。

Ruchir Puri、David S. Kung、Geert Janssen、Wei Zhang、Giacomo Domeniconi、Vladimir Zolotov、Ju- lian Dolby、Jie Chen、Mihir R. Choudhury、Lindsey Decker、Veronika Thost、Luca Buratti、Saurabh Pu- jar、Shyam Ramji、Ulrich Finkler、Susan Malaika 和 Frederick Reiss。2021年。CodeNet：用于学习多种编码任务的大规模人工智能代码数据集。收录于神经信息处理系统数据集与基准专题论文集第*1*卷，*NeurIPS 2021*数据集与基准，*2021*年*12*月，虚拟。

AlecRadford、Karthik Narasimhan、TimSalimans、Ilya Sutskever 等。2018年。通过生成式预训练提升语言理解能力。

Alec Radford、Jeffrey Wu、Rewon Child、David Luan、Dario Amodei、Ilya Sutskever 等。2019年。语言模型是无监督多任务学习器。*OpenAI* 博客，1(8):9。Colin Raffel、Noam Shazeer、AdamRoberts、Katherine Lee、Sharan Narang、Michael Matena、Yanqi Zhou、Wei Li 和 Peter J. Liu。2020年。使用统一的文本到文本转换器探索迁移学习的极限。机器学习研究期刊，21:140:1–140:67。

Scott E. Reed 和 Nando de Freitas。2016年。神经程序员-解释器。载于第 *4* 届国际学习表征会议（*ICLR 2016*），*2016*年*5*月*2*日至*4*日，波多黎各圣胡安，会议论文集。

David Saxton、Edward Grefenstette、Felix Hill 和 Pushmeet Kohli。2019年。分析神经模型的数学推理能力。载于第 *7* 届国际学习表征会议（*ICLR 2019*），*2019*年*5*月*6*日至*9*日，美国路易斯安那州新奥尔良。OpenReview.net。

Alexey Svyatkovskiy、Shao Kun Deng、Shengyu Fu 和 Neel Sundaresan。2020年。IntelliCode Compose：使用 Transformer 进行代码生成。载于 *ESEC/FSE '20*：第 *28* 届 *ACM* 欧洲软件工程会议与软件工程基础研讨会联合会议，虚拟活动，美国，*2020*年*11*月*8*日至*13*日，第 1433–1443 页。ACM。

Yi Tay、Mostafa Dehghani、Samira Abnar、Yikang Shen、Dara Bahri、Philip Pham、Jinfeng Rao、Liu Yang、Sebastian Ruder 和 Donald Metzler。2021年。长程竞技场：高效转换器的基准测试。载于第九届国际学习表征会议，*ICLR 2021*，虚拟活动，奥地利，*2021*年*5*月*3*日至*7*日。OpenReview.net。

Yi Tay、Mostafa Dehghani、Dara Bahri 和 Donald Metzler。2022年。高效转换器：一项综述。《*ACM*计算调查》，55(6)：1–28。

Ashish Vaswani、Noam Shazeer、Niki Parmar、Jakob Uszkoreit、Llion Jones、Aidan N Gomez、Łukasz Kaiser 和 Illia Polosukhin。2017年。注意力就是你所需要的一切。神经信息处理系统进展，30。

Petar Velickovic、Lars Buesing、Matthew C. Overlan、Razvan Pascanu、Oriol Vinyals 和 Charles Blun- dell。2020a。指针图网络。收录于《神经信息处理系统进展》第 *33* 卷：神经信息处理系统 *2020* 年年会，*NeurIPS 2020*，*2020*年*12*月*6*日至*12*日，虚拟。

Petar Velickovic、Rex Ying、Matilde Padovano、Raia Hadsell 和 Charles Blundell。2020b。图算法的神经执行。收录于第八届国际学习表征会议，*ICLR 2020*，亚的斯亚贝巴，埃塞俄比亚，*2020*年*4*月*26*日至*30*日。OpenReview.net。

Xin Wang、Yasheng Wang、Fei Mi、Pingyi Zhou、Yao Wan、Xiao Liu、Li Li、Hao Wu、Jin Liu 和 Xin Jiang。2021a。Syncobert：语法引导的多模态对比式代码表示预训练。*arXiv*预印本 *arXiv:2108.04556*。

Yu Wang、Ke Wang、Fengjuan Gao 和 Linzhang Wang。2020年。利用图区间神经网络学习语义程序嵌入。*ACM* 程序语言会议论文集，4（OOPSLA）：137:1–137:27。

Yue Wang、Weishi Wang、Shafiq R. Joty 和 Steven C. H. Hoi。2021b。CodeT5：面向标识符的统一预训练编码器-解码器模型，用于代码理解与生成。载于*2021*年自然语言处理实证方法会议论文集（*EMNLP 2021*），虚拟活动／多米尼加共和国蓬塔卡纳，*2021*年*11*月*7*日至*11*日，第8696–8708页。计算语言学协会。

Yujun Yan、Kevin Swersky、Danai Koutra、Parthasarathy Ranganathan 和 Milad Hashemi。2020年。神经执行引擎：学习执行子程序。载于神经信息处理系统进展 *33*：神经信息处理系统年度会议 *2020*，*NeurIPS 2020*，*2020*年*12*月*6*日至*12*日，虚拟活动。

Wojciech Zaremba 和 Ilya Sutskever。2014年。学习执行。*CoRR*，abs/1410.4615。

Hattie Zhou、Azade Nova、Hugo Larochelle、Aaron C. Courville、Behnam Neyshabur 和 Hanie Sedghi。2022。通过上下文学习教授算法推理。*CoRR*，abs/2211.09066。

数据集详情

为了获取可执行程序，我们基于CodeNet中针对竞赛编程问题的提交构建了Python代码执行数据集（Puri等人，2021年）。这些具有现实世界复杂性的人工编写程序源自在线评测网站AIZU [6] 和AtCoder [7]。CodeNet包含24万份Python提交，旨在解决800个不同的编程问题。每次提交都是一个从标准输入读取数据并向标准输出写入数据的单文件Python程序。每个编程问题至少提供一个样例输入，最多提供四个样例输入。由于执行程序依赖于输入，我们将从输入流读取数据的语句替换为把输入值赋给变量的赋值语句。我们在沙箱环境中运行每次提交，以获取该程序的执行轨迹。程序的执行时间限制为一秒，执行轨迹限制为1024行，并将被筛选如果超出限制，则将其剔除。我们还会捕获程序中引发的 Python 异常，移除在解析或执行过程中产生运行时错误的程序。由此得到一个包含 38.7 万个可执行程序的数据集，每个程序都配有一条执行轨迹。

为了构建大规模的可执行程序数据集，我们提出了一种基于变异的数据增强方法。我们首先介绍一组变异算子，如表 1 所示。其中大多数对应于强类型通用语言中使用的精选算子，并被应用于 Python 语言。我们还纳入了针对 Python 特性的算子，例如切片索引移除（SIR）和反向迭代循环（RIL）。接着，我们利用 tree-sitter[8] 将程序转换为抽象语法树，然后提取其节点类型信息，以获得所有可变字面量、算子和语句的候选列表。对于每个可变候选项，我们以 50% 的概率应用相应的变异算子。具体而言，我们将数值字面量 x 替换为从均值为x 、标准差为 100 的高斯分布中随机抽取的数值。我们会为字符串随机添加一两个字符，或缩短字符串。处理每个循环时，我们会从三个与循环相关的算子中随机选取一个，或者保持原样。所有算子都可以在执行变异后的程序之前应用；执行过程中会检测并剔除可能存在错误的变异体。对每个程序进行 20 次变异后，我们得到 320 万个去重后的程序，每个程序都配有一条执行轨迹。

我们使用 CodeNet Mutants（CodeNetMut）构建预训练数据集。为防止数据泄漏，同一问题的所有提交都划分到同一个子集中。我们使用 710 个问题的提交及其变异体构建预训练数据集。由于变异大大增强了多样性，这些程序蕴含丰富的语义并包含复杂的操作。其他提交（未经变异）用于构建验证集和测试集。这些由人工编写的程序确保了评估数据的质量。

B 模型配置

我们的模型基于 12 层 Transformer 构建，具有 768 维隐藏状态和 12 个注意力头。我们在词汇表中加入了 210 个额外的特殊标记，用于表示 200 个行号、 3 个预训练数据集名称和执行轨迹

[6] https://onlinejudge.u-aizu.ac.jp/ [7] https://atcoder.jp/

[8] https://tree-sitter.github.io/tree-sitter/

模型	阶段1 (S1)	阶段2 (S2)	阶段3 (S3)
CEL	SingleLine	教程	CodeNetMut
CodeExecutor	SingleLine	SingleLine（3M），教程	SingleLine (3M)、教程、CodeNetMut

表7：CEL 和 CodeExecutor 在三阶段预训练中使用的数据集。“SingleLine (3M)”表示 SingleLine 中对 CodeExecutor 而言最难生成的300万条实例。

Code:

```
from string import ascii_lowercase
def main():
    s = ['a']
    K = 25
    for i, c in enumerate(s):
        num = ord(c) - ord("a")
        if num > 0 and K - (26 - num) >= 0:
            s[i] = "a"
            K -= (26 - num)
    num = ord(s[-1]) - ord("a")
    last = (num + K) % 26
    s[-1] = ascii_lowercase[last]
    print("".join(s))
if __name__ == "__main__" :
    main()
```

Prediction:

```
<line> 1 <state> ascii_lowercase : abcdefghijklmnopqrstuvwxyz
<line> 2 <state> ascii_lowercase : abcdefghijklmnopqrstuvwxyz ; main : <function>
<line> 14 <state> ascii_lowercase : abcdefghijklmnopqrstuvwxyz ; main : <function>
<line> 15 <state>
<line> 2 <state>
<line> 3 <state> s : [a]
<line> 4 <state> s : [a] ; K : 25
<line> 5 <state> s : [a] ; K : 25 ; i : 0 ; c : a
<line> 6 <state> s : [a] ; K : 25 ; i : 0 ; c : a ; num : 0
<line> 7 <state> s : [a] ; K : 25 ; i : 0 ; c : a ; num : 0
<line> 5 <state> s : [a] ; K : 25 ; i : 0 ; c : a ; num : 0
<line> 10 <state> s : [a] ; K : 25 ; i : 0 ; c : a ; num : 0
<line> 11 <state> s : [a] ; K : 25 ; i : 0 ; c : a ; num : 0 ; last : 25
<line> 12 <state> s : [z] ; K : 25 ; i : 0 ; c : a ; num : 0 ; last : 25
<output> z
<line> 13 <state> s : [z] ; K : 25 ; i : 0 ; c : a ; num : 0 ; last : 25
<line> 13 <state> ascii_lowercase : abcdefghijklmnopqrstuvwxyz ; main : <function>
```

图 4：来自 CodeNetMut 测试集的一个示例，涵盖了 Python 编程知识的所有类别。CodeExecutor 给出了正确的预测。

结构如§4.2所述。在预训练期间，我们将输入序列的最大长度和批量大小分别设为 1024 和 256。我们使用 Adam 优化器，以 4e-4 的学习率更新模型参数。首先，我们使用 SingleLine 数据集，以代码执行目标对模型进行 500k 步预训练。随后，我们保留 SingleLine 中对模型而言最难生成的 300万条实例，并将 Tutorial 数据加入语料库，继续预训练 300k 步。我们将 CodeNetMut 加入语料库，并进一步预训练 300k 步。我们在一个由 16 个配备 32GB 内存的 NVIDIA Tesla V100 组成的集群上对模型进行预训练，总训练时间约为一个月。在推理时，我们将 beam search 的数量设为 10。

C 三阶段预训练

表 7中，我们分别列出了 CodeExecutor-Limited (CEL) 和 CodeExecutor 在三阶段预训练中使用的数据集。

CEL 预训练的第一阶段使用 SingleLine 数据集，得到模型 CEL-S1。在第二阶段，CEL 以 CEL-S1 为初始化模型，并使用 Tutorial 数据集进行预训练，得到模型 CEL-S2。在第三阶段，CEL 以 CEL-S2 为初始化模型，并使用 CodeNetMut 数据集进行预训练，得到模型 CEL-S3。

另一方面，CodeExecutor 首先使用 SingleLine 数据集进行预训练，然后根据模型的损失，选出 300 万条最具挑战性的 SingleLine 数据，用于后续训练阶段。在第二阶段，CodeExecutor 使用 300 万条困难的 SingleLine 数据以及教程数据集进行预训练。在第三阶段，CodeExecutor 使用 300 万条困难的 SingleLine 数据、完整的教程数据集和 CodeNetMut 数据集进行预训练。

D 定性示例

此处展示了更多示例。

图 4展示了一个示例，涵盖了表 4中 Python 编程知识的所有类别。CodeExecutor 生成了与真实值相同的执行轨迹。

图5是一个使用小数进行除法计算的示例。CodeExecutor 能够得出正确的前十五位数字，但在剩余两位数字上出现了错误。

Code:

```
x = 1.2379400392853809e-46
x /= 5
```

Ground truth:

```
x : 2.475880078570762e-47
```

Prediction:

```
x : 2.475880078570718e-47
```

图5：一个小数除法计算的示例，其中CodeExecutor 正确生成了前十五位数字，错误之处以<u>下划线</u>标出。