探索代码生成任务的数据增强

Pinzhen Chen[*]信息学院
爱丁堡大学
pinzhen.chen@ed.ac.uk

Gerasimos Lampouras诺亚方舟实验室 华为
gerasimos.lampouras@huawei.com

arXiv:2302.03499v1 [cs.CL] 5 Feb 2023

摘要

自然语言处理领域的进步，例如基于预训练语言模型的迁移学习，也影响了编程语言任务的模型训练方式。以往研究主要探索了代码预训练，并通过多模态和多任务扩展了这一方向，但下游任务的数据规模仍然有限。聚焦于下游任务的数据利用，我们提出并调整了多种增强方法，使代码翻译和摘要生成分别提升高达 6.9% 和 7.5%。进一步分析表明，这些方法彼此正交，并有助于改善输出代码风格和数值一致性。我们还讨论了测试数据缺陷。

1 引言

近年来，预训练语言模型（PLMs）迅速发展，使知识能够从通用文本迁移到特定的下游任务中（Devlin 等人，2019年；Liu 等人，2019年）。遵循相同范式，先在代码和文本数据上（继续）训练预训练语言模型，再针对特定任务对其进行微调，预训练语言模型也被应用于编程语言领域（Kanade 等人，2020年；Feng 等人，2020年）。通过将序列化语法树和数据流等代码专属模态纳入输入，预训练语言模型通常能够适配编程语言（Guo 等人，2021年，2022年；Tipirneni 等人，2022年）。这类工作在多种任务上的表现都优于基于规则的工具，例如在 CodeXGLUE 基准测试中（Lu 等人，2021年）。

尽管有大量原始代码可用于预训练，但满足下游需求的代码数据规模仍然不大。这是因为与文本不同，没有编程知识的人很难整理代码数据集。例如，CodeXGLUE 中的代码翻译数据规模为 1 万条，比自然语言文本数据集小了好几个数量级，而后者通常包含数百万个样本（Kocmi 等人，2022年）。

因此，我们希望在代码预训练语言模型的微调阶段，利用回译、单语、多语言和数字增强等自动数据增强（DA）方法来丰富数据。我们在代码翻译和摘要生成任务上进行了广泛实验：前者将一种编程语言转换为另一种编程语言，后者则根据代码数据块生成文本描述。即使资源有限，与基线相比，我们仍能将翻译任务的性能提升 6.9%，将摘要生成任务的性能提升 7.5%。通过人工检查和额外的评估指标，我们证明了这些方法能够带来代码领域特有的理想增强效果，即更优的输出代码风格和数字正确性。

2 方法

2.1 数据合成

回译（BT，Sennrich 等人，2016年）是一种源自机器翻译的数据增强技术，它利用辅助模型从单语资源中构造伪平行数据。这种方法可以直接应用于代码翻译。形式化地说，为了训练一个将编程语言 $f()$ 转换为 PL_x 的模型 PL_y，我们首先使用相同的平行数据训练逆向模型 $g(PL_y) \rightarrow PL_x$。有了逆向模型 $g()$后，PL_y 中的额外单语数据会被翻译成 PL'_x，从而形成伪平行数据对 PL'_x-PL_y，并可用于训练 $f()$。

对于代码摘要生成而言，回译并不适用，因为几乎不存在未与代码对齐的“单语”自然语言摘要（NL）。因此，我们提出将最初与单一编程语言关联的摘要用作其他编程语言的枢轴。在

[*]这项工作是在华为诺亚方舟实验室实习期间完成的。我们的代码将在以下网址公开：
https://github.com/huawei-noah/noah-research/tree/master/NLP/DA4CodeGeneration

对源端代码以多种编程语言（$PL_1 \rightarrow NL$、...、$PL_n \rightarrow NL$）提供的代码到文本数据进行反向处理后，我们训练了一个多语言文本到代码生成器。给定自然语言摘要和目标语言标签（$NL + tag_{\{1,...n\}} \rightarrow \{PL_1, \ldots, PL_n\}$），该生成器会输出指定编程语言的代码。无论原始$PL \rightarrow NL$对齐方式如何，只要输入摘要，该生成器便可迭代生成不同PL中的代码。尽管这些合成数据的质量较低，但可以扩充摘要生成的训练数据。

2.2 多语性的利用

Currey 等人 (2017年) 提出，在目标语言中加入单语数据，作为额外的自编码（AE）目标，有助于提升在有限数据上训练的翻译模型。我们将这一目标迁移到代码翻译中，通过混合 $PL_x \rightarrow PL_y$和 $PL_y \rightarrow PL_y$数据来实现。鉴于编程语言之间具有很高的相似性，即数字、语法标记、保留关键字等存在重叠，这实际上构建了一个多语言编码器，从而实现知识迁移。此过程将解码器端限制为单一编程语言 PL_y，以免增加复杂性。

在代码摘要生成中，由于目标自然语言与输入编程语言存在根本差异，自编码目标可能并无用处。相反，我们训练了一个“多语言”代码摘要模型 $\{PL_1, \ldots, PL_n\} \rightarrow NL$，该模型可接收任意编程语言并生成自然语言摘要。这样的多对一模型也有助于实现编码器知识共享，并使解码器接触更多自然语言摘要。

2.3 数值感知

引用变量及其值是编程语言中独特的组成部分；为了加深对这些值的理解，先前关于预训练的研究建议关注适当的模态，例如数据流（Guo 等人，2021年）。对于代码翻译而言，可能无需如此精细地处理值，因为只需将它们复制到目标端即可。然而，鉴于训练数据规模较小，任何翻译模型接触到的数值输入仍然有限。为提升模型鲁棒性，我们通过创建新的实例来扩充数据：对于所有包含数字的代码词元，随机将其中的每一位数字替换为另一位数字，并在源端和目标端保持一致。我们不区分

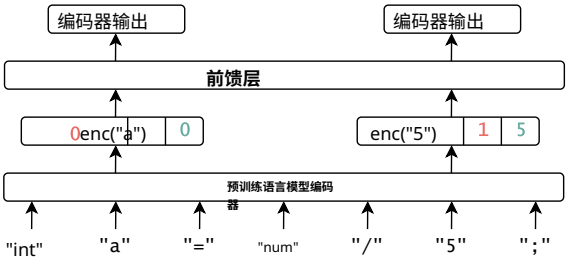

图 1：使用预训练语言模型编码器进行数值编码，示例说明“a”和“5”的编码方式有所不同。

纯数值型词元与包含数字的词元。例如，变量“num1”在增强后的代码对中可能变成“num4”。这种方法能够确保数字替换后的合成代码符合语法且可编译。

除了数值增强外，我们还提出将输入中的数字直接作为数学值纳入编码器输出，以补充其字符串嵌入表示。如图 1 所示，我们在原始编码器输出中追加两个维度。具体而言，一个维度（左侧红色）是二进制值（0/1），用于表示相应输入是否为数字；另一个维度（右侧绿色）继承输入的值，若输入不是数值，则为 0。扩展后的嵌入可通过前馈层缩减至原始大小；这种改动无需修改预训练编码器。

3 实验

3.1 任务、数据集和评估

我们使用代码任务套件 CodeXGLUE（Lu 等人，2021年）对我们的方法进行基准测试。该翻译任务使用最初以 Java 编写、后迁移至 C# 的代码，因此相应的 C#-Java 代码片段被视为平行语料。训练、验证和测试集的规模分别为 10K、0.5K 和 1K。对于回译，我们将其他 CodeXGLUE 任务中的 377 K 行单语 Java 代码（尽管其属于领域外数据）翻译成 C#。为确保目标端由真实数据构成，我们只实验了 C#→Java 方向，因为基准测试中没有其他 C# 代码可用于回译。

摘要任务采用 CodeSearch- Net（Husain 等人，2019年），涵盖六种语言：Ruby、JavaScript、Go、Python、Java 和 PHP。训练集大小从 25K 到 250K 不等，总计 908K；验证集和测试集的规模在 1K 到 15K 之间。我们通过反转数据集进行多语回译，

	BLEU 精确匹配	CodeBLEU†
CodeBERT		
论文	72.14 58.0	-
复现	72.92 57.4 78.93 (72.92/ 73.61/ 87.08/ 82.10)	
BT	77.34 61.4 83.36 (77.34 / 78.11 /**90.34** / 87.64)	
+ AE	**77.60 61.8 83.47**(**77.60** / **78.30** / 90.02 / **87.96**)	
GraphCodeBERT		
论文	72.64 58.8	-
复现	72.66 58.9 78.55 (72.66/ 73.35/ 87.44/ 80.74)	
BT	75.15 60.7 82.13 (75.15/ 75.86/ 90.06/ 87.46)	
+ AE	**76.15 62.5 82.88**(**76.15** / **76.87** / **90.54** / **87.95**)	

†平均值（n 元语法 / 加权 n 元语法 / 句法 / 数据流）

表 1：C#→Java 翻译测试结果。

因此没有引入外部数据；这产生了五倍的回译数据，共 4.5M（908K×5）。所有编程语言的原始数据和合成数据总量相同。此外，为了比较神经回译与手写规则转换的质量，我们使用 jsbuilder 从 Python 摘要数据创建了 80K 个 JavaScript-摘要配对。

我们以 BLEU-4（Papineni 等人，2002年）、精确行匹配（EM，以百分比表示）和 CodeBLEU 报告代码翻译结果。CodeBLEU 是四项准确率的加权和：n 元语法、加权 n 元语法、句法和数据流（Ren 等人，2020年）。代码摘要性能则采用自然语言文本中事实上的标准指标 BLEU-4 进行衡量。

3.2 系统

对于所有任务，我们都采用 CodeXGLUE 基线，即使用带有 Transformer 解码器的 CodeBERT，作为基础模型和逆向模型（用于数据合成）。我们先在增强后的数据上继续训练预训练语言模型，再在原始数据上进行微调；数值型增强除外，我们会将合成数据与训练集混合。单语和多语摘要生成实验采用相同的配置。对于使用 CodeBERT 进行数值编码的任务，我们添加一个前馈层，使基线模型的深度与所提出的网络相同。

为提供更强基线下的结果，我们还测试了用于翻译的 GraphCodeBERT（Guo 等人，2021年）以及用于摘要生成的 UniXcoder（Guo 等人，2022年）。这有助于验证数据增强性能在不同预训练语言模型架构之间的稳定性。除批次大小外，我们采用相关预训练语言模型的超参数。模型和训练细节，以及预处理和评估脚本的链接，见附录 A。

	Ruby	JavaScript	Go	Python	Java	PHP	平均值
CodeBERT							
论文	12.16	14.90	18.07	19.06	17.65	25.16	17.83
单语	12.39	14.13	17.89	18.22	18.66	25.14	17.73
+ 规则翻译	-	15.35	-	-	-	-	-
+ BT	13.76	15.00	18.30	18.60	19.64	25.69	18.50
多语言	**14.93**	15.53	18.68	18.71	19.70	25.96	18.92
+规则翻译	14.58	15.65	18.77	18.95	**19.86**	25.98	18.97
+ BT	14.91	**15.81**	**18.88**	**18.97**	19.69	**26.10**	**19.06**
UniXcoder							
论文	14.87	15.85	19.07	19.13	20.31	26.54	19.30
单语	14.81	15.28	18.93	19.05	20.22	26.66	19.16
多语言	**15.15**	15.64	19.03	19.22	**20.45**	26.59	19.35
+ BT	14.94	**15.85**	**19.29**	**19.36**	20.43	**26.69**	**19.43**

表2：代码摘要生成的 BLEU 测试结果。

3.3 结果与讨论

我们首先在表 1中展示翻译结果：对于两个预训练语言模型，回译都以较大优势超过基线；在此基础上，自编码还能带来小幅提升。表 2 显示，整体而言，回译也持续有助于代码摘要。两个预训练语言模型都呈现出一个有趣的规律：与其他语言相比，回译对 Ruby 和 Java 的帮助较小。此外，训练单个多语言模型优于训练多个独立的单语言模型，这可能是因为编程语言之间存在迁移学习，且输出端的自然语言数据规模有所增加。

表3报告了翻译任务中数值增强和数值编码的结果。将数字交换数据加入训练后，表现优于基线；而我们提出的数值编码方法则逊于基线。考虑到神经网络权重比代码中遇到的变量值小几个数量级，我们研究了线性值缩放和对数值缩放。随着缩放幅度减小，结果数值逐渐上升；对数变换效果最佳，模型由此取得最高性能。

为了直接评估我们的数值感知增强，我们在表3中计算并附上输出词元准确率，并区分数值词元和非数值词元。我们可以看到，数值型方法有助于数字生成，同时不会损害非数字内容；数字正确性的提升通常与 BLEU 和精确匹配的提升保持一致。附录B.2中的补充可视化表明，即使输出极长且复杂，数据增强模型也能保持数值一致性。

	BLEU	EM	CodeBLEU	词元准确率	
				数值型	非数值型
CodeBERT +前馈神经网络	72.88	58.0	78.07（72.88 / 73.66 / 86.15 / 79.59）	74.50	86.72
+ 数值增强	**74.00**	**59.5**	**79.43** （**74.00 /74.72 /87.01 /82.00**）	**76.14**	**87.30**
数值编码	72.95	58.1	78.77（72.95 / 73.74 / 86.96 / 81.45）	73.74	86.84
+ 采用值缩放的数值增强					
$\times 10^2$	71.32	51.6	77.71 (71.32 / 72.25 / 86.22 / 81.05)	72.48	85.98
$\times 1$ (不进行缩放)	72.51	57.4	78.45（72.51 / 73.38 / 86.47 / 81.46）	72.92	86.49
$\times 10^{-2}$	73.48	**59.2**	79.41 (73.48 / 74.28 / 87.31 / 82.56)	74.11	87.11
$\times 10^{-4}$	74.01	58.9	79.73 (74.07 / 74.75 / 87.29 / 82.87)	74.93	**87.48**
$\log_{10}()$	**74.16**	59.1	**79.84** （**74.16 /74.91 /87.39 /82.90**）	**75.22**	87.32

表3：带数值增强和编码的 C#→Java 翻译测试结果。

	BLEU	EM	CodeBLEU	词元准确率	
				数值型	非数值型
CodeBERT 复现	72.92	57.4	78.93（72.92 / 73.61 / 87.08 / 82.10）	74.64	87.54
BT	77.34	**61.4**	83.36（77.34 / 78.11 / 90.34 / 87.64）	78.09	88.62
+数值增强 仅原始数据	**77.69**	61.0	**83.44** （**77.69 /78.33** / 90.19 / 87.56）	**78.54**	**88.69**
+ 数值增强 回译和原始数据	77.37	60.9	83.43 (77.37 / 78.07 /**90.36** / **87.94**)	77.16	88.55
回译 + 自动编码器	77.60	61.8	83.47 (77.60 / 78.30 / 90.02 /**87.96**)	77.16	88.64
+数值增强 仅使用原始数据	**77.96**	**62.0**	**83.63** (**77.96 /78.62 /90.15** / 87.82)	**78.01**	**88.79**

表 4：C# Java 翻译的测试结果，采用多种增强方法 技术。

最后，表4 考察了上述方法，即回译和数值增强，是否能正交发挥作用。结果显示，将数值增强应用于原始数据时效果更好，但应用于回译后的数据时则不然。这可能是因为回译本身的质量较差，因此数值增强会引入额外噪声。尽管如此，将回译和自动编码器与应用于原始数据的数值增强相结合，效果最佳。

4 分析

检查翻译测试的输出后，我们发现，经过数据增强的模型对目标 Java 语言的接触更多：它学会了 Java 编程规范，而不是遵循输入代码的风格。我们列举了聚焦于元素检索方法的测试实例，并在表 5 中展示了源代码、参考译文以及 CodeBERT 基线和经过回译增强的模型的输出。虽然在 Java 中可以通过引用位置直接检索元素，但我们观察到，基线模型倾向于模仿源代码 C# 的风格，而 DA 模型则紧密遵循 Java 编码规范，倾向于使用内置方法 `get()`，而不是通过索引直接访问属性。

需要指出的是，在翻译测试集里，少数代码对似乎存在差异，这可能导致对翻译性能的估计不准确。我们在附录中记录了几个此类缺陷的示例B.1，但将深入调查和完善留待未来工作。

5 相关工作

近期，自然语言处理与编程语言交叉领域的研究主要集中于预训练。Kanade 等人（2020年）训练 CuBERT，以获取用于代码理解任务的嵌入表示。Feng 等人（2020年）通过在双模态文本-代码数据上训练 RoBERTa，并采用替换词元检测（Clark 等人，2020年），开发了 CodeBERT。在 GraphCodeBERT 中，Guo 等人（2021年）加入了数据流边预测和数据-变量对齐。研究人员也将仅解码器模型扩展到了代码领域，例如 CodeGPT、Codex 和盘古-Coder（Lu 等人，2021年；Chen 等人，2021年；Christopoulou 等人，2022年）。此外，研究人员还提出了通用编码器-解码器代码预训练语言模型：PyMT5、CodeT5、PLBART、UniXcoder 和 StructCoder（Clement 等人，2020年；Wang 等人，2021年；Ahmad 等人，2021年；Guo 等人，2022；Tipirneni 等人，2022年）。我们使用的 UniXcoder 在共享编码器-解码器网络中采用注意力掩码来控制编码器-解码器行为。

与代码相关的特定任务通常数据集较小，因此数据增强有助于提升性能。 Roziere等人 (2020年) 进行了结合

// 测试 *#85***C# 源码** ... GetEscherRecord(intindex){return escherRecords**[index];}Java 参考实现**... getEscherRecord(int index){return escherRecords**.get(index);}基线** ... getEscherRecord(intindex) {return escherRecords**[index];}DA 模型** ... getEscherRecord(intindex) {return escherRecords**.get(index);**} // 测试 *#90***C# 源码** public virtual IQueryNode GetChild(){return GetChildren()**[0];}Java 参考实现**public QueryNode getChild() {return getChildren()**.get(0);}基线** public QueryNode getChild() {return getChildren() **== 0);}DA 模型** public QueryNode getChild() {return getChildren()**.get(0);**} // 测试 *#978***C# 源码** public virtual SrndQuery GetSubQuery(int qn) { return m_queries**[qn]; }Java 参考实现**public SrndQuery getSubQuery(int qn) {return queries**.get(qn);}基线** public SrndQuery getSubQuery(int qn) {return queries**[qn];}DA 模型** public SrndQuery getSubQuery(int qn) {return queries**.get(qn);** }

表 5：数据增强前后，元素检索方法的 C#-Java 输出翻译。

跨语言掩码建模和迭代回译，以构建无监督代码转译器。Ahmad 等人（2022年）先进行代码到文本的摘要生成，再进行文本到代码生成，以获得翻译数据。相比之下，我们通过反转摘要数据来训练文本到代码生成模型；我们的方法在流程和目标任务上均有所不同。此外，Yu 等人（2022年）为源代码转换制定了规则，而我们的研究则关注自动神经方法。最后，恶意软件混淆中的死代码插入和变量重命名技术（YouandYim，2010年），以及字符串操作（例如词元加噪、交换、删除）也可能有所帮助。不过，这些方法并非特定于某项任务，因此可能更适用于通用代码预训练阶段。

6 结论

我们将多种数据增强技术应用于编程语言翻译和摘要任务。我们的研究包括数据合成、通过多语性实现的知识共享，以及数值感知技术。在多种预训练代码语言模型上开展的实验中观察到了性能提升，我们的分析表明，这些方法有助于改善输出代码风格和数值正确性。

7 局限性

我们认为，主要局限在于评估，因为我们对代码和文本生成都依赖自动文本指标。理想情况下，应采用软件测试实践来评估代码，例如代码审查、编译、单元测试等。鉴于测试数据存在问题，评估的可靠性也进一步受到影响见第 4 节和附录 B.1所述，因此进一步开展人工分析应会很有价值。

我们也未涵盖所有潜在的代码生成任务，例如代码合成，即根据文本描述创建代码片段。在此任务中，源端所携带的信息远少于目标端。我们采用了回译风格的数据增强方法，但其性能并未显著超越最先进的预训练语言模型。受篇幅限制，我们在附录 C 中提供了一些初步观点。

致谢

我们感谢 Ignacio Iacobacci 对数值输入缩放提出的意见，也感谢审稿人提出的定性分析建议。我们还感谢 MindSpore 团队提供的技术支持。[1,2]

Pinzhen Chen 获英国研究与创新署支持，资助来自英国政府的地平线欧洲资助保障计划[，资助编号为10052546 – 高性能语言技术]。

参考文献

Wasi Ahmad、Saikat Chakraborty、Baishakhi Ray 和 Kai-Wei Chang。2021年。面向程序理解与生成的统一预训练。载于*2021*年北美计算语言学协会年会论文集：人类语言技术。

Wasi Uddin Ahmad、Saikat Chakraborty、Baishakhi Ray 和 Kai-WeiChang。2022年。通过摘要和生成进行反向翻译：编程语言的无监督翻译。*arXiv* 预印本，abs/2205.11116v1。

[1]https://www.mindspore.cn/en [2] https://github.com/mindspore-ai

Mark Chen、Jerry Tworek、Heewoo Jun、Qiming Yuan、Henrique Ponde、Jared Kaplan、Harrison Edwards、Yura Burda、Nicholas Joseph、Greg Brockman、Alex Ray、Raul Puri、Gretchen Krueger、Michael Petrov、Heidy Khlaaf、Girish Sastry、Pamela Mishkin、Brooke Chan、Scott Gray、……以及Wojciech Zaremba。2021年。评估基于代码训练的大语言模型。*arXiv* 预印本，abs/2107.03374v2。

Fenia Christopoulou, Gerasimos Lampouras, Milan Gritta, Guchun Zhang, Yinpeng Guo, Zhong-Yi Li, Qi Zhang, Meng Xiao, Bo Shen, Lin Li, Hao Yu, Li Yan, Pingyi Zhou, Xin Wang, Yu Ma, Ignacio Ia- cobacci, Yasheng Wang, Guangtai Liang, Jia Wei, ..., and Qun Liu. 2022年。盘古-Coder：利用函数级语言建模进行程序合成。*arXiv*预印本，abs/2207.11280v1。

Kevin Clark, Minh-Thang Luong, Quoc V. Le, and Christopher D. Manning. 2020年。Electra：将文本编码器预训练为判别器，而非生成器。发表于国际学习表征会议。

Colin Clement, Dawn Drain, Jonathan Timcheck, Alexey Svyatkovskiy, and Neel Sundaresan. 2020年。PyMT5：使用Transformer对自然语言和Python代码进行多模式翻译。发表于*2020*年自然语言处理实证方法会议论文集（*EMNLP*）。

Anna Currey、Antonio ValerioMiceli Barone 和 Ken- neth Heafield。2017年。复制单语数据可提升低资源神经机器翻译的效果。载于第二届机器翻译会议论文集。

Jacob Devlin、Ming-Wei Chang、Kenton Lee 和 Kristina Toutanova。2019年。BERT：用于语言理解的深度双向Transformer预训练。载于计算语言学协会北美分会*2019*年会议论文集：人类语言技术，第*1*卷（长篇论文和短篇论文）。

Zhangyin Feng、Daya Guo、Duyu Tang、Nan Duan、Xi- aocheng Feng、Ming Gong、Linjun Shou、Bing Qin、Ting Liu、DaxinJiang 和 MingZhou。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*年。

Daya Guo、Shuai Lu、Nan Duan、Yanlin Wang、Ming Zhou 和 Jian Yin。2022年。UniXcoder：用于代码表征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）。

Daya Guo、Shuo Ren、Shuai Lu、Zhangyin Feng、Duyu Tang、Shujie Liu、Long Zhou、Nan Duan、AlexeySvyatkovskiy、Shengyu Fu、Michele Tufano、Shao Kun Deng、Colin Clement、Dawn Drain、Neel Sundaresan、Jian Yin、Daxin Jiang 和 Ming Zhou。2021年。GraphCodeBERT：利用数据流预训练代码表示。载于国际学习表征会议。Hamel Husain、Hongqi Wu、Tiferet Gazit、Miltiadis Allamanis 和 Marc Brockschmidt。2019年。CodeSearchNet 挑战赛：评估语义代码搜索。arXiv 预印本，abs/1909.09436v3。Srinivasan Iyer、Ioannis Konstas、Alvin Cheung 和 Luke Zettlemoyer。2018年。在程序化上下文中将语言映射为代码。载于2018年自然语言处理实证方法会议论文集。Aditya Kanade、Petros Maniatis、Gogul Balakrishnan 和 Kensen Shi。2020年。学习并评估源代码的上下文嵌入。载于第37届国际机器学习会议论文集。Tom Kocmi、Rachel Bawden、Ondřej Bojar、Anton Dvorkovich、Christian Federmann、Mark Fishel、Thamme Gowda、Yvette Graham、Roman Grundkiewicz、Barry Haddow、Rebecca Knowles、Philipp Koehn、Christof Monz、Makoto Morishita、Masaaki Nagata、Toshiaki Nakazawa、Michal Novák、Martin Popel 和 Maja Popovi´c。2022年。2022年机器翻译会议（WMT22）论文集。载于第七届机器翻译会议（WMT）论文集。Yinhan Liu、Myle Ott、Naman Goyal、Jingfei Du、Mandar Joshi、Danqi Chen、Omer Levy、Mike Lewis、Luke Zettlemoyer 和 Veselin Stoyanov。2019年。RoBERTa：一种稳健优化的 BERT 预训练方法。arXiv 预印本，abs/1907.11692v1。Shuai Lu、Daya Guo、Shuo Ren、Junjie Huang、Alexey Svyatkovskiy、Ambrosio Blanco、Colin Clement、Dawn Drain、Daxin Jiang、Duyu Tang、Ge Li、Lidong Zhou、Linjun Shou、Long Zhou、Michele Tufano、Ming Gong、Ming Zhou、Nan Duan、Neel Sundaresan、…… 和 Shujie Liu。2021年。CodeXGLUE：用于代码理解与生成的机器学习基准数据集。载于第三十五届神经信息处理系统会议数据集与基准赛道。Kishore Papineni、Salim Roukos、Todd Ward 和 Wei-Jing Zhu。2002年。BLEU：一种机器翻译自动评估方法。载于计算语言学协会第40届年会论文集。Shuo Ren、Daya Guo、Shuai Lu、Long Zhou、Shujie Liu、Duyu Tang、M. Zhou、Ambrosio Blanco 和 Shuai Ma。2020年。CodeBLEU：一种代码合成自动评估方法。arXiv 预印本，abs/2009.10297v2。Baptiste Roziere、Marie-Anne Lachaux、Lowik Chanussot 和 Guillaume Lample。2020年。编程语言的无监督翻译。载于神经信息处理系统进展。

Rico Sennrich、Barry Haddow 和 Alexandra Birch。2016年。利用单语数据改进神经机器翻译模型。载于计算语言学协会第*54*届年会论文集（第*1*卷：长论文）。

Sindhu Tipirneni、Ming Zhu 和 Chandan K. Reddy。2022年。Structcoder：面向代码生成的结构感知型 Transformer。*arXiv* 预印本，abs/2206.05239v1。

Yue Wang、Weishi Wang、Shafiq Joty 和 Steven C.H. Hoi。2021年。CodeT5：面向代码理解与生成的标识符感知统一预训练编码器-解码器模型。载于*2021*年自然语言处理经验方法会议论文集。

Ilsun You 和 Kangbin Yim。2010年。恶意软件混淆技术：简要综述。载于 *2010*年国际宽带、无线计算、通信与应用会议。

Shiwen Yu、Ting Wang 和 Ji Wang。2022年。通过程序变换进行数据增强。《系统与软件杂志》，190:111304。

模型配置

我们的训练和模型配置总结如下，并见表6。我们尽可能保留相关预训练语言模型的默认配置，唯一的例外是针对使用 UniXcoder 进行代码摘要的学习率进行了网格搜索。我们还调整了批次大小，以充分利用我们的图形处理器。

随机初始化并连接到 CodeBERT 和 GraphCodeBERT 的 Transformer 解码器有 6 层、12 个注意力头、768 个隐藏维度，其他超参数均采用 PyTorch 默认值。在使用 CodeBERT 进行数值编码实验时，我们在 CodeBERT 的 768 维编码器输出后追加 2 个维度，然后通过一个线性层将其转换回 768 维。为确保比较公平，我们在基线上添加一个 768 维到 768 维的层，使其深度相同。

所有实验都设定了固定的运行预算。我们根据验证集 BLEU 保存最佳检查点。论文中的结果基于单次运行，但我们在不同架构的预训练语言模型上对实验进行了基准测试，以体现稳定性。

超参数	**值**
预训练语言模型检查点	CodeBERT：https://huggingface.co/microsoft/codebert-base GraphCodeBERT：https://huggingface.co/microsoft/graphcodebert-base UniXcoder：https://huggingface.co/microsoft/unixcoder-base CodeGPT：https://huggingface.co/microsoft/CodeGPT-small-java-adaptedGPT2 StructCoder：https://github.com/reddy-lab-code-research/structcoder
可训练参数	CodeBERT：172.5M +数值编码：+591k GraphCodeBERT：172.5M UniXcoder：126.5M CodeGPT：124.4M StructCoder：223.4M
学习率	翻译：$5e^{-5}$ 摘要生成：$1e^{-5}$, $5e^{-5}$, $\mathbf{1e^{-6}}$, $5e^{-6}$ 合成：$5e^{-5}$
优化器	Adam（epsilon=$1e^{-8}$）
训练损失	交叉熵（困惑度）
验证指标	最佳 BLEU
束宽	10
CodeXGLUE	https://github.com/microsoft/CodeXGLUE
jsbuilder	https://github.com/tvst/jsbuilder
javalang	https://github.com/c2nes/javalang

表6：模型和训练配置。

B 翻译测试的更多检查

B.1 测试缺陷

我们在表7中展示了一些并非完全平行的翻译测试实例。在这些情况下，两种语言的代码执行时不会完全表现相同。

B.2 数值一致性

作为对第 3.3 节所报告数字准确性数据的补充，我们在表 9 中列出了包含数字的翻译输出，以供可视化查看。这表明，我们的 DA 模型即使面对非常冗长复杂的输出，也能确保数字一致性。例如，在基线输出中的测试#436里，数字错误还会导致不理想的幻觉，而 DA 模型的输出可以避免这种情况。

```
// 测试 #307
C# 源代码          public override string ToString(){return "IndexSearcher("}
                       + 读取器 + "; 执行器=" + 执行器 + ")";}
Java 参考 public String toString() {return "IndexSearcher("
                       + 读取器 + "; 执行器=" + 执行器
                       + "; 切片执行控制平面 " + 切片执行器 + ")";}

// 测试 #518
C# 源代码          公共重写 推送连接 OpenPush() 抛出
                       {新建 NGit.Errors.NotSupportedException(
                           JGitText.Get().pushIsNotSupportedForBundleTransport);}
Java 参考 public 推送连接 openPush() throws
                       {传输异常 return new TCP 推送连接();}

// 测试 #892
C# 源代码          public Builder():      base(){lastDocID = -1;wordNum = -1;word = 0;}
Java 引用 public Builder(){this(true);}

// 测试 #902
C# 源代码          public override string ToString(){return "术语="+ 术语+",                字段="
                       +字段+", 值="+值;}
Java 参考public String toString() {return "term="+term+",                field="
                       +字段+", 值="+valueToString()+",docIDUpto="+docIDUpto;}
```

表7：C#-Java 测试实例并非完全平行，差异以粗体标出。

通过增强实现 C 代码合成

对于代码合成，虽然反转摘要生成数据是一种自然的解决方案，但难点在于构建类环境（可见且可用的变量和方法）。我们解析摘要生成实例中的代码以获取正词元，同时从其他真实代码中随机采样词元作为负信号。换句话说，我们从 $PL \rightarrow NL$ 对中构建代码合成数据 $NL + parse(PL) + random(PL') \rightarrow PL$。

我们在 CodeXGLUE 的代码合成任务上进行实验，该任务从 CONCODE（Iyer 等人，2018年）中抽取数据，训练集/验证集/测试集分别为 100K/2K/2K。源数据包含文本描述以及可用的类变量和函数名称，目标则是对应的 Java 代码。我们反转 Java 摘要数据，生成 181K 条合成数据；为获取可用的变量和方法名称，我们使用 `javalang` 将代码解析为词元。遵循 CodeXGLUE 的做法，我们使用 CodeGPT-adapted 作为基础模型；此外，我们还使用 StructCoder（Tipirneni 等人，2022年）进行实验，它是一种更新的代码预训练语言模型。

输出采用 BLEU、精确匹配和 CodeBLEU 进行评估，与翻译任务类似。请注意，测试集的参考答案并未公开，且测试集的预测结果需要发送给 CodeXGLUE 作者进行评估，因此我们同时报告验证集和测试集上的结果，以便复现。

我们注意到，对于 CodeGPT，我们的增强工作使验证集和测试集的表现略有提升。然而，由于以下几个可能的原因，它并未超越最新的 PLM：1） StructCoder 明显强于 CodeGPT，因此提升空间有限；2）我们用于增强合成任务的摘要生成数据可能在主题、长度、风格等方面有所不同，从而导致领域漂移。

	BLEU	EM	CodeBLEU
CodeGPT 在验证集上			
复现	28.13	16.1	31.65
+ 增强	**29.04**	**16.6**	**32.35**
StructCoder 在验证集上			
复现	37.30	18.2	40.42
+ 增强	**37.48**	**18.7**	**40.47**
CodeGPT 关于测试			
论文	32.79	20.1	35.98
复现	32.66	**20.1**	35.89
+ 增强	**33.45**	19.2	**36.47**
StructCoder 测试			
论文	40.91	22.4	44.77
复现	**41.57**	**22.6**	**44.61**
+ 增强	41.32	21.4	44.04

表8：代码合成结果。

```
// 测试 #131
```

C# 源码

```
public 扩缩集群请求(): base("CS", "2015-12-15", "ScaleCluster", "cs", "开放 API"){URI 模式 = "/clusters/{集群 ID}";
方法 = MethodType.PUT;}
```

Java 参考

```
public ScaleClusterRequest() {super("CS", "2015-12-15", "ScaleCluster", "csk");setUriPattern("/clusters/{集群 ID}");
setMethod(MethodType.PUT);}
```

基线

```
publicscaleClusterRequest() {super("CS", "2018年-12-15", "ScaleCluster", "cs");setUriPattern("/clusters/{集群 ID}");
setMethod(MethodType.PUT);}
```

DA 模型

```
public ClusterRequest() {super("CS", "2015-12-15", "ScaleCluster", "cs");setUriPattern("/clusters/{集群 ID}");
setMethod(MethodType.PUT);}
```

```
// 测试#436
```

C# 源代码

```
public void 复制到(字节[] b, int o){FormatHexByte(b, o + 0,w1);
FormatHexByte(b, o + 8,w2);FormatHexByte(b, o + 16,w3);
FormatHexByte(b, o + 24, w4);FormatHexByte(b, o + 32, w5);}
```

Java 引用

```
public void copyTo(字节[] b, int o) {formatHexByte(b, o + 0,w1);
formatHexByte(b, o + 8,w2);formatHexByte(b, o + 16,w3);
formatHexByte(b, o + 24, w4);formatHexByte(b, o + 32, w5);}
```

基线

```
public void 复制到(byte[] b, int o) {formatHexByte(b, o1);
formatHexByte(b, o2);formatHexByte(b, o2);
formatHexByte(b, o3);formatHexByte(b,o + 24, w4);
formatHexByte(b, o + 32, w5);}
```

DA 模型

```
public void 复制到(int[] b, int o) {formatHexByte(b, o + 0,w1);
formatHexByte(b, o + 8,w2);formatHexByte(b, o + 16,w3);
formatHexByte(b, o + 24, w4);formatHexByte(b, o + 32, w5);}
```

```
// 测试 #716
```

C# 源码

```
public override void 解码(byte[] blocks, int 数据块Offset, int[] 值, 整数 值偏移, 整数 迭代次数){for (整数 j = 0;
j < 迭代次数; ++j){var 数据块 = 数据块[数据块偏移++];
值[值偏移++] =((整数)((uint)数据块 >> 7)) & 1;
值[值偏移++] =((整数)((uint)数据块 >> 6)) & 1;
值[值偏移++] =((int)((uint)数据块 >> 5)) & 1;
值[值偏移++] =((int)((uint)数据块 >> 4)) & 1;
值[值偏移++] =((int)((uint)数据块 >> 3)) & 1;
值[值偏移++] = ((int)((uint)数据块 >> 2)) & 1;
值[值偏移++] = ((int)((uint)数据块 >> 1)) & 1;
值[值偏移++] = 数据块 & 1;}}
```

Java 参考

```
public void 解码(byte[] 数据块, int 数据块偏移量, int[] 值, int 值偏移量, int 迭代次数) {for (int j = 0;
j < 迭代次数; ++j) {final 字节数据块 = 数据块[数据块偏移量++];
值[值偏移量++] = (数据块 >>> 7) & 1;
值[值偏移量++] = (数据块 >>> 6) & 1;
值[值偏移量++] = (数据块 >>> 5) & 1;
值[值偏移量++] = (数据块 >>> 4) & 1;
值[值偏移量++] =(数据块 >>> 3) & 1;
值[值偏移量++] = (数据块 >>> 2) & 1;
值[值偏移量++] = (数据块 >>> 1) & 1;
值[值偏移量++] = 数据块 & 1;}}
```

基线

```
public void 解码(字节[] 数据块, int 数据块偏移量, int[] 值, int 值偏移量, int 迭代次数) {for (int j = 0;
j < 迭代次数; ++j) {final 字节 数据块 = 数据块[数据块偏移量++];
值[值偏移量++] = (数据块 >>> 7) & 1;
值[值偏移量++] = (数据块 >>> 6) & 1;
值[值偏移量++] = (数据块 >>> 5) & 1;
值[值偏移量++] = (数据块 >>> 4) & 1;
值[值偏移量++] =(数据块 >>> 4) & 1;
值[值偏移量++] = (数据块 >>> 2) & 1;
值[值偏移量++] = (数据块 >>> 1) & 1;
值[值偏移量++] = 数据块 & 1;}}
```

DA 模型

```
public void 解码(字节[] 数据块, 整数 数据块偏移量, 整数[] 值, int values偏移量, int 迭代次数) {for (int j = 0;
j < 迭代次数; ++j) {final 字节 数据块 = 数据块[数据块偏移量++];
值[值偏移量++] = (数据块 >>> 7) & 1;
值[值偏移量++] = (数据块 >>> 6) & 1;
值[值偏移量++] = (数据块 >>> 5) & 1;
值[值偏移量++] = (数据块 >>> 4) & 1;
值[值偏移量++] =(数据块 >>> 3) & 1;
值[值偏移量++] = (数据块 >>> 2) & 1;
值[值偏移量++] = (数据块 >>> 1) & 1;
值[值偏移量++] = 数据块 & 1;}}
```

表9：数据增强前后包含数字的 C#-Java 输出翻译。