CodeT5：用于代码生成和理解的统一预训练编解码器模型论文复现报告

摘要

像 BERT 和 GPT 这样的自然语言（NL）的预训练模型最近已经被证明可以很好地转移到编程语言（PL）上，并在很大程度上有利于一系列与代码相关的任务。尽管它们取得了成功，但目前的大多数方法仅依赖于仅编码器（或仅解码器）的预训练，或者以 NL 相同的方式执行任务或处理代码片段，而忽略了 PL 的特殊特征，如标识符类型。本文作者提出了 CodeT5，一个统一的预先训练的编码-解码器转换模型，更好地利用从开发人员分配的标识符传递的代码语义。本文作者的模型采用了一个统一的框架来无缝地支持代码理解和生成任务，并允许多任务学习。此外，本文作者提出了一种新的标识符感知的预训练任务，使模型能够区分哪些代码标记是标识符，并在它们被掩蔽时恢复它们。此外，本文作者利用用户编写的代码注释与双模态双向生成任务，以更好的 NL-PL 对齐。综合实验表明，CodeT5 在理解代码缺陷检测和克隆检测等任务，以及跨 PL-NL、NL-PL 和 PL-PL 等不同方向的生成任务方面明显优于以往的方法。进一步的分析表明，该模型可以更好地从代码中捕获语义信息。

关键词： Transformer；自然语言处理；代码生成

1 引言

鉴于通过机器学习方法提高软件开发效率的目标，软件智能研究在过去十年中越来越受到学术界和工业界的关注。软件代码智能技术可以帮助开发人员减少繁琐的重复性工作量，提高编程质量，提高整体软件开发效率。这将减少编写软件所花费的时间，并降低计算和运营成本。在这项工作中，本文作者重点关注软件代码预训练的根本挑战，它具有推动软件开发生命周期中广泛下游应用程序的巨大潜力。

2 相关工作

2.1 自然语言的预训练

基于 Transformer 架构的预训练模型 [25] 在广泛的 NLP 任务上取得了最先进的性能。它们通常可以分为三组: 纯编码器模型，如 BERT [4]、RoBERTa [13] 和 ELECTRA [2]，纯解码器模型，如 GPT [17]，以及编码器-解码器模型，如 MASS [22]、BART [11] 和 T5 [18]。与仅支持理解任务和生成任务的仅支持编码器和仅支持解码器模型相比，编码器-解码器模型

可以很好地支持这两种类型的任务。它们通常采用去噪序列到序列的预训练目标，这些目标会破坏源输入并要求解码器恢复它们。在这项工作中，原文作者将 T5 扩展到编程语言中，并提出了一种新的标识符感知去噪目标，使模型能够更好地理解代码。

2.2 编程语言的预训练

程序设计语言的预训练。编程语言的预训练是一个新兴领域，最近的许多工作试图将 NLP 预训练方法扩展到源代码。CodeBERT [6] 是先锋模型。CodeBERT 添加了替换令牌检测任务来学习自然语言-编程语言跨模态表示。Transcoder [19] 探索了在无监督环境下的编程语言翻译。与他们不同的是，原文作者探索了基于 T5 的编码器-解码器模型，用于编程语言的预训练，并支持更全面的任务集。

一些新兴的工作 [3, 5, 15] 在最近的文献中也探讨了代码上的 T5 框架，但他们只关注生成任务的有限子集，不支持理解任务。除此之外，基于另一种编码器-解码器模型 BART 的 PLBART [1] 也可以支持理解和生成任务。然而，上述所有先前的工作都只是以与自然语言相同的方式处理代码，并且在很大程度上忽略了特定于代码的特征。相反，原文作者建议利用代码中的标识符信息进行预训练。

最近，GraphCodeBERT [7] 将从代码结构中提取的数据流合并到 CodeBERT 中，而 [20] 中提出了一个去混淆目标，以利用 PL 的结构方面。这些模型只专注于训练更好的特定于代码的编码器。 [28] 中建议捕获代码结构上代码令牌之间的相对距离。相比之下，原文作者特别关注保留丰富代码语义的标识符，并将这些信息通过两个新的方法：标识符标记和预测任务，融合到 Seq2Seq 模型中。

3 本文方法

3.1 本文方法概述

本文提出的 CodeT5 是构建在 T5 [18] 上的。它旨在通过对未标记的源代码进行预训练，推导出编程语言 (PL) 和自然语言 (NL) 的通用表示。如图 1所示，原文作者在 T5 中扩展了去噪 Seq2Seq 目标，提出了两个标识符标记和预测任务，以使模型能够更好地利用编程语言中的令牌类型信息，这些信息是开发人员分配的标识符。为了提高自然语言 (NL)-编程语言 (PL) 的一致性，原文作者进一步提出了一个双模态双向生成学习目标，用于 NL 和 PL 之间的双向转换。

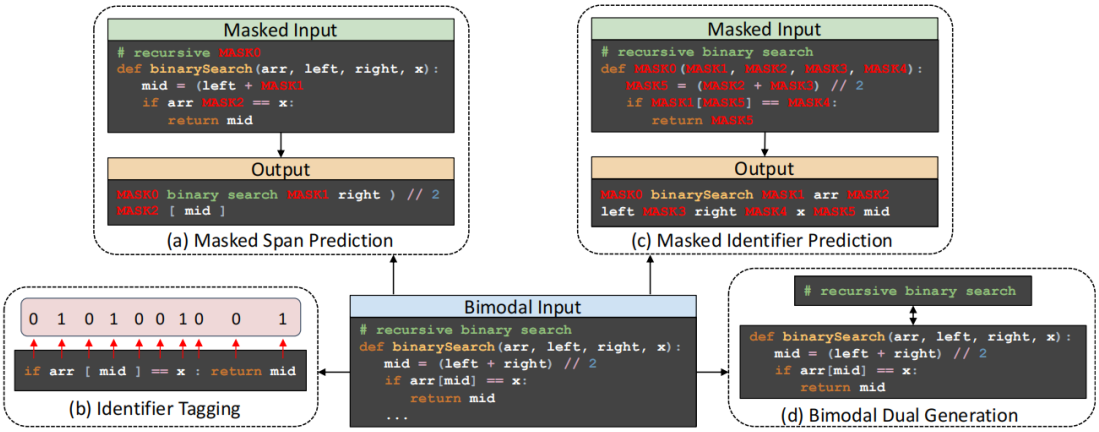

图 1. CodeT5 的预训练任务。首先在单模态和双模态数据上交替训练 MSP、MIP 和 IT，然后利用双模态数据进行双向生成训练。

3.2 对自然语言和编程语言编码

在预训练阶段，本文作者的模型将根据代码片段是否附带 NL 描述接收纯 PL 或 NL-PL 作为输入。对于 NL-PL 双模态输入，本文作者用分隔符 [SEP] 将它们连接成一个序列，并将整个输入序列表示为 $\mathrm{x} = ([CLS], \omega_1, ..., \omega_n, [SEP], c_1, ..., c_m, [SEP])$ 其中 n 和 m 分别表示 NL 字令牌个数和 PL 码令牌个数。对于仅限 PL 的单模态输入，NL 单词序列将为空。为了获取更多特定于代码的特性，本文作者建议利用代码中的令牌类型信息。作者专注于标识符的类型 (例如，函数名和变量)，因为它们是最与 PL 无关的特性之一，并且保留了丰富的代码语义。具体来说，本文作者将 PL 段转换为抽象语法树 (AST)，并提取每个代码标记的节点类型。最后，本文作者为每一个 PL 段构造了一个二元标记序列 $\mathbf{y} \in \{0,1\}^{\mathrm{m}}$，其中每个 $\mathrm{y_i} \in \{0,1\}$ 代表当前代码标识是否为标识符。

3.3 标识符感知预训练

去噪序列到序列 (Seq2Seq) 预训练已被证明在广泛的 NLP 任务中非常有效 [11, 18, 22]。这种去噪目标通常首先用一些去噪函数破坏源序列，然后要求解码器恢复原始文本。在这项工作中，作者使用了类似于 T5 的跨度屏蔽目标 [18]，该目标随机屏蔽任意长度的跨度，然后结合解码器中的一些哨兵令牌预测这些被屏蔽的跨度。

3.3.1 掩码间隔预测 (MSP)

作者采用与 T5 相同的 15% 的损坏率，并通过统一采样 1 到 5 个令牌的跨度来确保平均跨度长度为 3。此外，作者在子词标记化之前通过采样跨度采用了整个词屏蔽，目的是避免屏蔽部分子词，并且被证明是有帮助的 [23]。值得注意的是，我们为各种 PLs 预训练了一个共享模型，以学习鲁棒的跨语言表示，如图 1 (a) 所示。并将屏蔽跨度预测损失描述为:

$$\mathcal{L}_{\mathrm{MSP}(\theta)} = \sum_{\mathrm{t}=1}^{\mathrm{k}} -\log \mathrm{P}_\theta \left(\mathrm{x}_{\mathrm{t}}^{\mathrm{mask}} | \mathrm{x}^{\backslash \mathbf{mask}}, \mathbf{x}_{<\mathrm{t}}^{\mathrm{mask}}\right) \tag{1}$$

其中 θ 为模型参数，$\mathrm{x}^{\backslash mask}$ 为被掩码输入，x^{mask} 为从解码器预测的被掩码序列，k 表示 x^{mask} 中的令牌个数，$\mathbf{x}_{<\mathrm{t}}^{\mathrm{mask}}$ 是迄今为止生成的序列。

3.3.2 标识符标记

它的目的是告知模型该代码标记是否为标识符，这与某些开发工具中的语法突出显示精神类似。如图 1 (b) 所示，我们将 CodeT5 编码器处 PL 段的最终隐藏状态映射为概率序列 $\mathbf{p} = (p_1, ..., p_m)$，并计算二元交叉熵损失用于序列标记:

$$\mathcal{L}_{\mathrm{IT}(\theta)} = \sum_{\mathrm{j}=1}^{\mathrm{m}} - [\mathrm{y_i log p_i} + (1 - \mathrm{y_i}) \log (1 - \mathrm{p_i})] \tag{2}$$

其中 θ_e 为编码器参数。值得注意的是，通过将任务转换为序列标记问题，模型将捕获代码语法和代码的数据流结构

3.3.3 掩码标识符预测

与 MSP 中的随机跨度屏蔽不同，作者屏蔽了 PL 段中的所有标识符，并为一个特定标识符的所有出现使用唯一的哨兵令牌。在软件工程领域，这被称为混淆，其中更改标识符名称不会影响代码语义。受 [20] 的启发，我们将带有哨兵令牌的唯一标识符排列成目标序列 I，如图 1 (c) 所示。然后我们以自回归的方式进行预测:

$$\mathcal{L}_{\mathrm{MIP}(\theta)} = \sum_{\mathrm{j}=1}^{|\mathcal{I}|} - \log \mathrm{P}_\theta \left(\mathcal{I}_{\mathrm{j}} | \mathbf{x}^{\setminus \mathbf{I}}, \mathbf{I}_{<\mathrm{j}} \right) \tag{3}$$

其中 $\mathrm{x}^{\setminus I}$ 是被屏蔽的输入。请注意，去混淆是一项更具挑战性的任务，它要求模型理解基于混淆代码的代码语义，并将相同标识符的出现链接在一起。

作者以等概率交替优化这三种损失，这构成了我们提出的标识符感知去噪预训练。

3.4 双模态双向生成

在预训练阶段，解码器只看到离散的掩码范围和标识符，这与解码器需要生成流畅的 NL 文本或语法正确的代码片段的下游任务不同。为了缩小预训练和微调之间的差距，作者利用 NL-PL 双模态数据来训练模型进行双向转换，如图 1 (d) 所示。

具体来说，作者认为 NL 到 PL 生成和 PL 将自然语言生成作为双重任务，同时对其进行模型优化。对于每个 NL、PL 双模态数据点，作者构建两个方向相反的训练实例，并添加语言 id。这个操作也可以看作是 T5 的跨屏蔽的一个特殊情况，通过从双模态输入屏蔽整个 NL 或 PL 段。这项任务旨在改善 NL 和 PL 之间的一致性

3.5 微调 CodeT5

在大规模未标记数据上进行预训练后,作者通过特定任务迁移学习或多任务学习使 CodeT5 适应下游任务

3.5.1 特定任务的迁移学习

与代码相关的任务可以分为生成任务和理解任务。对于前者，CodeT5 可以自然地适应它的 Seq2Seq 框架。对于理解任务，作者研究了两种方法，一种是将标签生成为单图目标序列 [18]，另一种是基于 [11] 之后的最后一个解码器隐藏状态，从类标签词汇表中预测标签。

3.5.2 多任务学习

作者还通过一次在多个任务上训练共享模型来探索多任务学习设置。多任务学习能够通过在许多任务中重用大部分模型权重来降低计算成本，并且已被证明可以提高 NL 预训练中的模型泛化能力 [12]。

此外，作者还遵循 [18] 中对所有任务采用相同的统一模型，而不添加任何特定于任务的网络，但允许为不同的任务选择不同的最佳检查点。为了通知模型正在处理哪个任务，作者设计了任务控制代码的统一格式，并将其添加到源输入中，如图 2所示。例如，我们使用 “Translate Java to CSharp” 作为从 Java 到 CSharp 的代码到代码转换任务的源提示符。

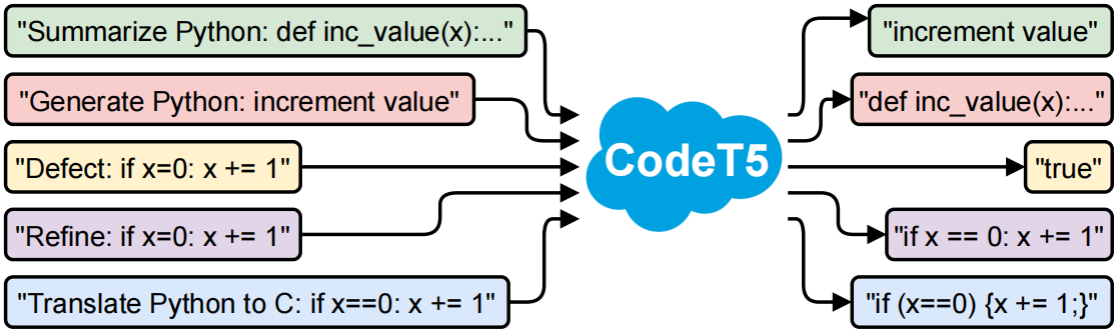

图 2. CodeT5 代码生成相关的理解和生成任务的演示

由于不同的任务具有不同的数据集大小，作者遵循 [10] 采用平衡采样策略。对于 N 个数据集 (或任务)，概率为 $\{q_i\}_{i=1}^{N}$ ，作者定义如下多项分布进行抽样:

$$q_i = \frac{r_i^{\alpha}}{\sum_{j=1}^{N} r_j^{\alpha}} \text{ ,where } r_i = \frac{n_i}{\sum_{k=1}^{N} n_k} \tag{4}$$

其中 n_i 是第 i 个任务的示例数，α 设为 0:7。这种平衡抽样旨在减轻对某些任务的偏见。

4 复现细节

4.1 与已有开源代码对比

这篇文章给出了完整的预训练以及评估代码以及经过了大量代码语料数据预训练过的模型，并不需要对代码进行结构上的修改。该代码目前包括两个预训练的模型（CodeT5-small 和 CodeT5-base）和用于在 4 个生成任务（代码摘要、代码生成、代码翻译和优化）以及 2 个理解任务（代码缺陷检测和克隆检测）上对其进行微调的代码。由于本次复现工作需要在预训练模型上进行上述六个特定任务的迁移学习并对经过微调后的模型输出结果进行评价，我们需要为对应任务收集相应的数据。

4.2 数据集收集

针对不同任务中不同子任务所对应的数据集描述如下表 1所示，从上到下依次是代码摘要、代码生成、代码翻译、代码优化、代码缺陷检测和代码克隆任务，但由于 CodeSearchNet [8]

表 1. 下游任务及对应数据集

任务	子任务	数据集描述
summarize	ruby/javascript/go/python/java/php	code summarization task on CodeSearchNet [8] data with six PLs
concode	none	text-to-code generation on Concode [9] data
translate	java-cs/cs-java	code-to-code translation between Java and C# [14]
refine	small/medium	code refinement on code repair data [24] with small/medium functions
defect	none	code defect detection [27] in C/C++ data
clone	none	code clone detection [26] in Java data

存放在 Amazon S3 数据库中的数据无法获取，我们采用了 [14] 以及该论文后续工作补充的数据集作为代码摘要生成任务的微调数据集。

4.3 实验环境搭建

1、安装特定的 Python 版本 3.10.12

2、安装 Pytorch 版本 1.7.1

https://pytorch.org/get-started/previous-versions/

3、安装其他依赖

pip install -r requirement.txt

4、使用服务器硬件加速器 A100(GPU) 进行复现工作

4.4 评价指标

1、BLEU-4

BLEU-4 [16] 是一种用于评估机器翻译结果的指标。BLEU-4 的计算过程分为两个步骤：计算短句惩罚系数和计算 n-gram 的精确匹配率。

计算短句惩罚系数 (BP)。短句惩罚系数主要用于惩罚过长的翻译结果。它通过计算参考答案和机器翻译结果的短句长度比例来确定。如果机器翻译结果的长度超过参考答案的长度，则 BP 小于等于 1, 否则 BP 等于 exp(1-参考答案长度/机器翻译结果长度)。短句惩罚系数的作用是防止机器生成过长的翻译结果，从而使得评估结果更加准确。

n-gram 的精确匹配率 (p_{n})。n-gram 是指由 n 个连续的词组成的序列。计算 n-gram 的精确匹配率需要统计机器翻译结果中 n-gram 的数量，并与参考答案中的 n-gram 进行比较。如果机器翻译结果中的 n-gram 在参考答案中也存在，则认为该 n-gram 是正确匹配的。精确匹配率的计算公式为：p_{n}= 机器翻译结果中正确匹配的 n-gram 数量/机器翻译结果中的 n-gram 数量。

综上所述 $BLEU_{\text{-}4}$ 计算公式如下：

$$BLEU_{\text{-}4} = BP \cdot exp\left(\sum_{N=1}^{4} \omega_{\mathrm{n}} log\left(\mathrm{p_n}\right)\right) \tag{5}$$

其中，BP 为短句惩罚系数,p_{n} 为 n-gram 的精确匹配率,exp 为指数函数。

2、Exact Match

Exact Match 是一种通过比较两个字符串每个字符是否相等以用于确定两个字符串是否完全相等的方法。在问答任务中可以衡量问题在回答任务中能否给出与参考答案完全一致的

表 2. 下游任务及对应 SOTA 预训练模型

任务类型	SOTA 预训练模型
代码摘要	RoBERTa [13]、CodeBERT [6]、DOBF [20]、PLBART [1]
代码生成	GPT-2 [17]、CodeGPT-2 [17]、CodeGPT-adapted [17]、PLBART [1]
代码翻译与优化	RoBERTa [13]、CodeBERT [6]、GraphCodeBERT [7]、PLBART [1]
代码缺陷与克隆检测	RoBERTa [13]、CodeBERT [6]、DOBF [20]、GraphCodeBERT [7]、PLBART [1]

回答。可用于评价模型在代码生成、翻译与优化任务中的表现。

3、Accuracy

用于评判二分类任务的准确性，也可用于评价其他分类任务的准确性。可以用于评估模型在缺陷检测任务性能。

4、F1 Score

F1 Score 是统计学中用来衡量二分类模型精确度的一种指标，同时兼顾了准确率与召回率，是两者的调和平均，反应模型查的又准又全的能力。经常用于评价不平衡数据集下的分类任务。

4.5 基线

对于不同的下游任务，我们将 CodeT5 与专注于特定任务的 SOTA 预训练模型进行比较，表 2展示了不同任务选择的 SOTA 模型。值得注意的是，我们在后续对比的数据是来源于原论文所给的数据。

4.6 使用说明

1、在自定义数据集中微调

若要对数据集进行微调，在 configs.py 中添加自己的任务和 sub_task：

```
parser.add_argument("--task", type=str, required=True,
                        choices=['summarize', 'concode',
                        'translate', 'refine', 'defect', 'clone'])
```

并在 utils.py 中：

```
def get_filenames(data_root, task, sub_task, split=''):
    if task == 'concode':
        data_dir = '{}/{}'.format(data_root, task)
        train_fn = '{}/train.json'.format(data_dir)
        dev_fn = '{}/dev.json'.format(data_dir)
        test_fn = '{}/test.json'.format(data_dir)
```

按照上述格式添加数据路径和要读取的函数。读取功能可以在 _utils.py 中实现：

```
def read_concode_examples(filename, data_num):
    examples = []
```

```
    with open(filename) as f:
        for idx, line in enumerate(f):
            x = json.loads(line)
            examples.append(
                Example(#根据数据格式修改内容
                    idx=idx,
                    source=x["nl"].strip(),
                    target=x["code"].strip()
                )
            )
            idx += 1
            if idx == data_num:
                break
    return examples
```

与此类似。若要添加的任务是生成任务，简单地重用 run_gen.py 即可。

2、运行代码

转到 *sh* 文件夹，将 exp_with_args.sh 中的 WORKDIR 设置为克隆的 CodeT5 存储库路径。使用 run_exp.py 通过简单地传递 model_tag、task 和 sub_task 参数来运行实验。对于每个任务，我们使用 sub_task 来指定要微调的特定数据集，如表 1所示。

例如，如果想在 Python 的代码克隆检测任务上运行基于 CodeT5 的预训练模型，可以在命令行中执行：

```
python run_exp.py --model_tag codet5_base --task summarize
    --sub_task python
```

3、评价模型性能

将 exp_with_args.sh 中的原始代码

```
--do_train --do_eval --do_eval_bleu --do_test
    ${MULTI_TASK_AUG}  \
```

更改为如下代码

```
--do_test ${MULTI_TASK_AUG}  \
```

并将第二步运行得到的检查点模型所保存的路径在 run_exp.py 中做如下更改

```
for criteria in ['best-bleu']:
            file = os.path.join(args.output_dir,
            '\保存的模型路径'.format(criteria))
            logger.info("Reload model from {}".format(file))
            model.load_state_dict(torch.load(file))
```

对于每一个子任务，需要在上述代码中更改为当前子任务在步骤二中产生的最优检查点的路径再运行，假如我们需要对经过克隆检测任务微调后的 CodeT5-base 模型进行评估，我

们需要运行如下代码

```
python run_exp.py --model_tag codet5_base --task clone
    --sub_task none
```

评估运行界面如图 3所示。

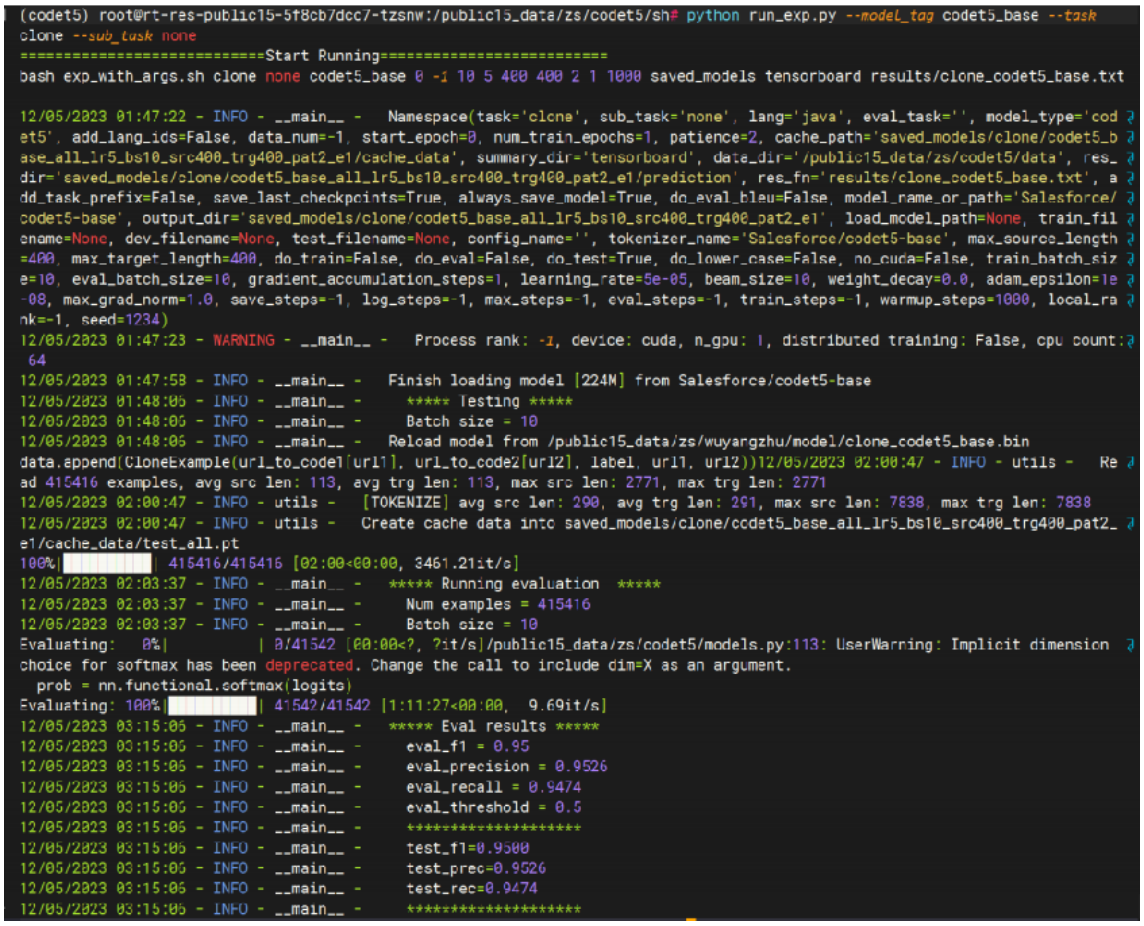

图 3. CodeT5-base 克隆检测任务运行结果演示

4.7 创新点

使用了自己收集的数据对 CodeT5-base 和 CodeT5-small 两个模型进行微调，根据自定义数据的格式修改了数据读取方式。并在代码生成任务上尝试使用将强化学习融入到预训练过程中，具体来说就是使用 Actor-Critic 算法，伪代码如下所示，将代码生成器作为 Actor 生成代码，使用另一个 Critic 网络对 Actor 生成的代码质量进行评估并回馈给 Actor 优化 Actor 生成的代码，评估的指标可以是当前代码通过测试的概率，此时，我们需要使用编译器以及问题描述-测试用例对作为环境来获取当前代码的运行情况。虽然通过 LeetCode、Stackoverflow 等等开源网站获取到了预训练所需的自然语言文本描述、测试用例以及正确的代码答案，但由于强化学习方法较难收敛，训练时长过长，此改进并未实现。后续将通过修改网络结构、调整编译器对于不同代码执行情况奖励机制的优化、使用外部经验记忆或更换使用 RLHF 中 PPO [21] 算法等等方式进行改进的实现。

Algorithm 1 Actor-Critic 算法

Input: env: 编译器环境; π_θ: Actor 网络; V_ϕ: Critic 网络; θ:Actor 网络参数; ϕ:Critic 网络参数; D: 经验回放区;

Output: 训练完成的两个模型

随机初始化两个网络的参数 θ ϕ 初始化经验回放区 D
repeat
    初始化状态 s
    根据策略 $\pi_\theta(s)$ 采样动作 a
    执行动作 a 并获得奖励 r 以及下一个状态 s'
    在经验回放区 D 中保存四元组数据 (s, a, r, s')
    从经验回放区 D 中采样一小批量数据
    repeat
        计算 TD error: $\delta_i = r_i + \gamma V_\phi(s'_i) - V_\phi(s_i)$
        反向传播修改 Critic 网络参数: $\phi \leftarrow \phi + \alpha_\phi \delta_i \nabla_\phi V_\phi(s_i)$
        反向传播修改 Actor 网络参数: $\theta \leftarrow \theta + \alpha_\theta \delta_i \nabla_\theta \log \pi_\theta(a_i|s_i)$
    until 遍历完批量中每一个四元组数据 (s_i, a_i, r_i, s'_i)
    更新状态: $s \leftarrow s'$
until 分幕结束

5 实验结果分析

在本节中，我们将 CodeT5 与 SOTA 模型在广泛的 CodeXGLUE 下游任务上进行比较。表 3展示了平滑 BLEU-4 对于六个编程语言数据的总结结果。原文数据中，所有的模型版本都明显优于之前的纯编码 (RoBERTa, CodeBERT, DOBF) 或编码器-解码器框架 (PLBART) 的工作。与 SOTA 编码器-解码器模型 PLBART 相比，发现具有更小规模的 CodeT5-small 也能产生更好的总体分数。作者将这种改进归因于标识符感知去噪预训练和更好地使用双模态训练数据。但可以看出我们复现的数据和原文数据有一定的差距，我们通过重新对模型使用同样的数据集进行评估，并找出其中大部分导致分数比较低的用例，我们发现了异常样本中存在语法错误的、与自然语言功能描述不一致的编程语言实现等等错误样本。我们将这样的差异归咎于我们收集数据的准确性和正确性，在后续工作中需要对错误数据进行清洗或增强。

表 4中将 CodeT5 与 GPT 风格的模型和 PLBART 进行比较，我们可以看出即使是小规模的 CodeT5-small 优于所有的解码器模型以及 PLBART，再次证明了编解码器模型在代码生成任务中的优越性。此外我们也可以发现我们复现的结果与原文相差不大。

表 5中比较了两种代码到代码的生成任务：代码翻译和代码优化，其中 Refine Small 和 Refine Medium 分别对应对于代码规模较小的和中等的方法级别代码进行优化。在代码翻译任务中，CodeT5-small 优于大多数基线，并获得与 PLBART 相当的结果，这表明编码器-解码器模型在代码到代码生成设置中的优势。CodeT5-base 在从 Java 到 c# 转换的各种指标上进一步实现了对 PLBART 的一致改进，反之亦然。代码优化，这是一项具有挑战性的任务，需要检测代码的哪些部分有 bug，并通过生成无 bug 的代码序列来修复它们，由于源代码和目标代码的大量重叠，即使是简单的复制方法也会产生非常高的 BLEU 分数，但没有精确匹

配，因此，原作者将重点放在精确匹配 (EM) 度量来评估此任务。可以观察到小规模的 EM 分数始终高于中等规模的分数，这表明较长的代码片段更难修复错误。此外，CodeT5-base 在 EM 上的表现明显优于所有基线，特别是在更具挑战性的中等任务中提高了 4.8 分以上 (13.96 相较于 GraphCodeBERT 的 9.10)。而我们的复现结果与原文相差不大。

表 6比较了缺陷检测和克隆检测两个理解任务。其中缺陷检测任务是根据解码器中生成的二进制标签作为检测目标，而克隆检测任务则将最后一个解码器状态获得的每个代码片段序列嵌入通过测量它们的相似性来预测标签。我们可以观察到 CodeT5-small 和 CodeT5-base 在缺陷检测任务上都优于所有基线，而 CodeT5-base 比 PLBART 的准确率提高了 2.6 分。对于克隆检测任务，CodeT5 模型实现了与 SOTA GraphCodeBERT 和 PLBART 模型相当的结果。这些结果表明，使用编解码器框架，CodeT5 仍然可以很好地适应理解任务。而我们的复现结果与原文相差不大。

表 3. 代码摘要任务的 Smoothed BLEU-4 分数。“总体”一栏显示了六个 PLs 的平均得分。CodeT5-base 与 CodeT5-small 为原文结果，RP-CodeT5-base 与 RP-CodeT5-small 为复现结果

Methods	Ruby	JavaScript	Go	Python	Java	PHP	Overall
RoBERTa	11.17	11.90	17.72	18.14	16.47	24.02	16.57
CodeBERT	12.16	14.90	18.07	19.06	17.65	25.16	17.83
DOBF	-	-	-	18.24	19.05	-	-
PLBART	14.11	15.56	18.91	19.30	18.43	23.58	18.32
CodeT5-small	14.87	15.32	19.25	20.04	19.92	25.64	19.14
CodeT5-base	15.24	16.16	19.56	20.01	20.31	26.03	19.55
RP-CodeT5-small	11.21	11.60	17.02	15.99	12.56	19.03	14.57
RP-CodeT5-base	11.38	11.72	16.13	16.41	12.60	19.15	14.58

表 4. 代码生成任务的总体结果。EM 表示精确匹配。CodeT5-base 与 CodeT5-small 为原文结果，RP-CodeT5-base 与 RP-CodeT5-small 为复现结果

Methods	EM	BLEU	CodeBLEU
GPT-2	17.35	25.37	29.69
CodeGPT-2	18.25	28.69	32.71
CodeGPT-adapted	20.10	32.79	35.98
PLBART	18.75	36.69	38.52
CodeT5-small	21.55	38.13	41.39
CodeT5-base	22.30	40.73	43.20
RP-CodeT5-small	22.15	39.60	43.79
RP-CodeT5-base	22.65	42.66	45.04

表 5. 代码翻译与优化任务的总体结果。EM 表示精确匹配。CodeT5-base 与 CodeT5-small 为原文结果，RP-CodeT5-base 与 RP-CodeT5-small 为复现结果

Methods	JAVA to C#		C# to JAVA		Refine Small		Refine Medium	
	BLEU	EM	BLEU	EM	BLEU	EM	BLEU	EM
RoBERTa(code)	77.46	56.10	71.99	57.90	77.30	15.90	90.07	4.10
CodeBERT	79.92	59.00	71.99	57.90	77.30	15.90	90.07	5.20
GraphCodeBERT	80.58	59.40	72.64	58.80	80.02	17.30	91.31	9.10
PLBART	83.02	64.60	78.35	65.00	77.02	19.21	88.50	8.98
CodeT5-small	82.98	64.10	79.10	65.60	76.23	19.06	89.20	10.92
CodeT5-base	84.03	65.90	79.87	66.90	77.43	21.61	87.64	13.96
RP-CodeT5-small	84.30	66.30	79.68	67.40	77.82	21.30	89.22	14.44
RP-CodeT5-base	83.56	66.00	79.77	67.00	77.38	21.70	89.33	14.77

表 6. 代码缺陷检测和克隆检测任务的结果。CodeT5-base 与 CodeT5-small 为原文结果，RP-CodeT5-base 与 RP-CodeT5-small 为复现结果

Methods	Defect Accuracy	Clone F1
RoBERTa	61.05	94.9
CodeBERT	62.08	96.5
DOBF	-	96.5
GraphCodeBERT	-	97.1
PLBART	63.18	97.2
CodeT5-small	63.40	97.1
CodeT5-base	65.78	97.2
RP-CodeT5-small	64.13	95.1
RP-CodeT5-base	64.36	95.3

6 总结与展望

本文作者提出了 CodeT5，这是一个预训练的编码器-解码器模型，它包含了代码中的令牌类型信息。此外，作者还提出了一个新的标识符感知预训练目标，以更好地利用标识符，并提出了一个双模态双向生成任务，以使用代码及其注释学习更好的 NL-PL 对齐。该模型可以支持代码理解和生成任务，并允许多任务学习。实验表明，CodeT5 在大多数 CodeXGLUE 任务中显著优于所有先前的工作。进一步的分析还表明，它具有跨各种编程语言的更好的代码理解能力。

在本次复现过程中，我们通过收集数据、配置环境、修改并运行代码等等工作对实验结果进行复现。实验过程中我们收集到的数据部分存在问题导致在代码摘要任务上与原文的结果相差较大。但在代码生成、代码翻译、代码优化、代码缺陷检测和代码克隆检测任务中我

们复现的结果与原文相差不大。虽然在复现过程中由于强化学习的性质以及数据集的问题未能实现前面创新点提出的意见，在未来我们可以针对这些问题对其进行研究和完善。

虽然本文提出的预训练方法能很好的利用代码标识符等其他信息用于代码生成任务，但这种生成方式与生成任务的目标有一定的差距。因此我们可以在预训练过程中使用类似 GPT 预训练时使用的 Next Token Prediction 来弥补这个差距。此外，当前模型在面向简单任务进行代码生成时可以取得较好的结果，但将 Leetcode 等其他 OJ 网站的编程题目输入到模型并要求模型输出相应代码时，输出的结果并不能通过测试用例。面对这个问题，我们可以更进一步利用自然语言描述中的测试用例以及生成的代码在编译器中的运行结果来辅助模型在困难的编程任务中生成更好的代码。

参考文献

[1] Wasi Uddin Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. Unified pre-training for program understanding and generation. *arXiv preprint arXiv:2103.06333*, 2021.

[2] Kevin Clark, Minh-Thang Luong, Quoc V Le, and Christopher D Manning. Electra: Pre-training text encoders as discriminators rather than generators. *arXiv preprint arXiv:2003.10555*, 2020.

[3] Colin B Clement, Dawn Drain, Jonathan Timcheck, Alexey Svyatkovskiy, and Neel Sundaresan. Pymt5: multi-mode translation of natural language and python code with transformers. *arXiv preprint arXiv:2010.03150*, 2020.

[4] 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.

[5] Ahmed Elnaggar, Wei Ding, Llion Jones, Tom Gibbs, Tamas Feher, Christoph Angerer, Silvia Severini, Florian Matthes, and Burkhard Rost. Codetrans: Towards cracking the language of silicon's code through self-supervised deep learning and high performance computing. *arXiv preprint arXiv:2104.02443*, 2021.

[6] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. Codebert: A pre-trained model for programming and natural languages. *arXiv preprint arXiv:2002.08155*, 2020.

[7] Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, et al. Graphcodebert: Pre-training code representations with data flow. *arXiv preprint arXiv:2009.08366*, 2020.

[8] Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. Codesearchnet challenge: Evaluating the state of semantic code search. *arXiv preprint arXiv:1909.09436*, 2019.

[9] Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, and Luke Zettlemoyer. Mapping language to code in programmatic context. *arXiv preprint arXiv:1808.09588*, 2018.

[10] Guillaume Lample and Alexis Conneau. Cross-lingual language model pretraining. *arXiv preprint arXiv:1901.07291*, 2019.

[11] Mike Lewis, Yinhan Liu, Naman Goyal, Marjan Ghazvininejad, Abdelrahman Mohamed, Omer Levy, Ves Stoyanov, and Luke Zettlemoyer. Bart: Denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension. *arXiv preprint arXiv:1910.13461*, 2019.

[12] Xiaodong Liu, Pengcheng He, Weizhu Chen, and Jianfeng Gao. Multi-task deep neural networks for natural language understanding. *arXiv preprint arXiv:1901.11504*, 2019.

[13] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. Roberta: A robustly optimized bert pretraining approach. *arXiv preprint arXiv:1907.11692*, 2019.

[14] Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin Clement, Dawn Drain, Daxin Jiang, Duyu Tang, et al. Codexglue: A machine learning benchmark dataset for code understanding and generation. *arXiv preprint arXiv:2102.04664*, 2021.

[15] Antonio Mastropaolo, Simone Scalabrino, Nathan Cooper, David Nader Palacio, Denys Poshyvanyk, Rocco Oliveto, and Gabriele Bavota. Studying the usage of text-to-text transfer transformer to support code-related tasks. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 336–347. IEEE, 2021.

[16] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th annual meeting of the Association for Computational Linguistics*, pages 311–318, 2002.

[17] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. Language models are unsupervised multitask learners. *OpenAI blog*, 1(8):9, 2019.

[18] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J Liu. Exploring the limits of transfer learning with a unified text-to-text transformer. *The Journal of Machine Learning Research*, 21(1):5485–5551, 2020.

[19] Baptiste Roziere, Marie-Anne Lachaux, Lowik Chanussot, and Guillaume Lample. Unsupervised translation of programming languages. *Advances in Neural Information Processing Systems*, 33:20601–20611, 2020.

[20] Baptiste Roziere, Marie-Anne Lachaux, Marc Szafraniec, and Guillaume Lample. Dobf: A deobfuscation pre-training objective for programming languages. *arXiv preprint arXiv:2102.07492*, 2021.

[21] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

[22] Kaitao Song, Xu Tan, Tao Qin, Jianfeng Lu, and Tie-Yan Liu. Mass: Masked sequence to sequence pre-training for language generation. *arXiv preprint arXiv:1905.02450*, 2019.

[23] Yu Sun, Shuohuan Wang, Yukun Li, Shikun Feng, Xuyi Chen, Han Zhang, Xin Tian, Danxiang Zhu, Hao Tian, and Hua Wu. Ernie: Enhanced representation through knowledge integration. *arXiv preprint arXiv:1904.09223*, 2019.

[24] Michele Tufano, Cody Watson, Gabriele Bavota, Massimiliano Di Penta, Martin White, and Denys Poshyvanyk. An empirical study on learning bug-fixing patches in the wild via neural machine translation. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 28(4):1–29, 2019.

[25] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

[26] Wenhan Wang, Ge Li, Bo Ma, Xin Xia, and Zhi Jin. Detecting code clones with graph neural network and flow-augmented abstract syntax tree. In *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 261–271. IEEE, 2020.

[27] Yaqin Zhou, Shangqing Liu, Jingkai Siow, Xiaoning Du, and Yang Liu. Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks. *Advances in neural information processing systems*, 32, 2019.

[28] Daniel Zügner, Tobias Kirschstein, Michele Catasta, Jure Leskovec, and Stephan Günnemann. Language-agnostic representation learning of source code from structure and context. *arXiv preprint arXiv:2103.11318*, 2021.