迈向预训练代码模型的高效微调：一项实验研究及后续探索

Ensheng Shi[a,†] YanlinWang[b,§,†] Hongyu Zhang[c]Lun Du[d] ShiHan[d]
Dongmei Zhang[d] HongbinSun[a,§][a]西安交通大学 b中山大学软件工程学院[c]重庆大学 d微软 s1530129650@stu.xjtu.edu.cn, wangylin36@mail.sysu.edu.cn, hyzhang@cqu.edu.cn {lun.du, shihan, dongmeiz}@microsoft.com,hsun@mail.xjtu.edu.cn

arXiv:2304.05216v1 [cs.SE] 11 Apr 2023

摘要

近年来，在下游任务上微调 CodeBERT 等预训练代码模型，已在许多软件测试与分析任务中取得显著成功。尽管这种方法有效且广泛应用，但微调预训练参数会带来高昂的计算成本。本文开展了一项广泛的实验研究，探索微调过程中逐层预训练表征及其编码的代码知识会发生什么变化。基于上述发现，我们进一步提出高效微调大型预训练代码模型的替代方案。实验研究表明：(1) 源代码的词汇、句法和结构属性分别编码在低层、中间层和高层中，而语义属性则贯穿整个模型。(2) 微调过程保留了大部分代码属性。具体而言，低层和中间层捕获的基本代码属性在微调过程中仍得以保留。此外，我们发现，对于各种下游任务，微调过程中变化最大的只有顶层的两层表征。(3) 基于上述发现，我们提出**Telly**，通过层冻结高效微调预训练代码模型。在五种不同下游任务上的大量实验结果表明，训练参数和相应耗时均大幅减少，同时性能相当或更优。

CCS 概念

• **软件及其工程 → 软件开发技术**；可复用性。

关键词

实证研究、预训练语言模型、高效微调、探测技术、表征相似性分析

特此授权免费制作本作品全部或部分内容的数字版或纸质版副本，供个人或课堂使用，但前提是不得为盈利或商业利益而制作或分发副本，并且副本首页须附有本声明及完整引文。对于本作品中由非ACM其他方拥有版权的组件，必须尊重其版权。经注明出处后允许摘编。若要以其他方式复制、再版、发布到服务器或重新分发到列表，则需事先获得具体许可和/或支付费用。请通过permissions@acm.org向我们申请许可。*ISSTA 2023，2023*年*7*月*17*日至*21*日，美国西雅图© 2018年计算机协会。ACMISBN 978-1-4503-XXXX-X/18/06...$15.00 https://doi.org/XXXXXXX.XXXXXXX

1 引言

近年来，预训练与微调范式 [2, 15, 17, 34]在许多软件测试和分析任务中取得了显著进展，例如漏洞检测 [13, 17], 补丁生成 [9, 18], 自动程序修复 [27], 代码审查 [42,55],代码生成 [2, 9, 17], 和克隆检测 [13, 17, 18]。这类方法首先在大量数据上预训练大型 Transformer 基础模型，以学习通用代码表示。随后，为使这些模型适应下游任务，通常会针对特定任务对其进行微调 [2, 15, 17, 18, 40, 55, 63]。

在这一范式下，微调预训练代码模型通常能在下游任务上取得显著更好的结果。尽管微调有效，但微调预训练参数会带来巨大的计算成本和同样巨大的能耗。据 CodeXGLUE [34], 报告，在配备两张 P100 显卡的机器上，为下游任务微调预训练模型通常需要 10 个多小时。尤其是，随着预训练模型或微调数据集变得更大，计算成本也会变得更高。例如，参数量约为 220MB 的 CodeT5 [63]在 CONCODE [24] 数据集上进行代码生成微调时，耗时超过 40 小时。事实上，这些做法与低碳深度学习 [48]背道而驰。在软件工程领域，只有少数研究探讨了预训练代码模型在微调过程中的变化。大多数相关研究 [15, 21, 30, 54, 58] 旨在了解预训练代码模型对源代码的认知。**显然，我们有必要了解预训练代码模型在微调过程中会发生什么，并进一步以更低的计算成本高效地将预训练模型适配到下游任务**。

本文首先探究预训练代码模型的逐层表示中编码了哪些代码属性，以及微调过程中这些表示会发生什么变化。随后，我们基于上述发现，为预训练代码模型提出了一些高效的微调替代方案。具体而言，首先，受编译过程 [3] 和静态程序分析技术 [38], 的启发，我们提出了四项探测任务（见第 3.2.1 节），涉及词汇、语法、语义和结构代码属性。接下来，我们开展实证研究，探究预训练代码模型中编码了哪些代码属性，以及不同层对理解这些编码属性有何贡献

§Yanlin Wang 和 Hongbin Sun 为通讯作者。
†该工作是在作者任职于微软亚洲研究院期间完成的。

这些编码属性。此外，我们针对五项各不相同的下游任务（见表1）开展了广泛的实验研究，深入探究微调过程中逐层表示会发生什么变化。这些任务包括代码搜索 [16, 46]、克隆检测 [11, 49]、代码摘要 [47, 61]、代码生成 [25] 和行级代码补全 [34]。通过大量实验，我们得出了关于预训练代码模型的以下主要发现。

第一个主要发现是，预训练代码模型主要在低层编码源代码的词汇属性，主要在中间层识别句法属性，并主要在高层理解结构属性。语义属性则由整个模型的各层共同感知。第二个主要发现是，微调过程保留了大部分代码属性。也就是说，在微调过程中，低层和中间层编码的基本代码知识（或属性）仍得以保留。只有高层捕获的知识变化最大。此外，我们的实验研究表明，在五种不同的下游任务上微调预训练模型时，低层的表示变化很小，只有最顶端的两层发生了显著变化。

基于上述发现，我们提出了通过层冻结实现预训练代码模型高效微调的 **Telly**-K。不同的 K 值表示我们方法的不同变体。具体而言，我们冻结底部 K 层的预训练参数，以减少训练参数；这些层在微调过程中变化不明显，其中 $K \in [0, 1, 2, 3, ..., L\text{-}1]$，第 0 层是嵌入层，$L$ 是预训练代码模型的最大层数（通常为 12）。因此，Telly-1 表示冻结嵌入层和第 1 个编码器层。我们从训练参数、耗时和性能三个方面，对来自三个方面的五个不同下游任务进行了广泛实验。所评估的预训练代码模型具有 12 个隐藏层。实验结果表明：(1) 对于几乎所有 Telly-K（$0 \le K \le 11$），训练耗时和训练参数都显著减少，而模型性能没有明显变化。(2) 冻结底部 $K(0 \le K \le 5)$层时，训练参数减少约 30% 至 65%，训练时间相应节省约 10% 至 75%。对于不同的下游任务，模型性能通常提高 1% 至 4%。(3) 冻结底部 $K(6 \le K \le 9)$ 层时，训练参数减少 65% 至 80%，相应地节省约 50% 至 80% 的训练时间，而模型性能仅发生轻微变化。(4) 当冻结层数大于九层（$10 \le K \le 11$）时，训练参数及相应的训练耗时大幅减少，但模型性能也显著下降。

我们的主要贡献总结如下：

- 我们提出了四项与词法、句法、语义和结构代码属性相关的探测任务。通过上述探测任务，我们探索了代码属性是什么，以及它们如何编码在逐层表示中。
- 据我们所知，我们首次开展了广泛的实验研究，分析预训练代码模型在微调过程中逐层表示及其编码的代码属性会发生什么变化。
- 我们提出了一种高效的方法，通过层冻结将预训练代码模型微调至下游任务。此外，我们在五种不同的下游任务上进行了大量实验，以证明该方法的效率。

本文其余部分的组织结构如下。第 2 节介绍相关背景知识。随后，我们在第 3 节开展实验研究，以了解预训练代码模型在微调过程中会发生什么。基于上述发现，我们在第 4 节提出 Telly-K，并在五种不同的下游任务上开展广泛实验，以展示其优势。第 5 节讨论缩短微调时间的重要性、改进微调的可行指南以及实验发现的普适性，并指出一些效度威胁。第 6 节介绍相关工作。最后，我们在第 7 节总结本文并讨论未来工作。

2 背景

2.1 预训练代码模型

大型预训练模型在许多领域都取得了显著成果，包括自然语言处理 [12, 33]、计算机视觉 [7, 19] 和软件工程 [2, 15, 17, 18, 64]。在软件工程领域，通常会先使用大量与源代码相关的数据对大型模型进行预训练，然后在下游任务上对其进行微调，以提升其性能。近期，已有许多预训练代码模型 [2, 15, 17, 18, 40, 63] 被提出，并在涉及软件测试、安全、维护和开发 [13, 17, 34, 42, 55] 的多项软件工程任务中展现出令人惊喜的潜力。我们从以下三个方面介绍这些模型。

基本架构。 最新的预训练代码模型通常采用多层 Transformer 模型 [56]（通常为 Transformer编码器）作为基本架构。Transformer编码器本质上由嵌入层、位置编码器和一组编码器层组成。一般而言，给定一段输入代码片段，首先通过嵌入层和位置编码器对其进行嵌入，以获得初始词嵌入。接下来，将它们输入多个堆叠的编码器层，逐层对输入信息进行编码。

从数学上来说，我们将输入标记表示为 $T = [t_1, t_2, ..., t_n]$，其中 n 是输入标记序列的长度。嵌入层将每个标记映射到高维语义空间。位置编码器用于对位置信息进行编码，然后通过以下方式将其注入输入嵌入中：

$$w_i = embed(t_i) + pos(t_i), \quad i = 1, 2, ..., n \tag{1}$$

其中，$embedd(*)$和 $pos(*)$分别表示嵌入层和位置编码器。$W = [w_1, w_2, ..., w_n]$ 是初始词嵌入。接下来，多个堆叠的编码器层通过以下方式生成一组逐层的上下文表示 H^0，H^1，$...\ H^L$：

$$\begin{aligned} H^0 &= [w_1, w_2, ..., w_n] \\ H^l &= encoder^l(H^{l-1}), \quad l = 1, 2, ..., L \end{aligned} \tag{2}$$

其中，L 是堆叠层的数量，$encoder^l(*)$表示第 l 个编码器层。$H^l = [h_1^l, h_2^l, ..., h_n^l]$ 表示第 l 层的上下文表示，H^0 是初始词嵌入。

预训练。预训练技术作为一种自监督学习方法，可以利用大型模型通过大量无标签数据集学习通用表示 [12, 19, 33, 43]。通常，这类技术会自动从无标签样本中生成虚拟标签，将无监督学习问题重新表述为监督学习问题。相应的监督任务称为预训练任务。例如，CodeBERT [15] 采用具有 768 维嵌入的 12 层 Transformer 编码器，并在名为 CodeSearchNet [23] 的大规模数据集上对所有参数进行预训练。该数据集包含 210 万条双模态数据（代码函数及其对应的自然语言注释），以及涵盖六种编程语言（Ruby、JavaScript、Go、Python、Java、PHP）的 640 万条单模态代码，并采用两种预训练任务，即掩码语言建模和替换令牌检测。前者用于预测被掩码位置的原始令牌，后者用于判断一个令牌是否为原始令牌。UniXcoder [17] 以代码/文本序列为输入，并在来自 T5 [43] 的 C4 数据集和来自 CodeSearchNet 的 410 万条单模态代码上，通过五种不同的预训练任务进行预训练。

微调。在海量数据集上完成预训练后，它们通过在目标数据集 [2, 14, 15, 17, 18, 40, 63] [17, 34] 上对所有预训练参数进行微调，使预训练模型适应下游任务。例如，在代码搜索中，先前的研究将模型的最后一层上下文表示（例如 H^L）进行平均，作为整体表示，通过向量距离衡量源代码与查询的表示之间的相似度，并通过拉近配对的代码和查询、推远未配对的代码和查询来对其进行微调。与预训练模型相比，在代码搜索上，CodeBERT和GraphCodeBERT 的微调模型的平均倒数排名值分别从0.001 和 0.156 提升至 0.694 和 0.713。

2.2 探测技术

探测技术已被自然语言处理领域广泛用于研究预训练语言模型捕获了哪些语言属性。具体来说，它们从预训练模型中提取上下文表示（例如 H^1、H^2）作为冻结特征，将其输入线性分类器，并且只训练该分类器，使其预测与语言属性相关的探测任务。它们还将随机表示作为基线，以比较并展示预训练表示编码语言属性的能力。例如，Tenney 等人 [53] 利用词性标注、依存句法分析、语义角色标注和共指消解标注等不同探测任务，考察预训练模型理解词性、依存关系、语义和共指关系等语言属性的能力。在软件工程领域，大多数相关研究 [21, 30, 58] 旨在了解预训练代码模型对源代码掌握了哪些信息。例如，Wan 等人 [58] 提出了一项新的探测任务，用于研究抽象语法树的结构是否编码在预训练代码模型的表示中。具体来说，他们从代码预训练模型中提取两个输入代码标记的逐层预训练表示，将其输入一个矩阵，并训练该矩阵重建抽象语法树中两个对应终端节点之间的距离，其中该距离用于表示句法结构。López 等人 [21] 提出了一项名为的全新探测任务。AST-Probe 能够从预训练语言模型的隐藏表征中恢复抽象语法树。具体而言，AST-probe 首先使用正交投影将预训练代码模型的逐层表示映射到一个称为句法子空间的潜在空间，然后利用该空间的几何结构预测输入代码片段的抽象语法树。Karmakar 等人 [30] 还提出了一些新的探测任务，用于研究预训练语言模型是否能够理解一些简单的代码属性。受编译过程 [3] 和静态分析技术 [38],的启发，我们提出了四项与不同代码属性相关的探测任务（第3.2.1 节）。除了探测预训练代码模型之外，我们还研究了它们在微调过程中的逐层表示。

2.3 表征相似性分析

表征相似性分析（RSA）最初用于认知神经科学 [31]，以研究人脑中的神经激活模式与计算模型针对一组给定刺激所形成的表示之间的关系。近年来，该方法被用于衡量两个表征空间之间的相似性 [1, 10, 37]。例如，给定一组输入，不同模型生成的不同表征空间会产生不同的表示。Merchant 等人 [37] 构建了两个距离矩阵，每个矩阵记录一个表征空间中各表示之间的向量距离（例如余弦相似度）。随后，通过计算这两个距离矩阵的皮尔逊相关系数 [45]，衡量两个表征空间的相似性。通常，相关系数介于 0.8 和 1 之间，表明两个表征空间相当相似；而低于 0.5 则意味着两个表征空间并不相似 [4, 45]。本文在第 3.3 节和第 3.5 节进行表征相似性分析，以研究预训练模型与微调模型的逐层表示之间的相似性。

3 预训练代码模型的实验研究

3.1 研究问题

尽管微调预训练代码模型有效且普遍，但会带来高昂的计算成本。在本工作中，我们首先开展实验研究，以探究逐层预训练表征中编码了哪些代码属性，以及这些表征在微调过程中会发生什么变化。关于该实验研究的研究问题详述如下。

研究问题*1*：逐层预训练表示中编码了哪些代码属性？探测技术已被自然语言处理领域广泛用于分析和解读预训练语言模型。受编译过程 [3] 和静态分析技术 [38], 的启发，我们首先提出了四项探测任务，分别涉及源代码的词汇、句法、语义和结构属性。这些任务将在第 3.2.1 节中详细介绍。随后，我们在第 3.2.2 节中通过上述探测任务，考察逐层预训练表征编码了哪些代码属性。同时，我们研究每一层的表示对理解这些代码属性有多大贡献。此外，我们还针对在下游任务中微调的模型进行相同的探测实验，

相关内容见第 3.5 节，并直观地了解预训练模型在微调过程中捕获的代码属性发生了什么变化。

研究问题*2*：微调期间逐层表示会发生什么变化？在研究问题1中，我们旨在借助探测任务，大致了解预训练代码模型在微调时会发生什么变化。我们进一步开展广泛的表示相似性分析（RSA），研究在下游任务上对预训练表示进行逐层微调时，这些表示会发生什么变化。第2.3节介绍的RSA是一种与任务无关的技术，不需要预先了解探测任务。第3.3节描述了如何将RSA应用于预训练模型和微调后的模型。为确保实验结果具有普遍性，我们在五种不同的下游任务上开展实验，包括代码搜索、克隆检测、代码摘要、代码生成和行级代码补全。

3.2 探测预训练代码模型

我们介绍四项与代码相关的探测任务和探测流程如下。

3.2.1 四项探测任务。我们在图 1 中设计并展示了这四项探测任务。它们分别与词汇、句法、语义和结构代码属性相关。下面逐一详细介绍。

词汇探测。 词汇探测旨在衡量上下文表示对源代码词汇属性的编码能力。众所周知，当源代码被编译时，第一步是词法分析，该步骤会对源代码字符串进行标记化，并确定每个代码标记的类型（例如 标识符、关键字）。不同类型在后续的程序分析和编译中发挥着截然不同的语义或句法作用。因此，了解预训练代码模型是否通过上下文表示捕获了源代码的词汇信息非常重要。为此，我们首先将预训练代码模型的上下文表示用作冻结特征，然后将其输入线性分类器，最后对其进行训练，以预测每个代码标记的类型。如图 1(d) 所示，每个标记属于五种类型之一，包括 标识符、关键字、算子、数字和字符串。由于篇幅限制，每种类型的详细定义以及对词汇探测的描述可在复现包的在线附录中找到 [51]。

句法探测。句法分析通常紧随程序编译过程中的词法分析之后 [3], ，在这一过程中，解析器将词法分析器生成的代码标记序列作为输入，并生成解析树或抽象语法树（AST）等数据结构。类似地，句法探测旨在研究上下文表示感知源代码句法属性的能力。其基本思想是识别一段代码与一个匿名 AST（名为 AST-Only，如图1(b)所示）是否匹配。具体而言，我们首先解析源代码以获得相应的 AST，其中包括非终端节点和终端节点。非终端节点表示句法信息，而终端节点由与句法元素对应的类型以及与源代码中的代码标记对应的值组成。模型很容易识别出 AST 是根据代码标记的重叠情况，由 onecode 片段进行解析。因此，如图1(b)所示，我们通过移除终端节点的值来构造仅抽象语法树。接下来，我们训练一个线性分类器，用于判断给定的仅抽象语法树是否由给定的代码片段解析而来。真实配对通过将代码与相应解析得到的仅抽象语法树配对来构造，而错误配对则通过将代码与由其他不同代码解析得到的仅抽象语法树配对来构造。句法探测的详细说明见在线附录 [51]。

语义探测。 为了了解预训练代码模型在多大程度上能够感知代码语义，我们进行了语义探测（图1(e)），考察其识别具有相同语义但实现方式不同的代码片段的能力。具体而言，我们使用 POJ-104 [39] 数据集作为评估数据集，该数据集包含 104 个问题，每个问题对应 500 个 C/C++ 实现。我们训练了一个线性映射器，以预训练表示作为输入，将语义相似的代码片段映射到相似嵌入。因此，可以通过它们之间的向量距离轻松召回具有相同语义的实现。关于语义探测的详细描述请参见在线附录 [51]。

结构探测。 除了词法、句法和语义属性之外，结构属性对于代码分析也十分重要。圈复杂度 [36], 表示程序的复杂程度，并可通过控制流图（CFG）体现，因此可以作为代码的结构属性。从数学上讲，圈复杂度 M 可以根据源代码的 CFG 计算，公式如下：

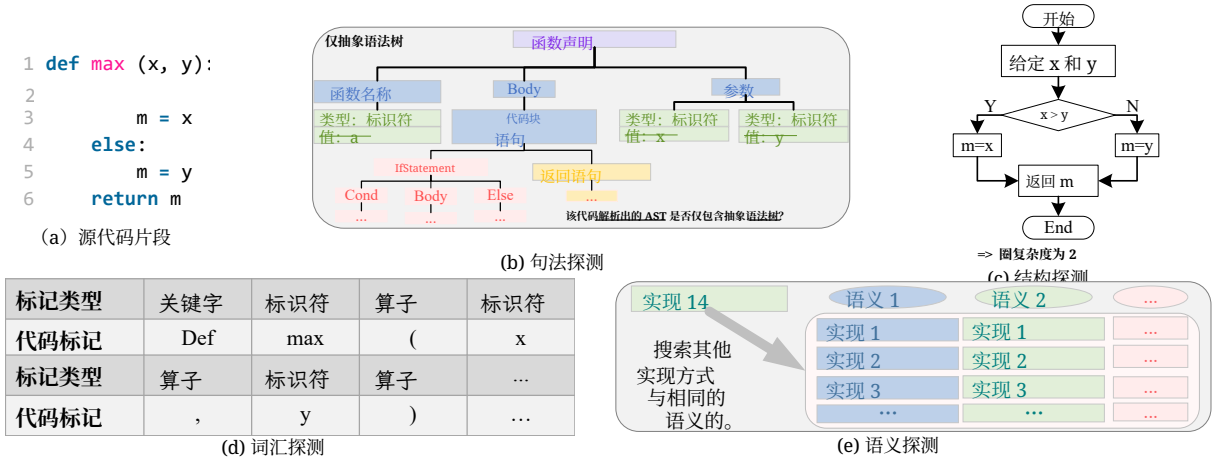
$$M = E - N + 2P \tag{3}$$

其中， E 和 N 分别表示图中的边数和节点数。 P 表示连通分量的数量。由于 CFG 是连通图，其值通常为 1。如图 1(c) 所示，CFG 有 7 个节点和 7 条边，因此该代码片段的圈复杂度为 $7-7+2=2$。我们将圈复杂度预测用作结构探测任务，以研究上下文表示对源代码结构属性的理解程度。结构探测的详细说明见在线附录 [51]。

3.2.2 探测流程。按照以往研究 [37, 52], ，为了探究预训练模型的逐层表示编码了哪些代码属性，我们训练了一个分类器，将这些逐层表示作为输入，以预测与某一代码属性相关联的探测任务。同时，我们学习所有层的上下文表示的线性组合，以研究各层的表示对理解这些代码属性的贡献程度。具体而言，对于预训练模型的逐层表示 H^0， H^1，……， H^L，我们将它们组合如下：

$$F = \sum_{l=1}^{L} \lambda^l H^l, \quad \lambda^l = \frac{\exp a_l}{\sum_{i=0}^{L} \exp a_i} \tag{4}$$

其中，层级权重 a_l 与探测分类器共同学习。一方面，我们比较组合表示 F 与随机初始化的表示的性能，以研究预训练上下文表示的效果如何



（a）源代码片段 (b) 句法探测 (c) 结构探测 (d) 词汇探测 (e) 语义探测

图 1：源代码和探测任务示例

表 1：下游任务概览，包括任务描述以及评估所使用的数据集、编程语言和指标。#Size 按顺序表示训练集、验证集和测试集的大小。对于指标，精确率、召回率、F1 值分别是 precision、recall 和 F1分数的简写。精确匹配和编辑相似度分别是完全匹配准确率和 Levenshtein编辑相似度的简写。

Task	描述	数据集名称	语言	#大小	指标
代码搜索	搜索语义相关的代码片段 针对给定的自然语言查询。	CodeSearchNet [23]	Python	251K/9.6K/1K	平均倒数排名、R@1
			Ruby	24.9K/1.4K/1.3K	R@5，R@10
克隆检测	检测两个代码片段是否相同 是功能等价。	BigCloneBench [49]	Java	901K/416K/416K	精确率、召回率、F1 值
代码摘要	生成简洁的自然语言 给定代码片段的描述。	CodeSearchNet [23]	Python	251K/9.6K/1K	BLEU、METEOR,
			Ruby	24.9K/1.4K/1.3K	Rouge-L、CIDEr
代码生成	为.....生成函数级代码片段 给定的自然语言描述。	CONCODE [24]	Java	100K/2K/2K	BLEU、精确匹配
代码补全	为给定内容预测下一行代码。 之前的代码上下文。	GitHub Java语料库 [5]	Java	12K/1.5K/1.5K	编辑相似度，精确匹配

编码源代码的属性。我们还比较预训练表示与微调表示之间的性能，以研究微调过程中代码属性发生了什么变化。另一方面，为了研究每一层的表示在编码代码属性方面的贡献程度，并探究预训练模型与微调模型之间的差异，我们针对每项探测任务给出了预训练模型和微调模型的逐层权重 a_l，并进一步分析了第 3.5.1 节中的实验结果。

3.3 表征相似性分析

按照先前的研究 [37]，我们随机抽取 N 个代码片段，并获取预训练模型和微调模型的逐层表示。然后，对于每一层，我们通过计算该层任意两个代码片段的表示向量之间的余弦相似度，得到第 l 层的距离矩阵 A^l（在第 2.3 节中介绍，大小为 $N \times N$）。表示向量通过对该层的上下文表示取平均得到。数学上，我们将预训练或微调代码模型的第 l 层上下文表示记为

第 k 个代码片段的上下文表示记为 $H_k{}^l$。距离矩阵 A^l 的计算方式如下：

$$A^l_{i,j} = \frac{v^l_i \cdot v^l_j}{\|v^l_i\| \|v^l_j\|}, \quad v^l_k = mean(H_k{}^l), \quad i,j,k \in [1,2,...,N] \tag{5}$$

接下来，对于第 l层，我们分别计算从预训练模型和微调模型中获得的两个距离矩阵之间的皮尔逊相关系数 ρ^l。具体而言，我们在五个多样化的下游任务上开展实验，包括代码搜索、克隆检测、代码摘要、代码生成和行级代码补全。这些任务的概览见表1。实验结果如第3.5.2节所示。

3.4 实验设置

在本研究中，我们分析了当前最先进的预训练代码模型 UniXcoder [17] 和 GraphCodeBERT [18]。二者均为 12 层、768 维的 Transformer，参数总量约为 120 MB。UniXcoder 是一种统一的预训练代码模型，可通过特殊指示标记用作编码器、解码器或编码器-解码器架构。GraphCodeBERT 考虑了数据流信息，并使用大量双模态数据（代码函数与自然

表2：随机、预训练和微调表示的探测任务性能。

探测任务	性能		
	随机	预训练的	微调后的
词汇探测	76.14	99.98	99.95
句法探测	64.70	95.60	95.10
语义探测	46.10	73.25	71.55
结构探测	34.80	89.80	62.20

语言注释配对）和单模态代码数据对大型模型进行预训练。我们在 UniXcoder 和 GraphCodeBERT 上开展实验，因为它们在许多代码智能任务中都取得了令人鼓舞的结果。

在探测实验中，我们通过 CodeSeachNet 和 POJ-104 [39] 构建评估数据集，如表 1 所示。词汇、句法和结构探测使用包含 Python 的 CodeSeachNet 数据集，语义探测则使用 POJ-104 [39]。遵循 UniXcoder/GraphCodeBERT 在代码搜索任务上的微调实验设置，我们在包含 Python 的 CodeSeachNet 数据集上微调预训练模型，并使用四种探测任务对预训练模型和微调后的模型进行探测。探测时，代码片段的最大长度设为 512。最大训练轮次和批次大小分别设为 30 和 32。我们采用学习率为 1e-4 的 Adam优化器，并在验证集上进行早停。我们使用随机种子 0、1、2 运行实验 3 次，并在论文中展示平均值。

对于表征相似性分析，按照先前的研究，将 [37], N 设置为 5,000。遵循 UniXcoder/GraphCodeBERT 的微调实验设置，我们在表 1 所示的五个下游任务上对其进行微调。

3.5 实验结果

在本节中，我们展示并分析上述两个研究问题的结果。由于篇幅限制，我们仅展示基于UniXcoder的Telly-K结果，并将基于GraphCodeBERT的Telly-K结果放在在线附录 [51]中。对UniXcoder成立的结论和发现通常也对GraphCodeBERT成立。

3.5.1 研究问题*1*：预训练模型的逐层表示中编码了哪些代码属性？我们使用与词汇、句法、语义和结构属性相关的四个探测任务，探索预训练模型的逐层表示中编码了哪些代码属性，以及每一层的表示在理解这些代码属性方面的贡献程度。同时，我们还比较了预训练和微调后的逐层表示在

同样的设置。探测任务上的性能如表2所示，逐层贡献如图2所示。

在表2中，词法、句法和结构探测的结果采用准确率进行衡量。语义探测的结果采用平均精度均值（MAP）进行衡量 [44]。有关准确率和MAP指标的更详细描述，请参阅在线附录 [51]。从表 2 中，我们可以发现：（1）与随机表示相比，预训练和微调后的表示能够更好地理解代码属性；（2）微调后，词法、句法和语义代码属性仍然得到了良好捕捉，而捕捉结构属性的能力则显著下降。第一个发现符合预期，因为预训练代码

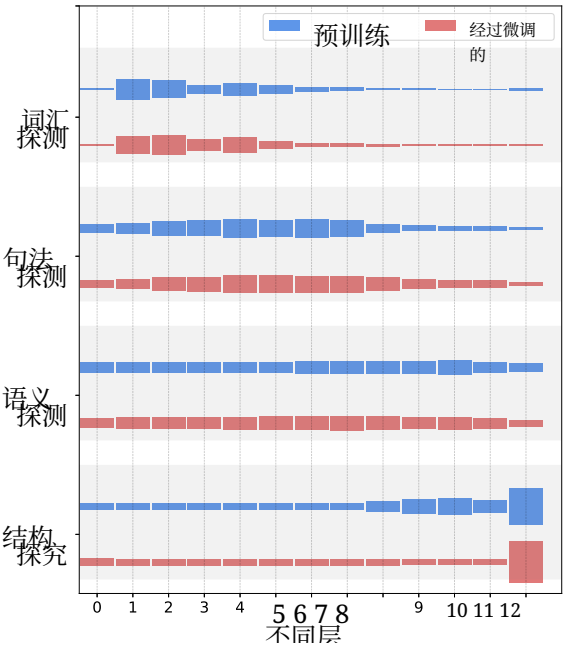


图2：预训练和微调代码模型在不同探测任务上的逐层贡献。

模型可以利用更大的数据集和模型规模，将基本的代码知识编码到其表示中。微调后，某些代码属性仍然得以保留。这可能与下游任务的特征有关，因为代码搜索主要依赖代码的词法、语法和语义信息，而不是结构信息。这也可能是梯度消失 [8] 的结果，因为低层编码的代码属性变化很小，而高层编码的代码属性变化明显。实际上，梯度消失对预训练代码模型的优化影响很小，因为模型采用的基本架构使用了残差连接 [20, 56]，可以有效避免梯度消失问题。

图2展示了预训练模型和微调模型的逐层贡献（λ^l 在公式4中）。从图2中，一方面，我们可以观察到，对于预训练模型或微调模型，源代码的词汇、句法和结构属性大多分别由低层、中间层和高层捕获，而语义属性几乎贯穿整个模型。我们进行显著性检验[1]，以考察贡献差异是否显著。实验结果表明，对于词汇探测，第1、2和4层的贡献显著高于其他层。对于句法探测，第4至第7层的贡献显著高于其他层。对于语义探测，不同层之间的贡献没有显著差异。对于结构探测，预训练模型和微调模型最后几层的贡献显著高于其他层。另一方面，我们发现微调过程保留了大部分代码属性。具体而言，低层捕获的基本代码属性】【。

[1]由于篇幅限制，结果被放在在线附录中 [51]

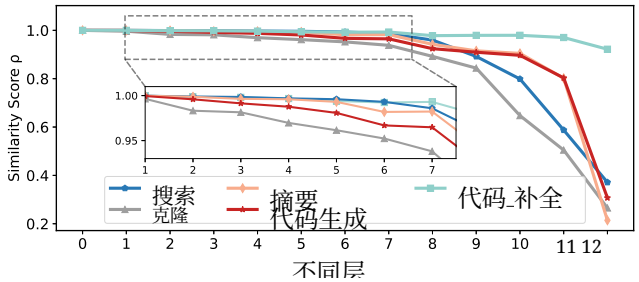


图3：五个下游任务中预训练模型与微调模型之间的相似度分数。

并且在微调过程中，中间层仍然得以保留。只有结构探测任务的性能发生了明显变化。

摘要。 对于预训练模型的逐层表示而言，源代码的词法、句法和结构属性主要分别由低层、中间层和高层捕获，而语义属性几乎贯穿整个模型。同时，低层和中间层捕获的基本代码属性在微调过程中仍得以保留。

3.5.2 研究问题*2*：微调过程中逐层表示会发生什么变化？我们还针对五个不同的下游任务开展了广泛的表示相似性分析（RSA）实验，以研究在不借助探测任务的情况下，预训练模型的逐层表示在微调过程中会发生什么变化。结果如图3所示。从所呈现的结果可以看出，底部9层的相关系数（图3中的相似度分数）均大于0.8。这意味着，对于这五个下游任务，底部9层在预训练模型与微调模型中的表示相似。除代码补全外，顶层的表示并不相似（$\rho \leq 0.5$）。这是因为预训练模型UniXcoder的顶层用于预测掩码词元，这与代码补全的实验设置相似。此外，我们发现，底部7层（$\rho \geq 0.9$）的表示具有很强的相关性，而底部5层（$\rho \geq 0.95$）的表示高度相似。

摘要。 在五个下游任务中，预训练模型与微调模型的底部九层表示相似。微调过程中，只有顶部两层的表示发生了显著变化。

4 预训练代码模型的高效微调

4.1 研究问题

研究问题*3*：是否存在微调的高效替代方案？基于实验研究的结果，我们探索了微调预训练代码模型的更高效替代方案。我们的主要动机是冻结那些在下游任务微调过程中变化很小的层的预训练参数。我们提出了**Telly-*K***，意为通过层冻结高效微调预训练代码模型。Telly-*K* 表示冻结底部 *K* 层的预训练参数，而不同的*K* 表示我们方法的不同变体。第 0 层是 the 嵌入层，并且我们研究的预训练代码模型的最大层数为 12。如果将 K 设置为 12，那么 Telly-12 将冻结所有参数，模型也将退化为原始预训练模型。因此，为了确保实验的全面性，我们将 K 从 0 变化到 11，并针对这 12 个模型变体在五个下游任务上开展了广泛的实验。接下来，我们将介绍实验设置、结果和分析。

4.2 实验设置

我们的实验涉及五种不同的下游任务，包括代码搜索、代码检测、代码摘要、代码生成和行级代码补全。表1概述了这些任务。*代码搜索* 使用广泛采用的 CodeSeachNet 数据集进行评估，涵盖 Python 和 Ruby；性能通过平均倒数排名（MRR）和前k召回率（R@k，k=1,5,10） [15, 18, 23]衡量。对于*克隆检测*，我们在常用的 BigCloneBench 数据集上进行实验，并采用精确率（P）、召回率（R）和 F1分数（F1）作为评估指标 [17, 18]。对于*代码摘要*，我们在 CodeSearchNet 数据集的 Python 和 Ruby 数据上对预训练模型 [17, 18] 进行微调。评估指标包括句子级平滑BLEU [41], 、METEOR [6], 、Rouge-L [32] 和 CIDEr [57]。*代码生成* 使用广泛采用的 CONCODE 数据集进行评估，性能通过句子级平滑BLEU [41] 和完全匹配准确率（EM）衡量。对于*行级代码补全*，我们在 CodeXGLUE 中的大型数据集（名为 GitHub Java语料库）上开展实验。与以往工作 [34], 相同，性能通过 EM 和 Levenshtein编辑相似度（编辑相似度） [50]衡量。行级代码补全的微调遵循 CodeXGLUE 中的实验设置 [34]。其他任务则遵循以往研究 [17, 18]中的设置。我们采用 Adam优化器，最大训练轮次为 30，并在验证集上进行早停。我们使用随机种子0、1、2运行实验3次，并在论文中报告平均值。所有实验均在配备 Tesla A100 GPU 的机器上进行。详细的实验设置见复现包 [51]中的在线附录。

4.3 实验结果

我们使用 Telly-*K*进行实验，在五个下游任务上微调预训练代码模型时冻结其底部的*K* 层。我们首先逐一呈现并分析各任务的实验结果，包括训练参数、训练耗时和性能，然后总结下游任务的发现。受篇幅限制，我们仅呈现基于 UniXcoder 的 Telly-*K* 结果，并将基于 GraphCodeBERT 的 Telly-*K* 结果放在在线附录 [51]中。对于 UniXcoder 得出的结论和发现通常也适用于 GraphCodeBERT。

4.3.1 代码搜索。我们研究了12 个Telly-*K* 变体在代码搜索上的性能。结果如表 3 所示。我们仅展示 Python数据集上的结果，并将Ruby上的结果（与 Python 上的结果类似）放在在线附录 [51] 中。

在表 3 中，我们展示了基础模型和12 个模型变体的训练参数数量、训练时间和性能。基础模型对所有预训练参数进行微调，而不同的变体则会冻结部分参数。我们同时报告了每个训练轮次的训练时间以及模型收敛所需的时间。

表3：代码搜索实验结果。#Params 是训练参数数量的缩写。M 是百万的缩写。与基础模型相比的变化比例显示在括号中。

模型	Note	#Params	训练时间		性能			
			每个训练轮次	收敛	MRR	R@1	R@5	R@10
Base	全参数微调	125.93M	17m14s	2小时35分06秒	0.720	0.612	0.838	0.889
Telly-0	冻结底部 0 层	85.0M(↓32%)	15分49秒(↓8%)	2小时06分32秒（↓18%）	0.727(↑1%)	0.630(↑3%)	0.846(↑1%)	0.895(↑1%)
Telly-1	冻结底部 1 层	78.0M(↓38%)	14分53秒（↓14%）	1小时59分04秒（↓23%）	0.727(↑1%)	0.629(↑3%)	0.848（↑1%）	0.895(↑1%)
Telly-2	冻结底部 2 层	70.9M(↓44%)	14m06s(↓18%)	1小时52分48秒（↓27%）	0.727(↑1%)	0.630(↑3%)	0.849(↑1%)	0.896(↑1%)
Telly-3	冻结底部 3 层	63.8M(↓49%)	13m12s(↓23%)	0小时39分钟36秒（↓74%）	0.724(↑1%)	0.626(↑2%)	0.842(↑0%)	0.895(↑1%)
Telly-4	冻结底部 4 层	56.7M(↓55%)	12m12s(↓29%)	0小时48分48秒(↓69%)	0.724(↑1%)	0.626(↑2%)	0.844(↑1%)	0.896(↑1%)
Telly-5	冻结底部 5 层	49.6M(↓61%)	11分42秒（↓32%）	0小时35分06秒(↓77%)	0.726(↑1%)	0.628(↑3%)	0.848(↑1%)	0.896(↑1%)
Telly-6	冻结底部 6 层	42.5M(↓66%)	10m50s(↓37%)	0小时32分钟30秒（↓79%）	0.727(↑1%)	0.629(↑3%)	0.848（↑1%）	0.896（↑1%）
Telly-7	冻结底部 7 层	35.4M(↓72%)	9m57s(↓42%)	0小时29分51秒(↓81%)	0.725(↑1%)	0.627(↑2%)	0.847(↑1%)	0.896(↑1%)
Telly-8	冻结底部 8 层	28.4M(↓77%)	9m04s(↓47%)	0小时36分16秒（↓77%）	0.723(↑0%)	0.625(↑2%)	0.844（↑1%）	0.894(↑1%)
Telly-9	冻结底部 9 层	21.3M(↓83%)	8分14秒(↓52%)	0h24m42s(↓84%)	0.718(↓0%)	0.620(↑1%)	0.838（0%）	0.888（↓0%）
Telly-10	冻结底部10层	14.2M(↓89%)	7分10秒(↓58%)	0小时21分30秒(↓86%)	0.710(↓1%)	0.612(↓0%)	0.829(↓1%)	0.882（↓1%）
Telly-11	冻结底部 11 层	7.1M(↓94%)	6分21秒（↓63%）	0小时19分03秒(↓88%)	0.694(↓4%)	0.593(↓3%)	0.815(↓3%)	0.871(↓2%)

表 4：克隆检测和代码摘要的实验结果。#Params 是训练参数数量的缩写。M 是百万的缩写。与基础模型相比的变化比例显示在括号中。

模型	克隆检测						代码摘要				
	#参数	训练时间		性能			#参数	训练时间		性能	
		每个训练轮次	收敛	召回率	精确率	F1分数		每个训练轮次	收敛	BLEU	METEOR
Base	127.1M	20分21秒	1小时41分45秒	0.95	0.95	0.95	125.93M	22分44秒	1小时53分40秒	19.15	17.26
Telly-0	86.2M(↓32%)	19m29s(↓4%)	1h17m56s(↓23%)	0.96(↑1%)	0.94(↓1%)	0.95(0%)	85.0M(↓32%)	21分30秒(↓5%)	1小时47分30秒(↓5%)	19.19(↑0%)	17.32(↑0%)
Telly-1	79.2M（↓38%）	18m36s（↓9%）	1h14m24s（↓27%）	0.95（0%）	0.95（0%）	0.95（0%）	78.0M(↓38%)	20m15s(↓11%)	1h41m15s(↓11%)	19.21(↑0%)	17.33(↑0%)
Telly-2	72.1M(↓43%)	17m34s(↓14%)	1h10m16s(↓31%)	0.95(0%)	0.95(0%)	0.95(0%)	70.9M(↓44%)	19m11s(↓16%)	1h35m55s(↓16%)	19.17(↑0%)	17.30(↑0%)
Telly-3	65.0M(↓49%)	15m20s(↓25%)	1h01m20s(↓40%)	0.95(0%)	0.95(0%)	0.95(0%)	63.8M(↓49%)	17m59s(↓21%)	0h53m57s(↓53%)	19.16(↑0%)	17.26(0%)
Telly-4	57.9M(↓54%)	14m28s(↓29%)	0h57m52s(↓43%)	0.94(↓1%)	0.96(↑1%)	0.95(0%)	56.7M(↓55%)	17m07s(↓25%)	0h51m21s(↓55%)	19.13(↓0%)	17.26(0%)
Telly-5	50.8M(↓60%)	13m25s(↓34%)	1h07m05s(↓34%)	0.96(↑1%)	0.94(↓1%)	0.95(0%)	49.6M(↓61%)	16m18s(↓28%)	0h48m54s(↓57%)	19.18(↑0%)	17.26(0%)
Telly-6	43.7M(↓66%)	12m35s(↓38%)	0h50m20s(↓51%)	0.96(↑1%)	0.95(0%)	0.95(0%)	42.5M(↓66%)	15m10s(↓33%)	0h45m30s(↓60%)	19.36(↑1%)	17.35(↑1%)
Telly-7	36.6M(↓71%)	11m44s(↓42%)	0h58m40s(↓42%)	0.95(0%)	0.93(↓2%)	0.94(↓1%)	35.4M(↓72%)	14m08s(↓38%)	0h28m16s(↓75%)	19.37(↑1%)	17.28(↑0%)
Telly-8	29.5M(↓77%)	10m41s(↓48%)	0h53m25s(↓48%)	0.95(0%)	0.94(↓1%)	0.95(0%)	28.4M(↓77%)	12m59s(↓43%)	0h25m58s(↓77%)	19.34(↑1%)	17.26(0%)
Telly-9	22.4M(↓82%)	9m55s(↓51%)	0h29m45s(↓71%)	0.95(0%)	0.92(↓3%)	0.93(↓2%)	21.3M(↓83%)	11m28s(↓50%)	0h22m56s(↓80%)	19.18(↑0%)	17.22(↓0%)
Telly-10	15.4M(↓88%)	8m51s(↓57%)	0h35m24s(↓65%)	0.97(↑2%)	0.92(↓3%)	0.94(↓1%)	14.2M(↓89%)	10m19s(↓55%)	0h10m19s(↓91%)	19.11(↓0%)	17.18(↓0%)
Telly-11	8.3M(↓93%)	8分00秒(↓61%)	0小时32分00秒(↓69%)	0.96(↑1%)	0.92(↓3%)	0.94(↓1%)	7.1M(↓94%)	09m14s(↓59%)	0h09m14s(↓92%)	19.10(↓0%)	17.20(↓0%)

不同变体模型相对于基础模型的变化比例显示在括号中。从表 3 的结果可以看出：

• 对于所有变体模型，与基础模型相比，训练耗时（尤其是收敛耗时）和训练参数均显著减少，而四项指标的性能变化不大。特别是，对于冻结底部11层的Telly–11，模型收敛耗时和训练参数分别减少了 88% 和 94%，而性能仅下降了约 3%。

• 对于 Telly-K ($0 \le K \le 8$)，它们将训练参数减少32%至77%，相应地节省约18%至81%的训练时间，同时在所有指标上的性能提升为0%至3%。

• 冻结底部 9 层后，训练参数减少了 83%，训练时间相应节省了 84%，而性能仅有轻微变化。例如，与基础模型相比，Telly-9 的 MRR 和 R@10 值】【。

下降不到 1%，R@1 提高约 1%，R@5 保持不变。

• 对于 Telly-K ($K \ge 10$)，在四项指标上性能均持续下降。然而，即使冻结底部 11 层，性能也不会显著下降，同时训练参数和相应的训练时间大幅减少。

摘要。 在代码搜索任务中，与基础模型相比，Telly-K的性能在 $0 \le K \le 8$上有所提升，在 $K = 9$上略有变化，在 $K \ge 10$ 上略有下降。

4.3.2 克隆检测。我们在克隆检测任务上使用不同的 Telly-K 变体进行实验，结果如表 4左半部分所示。由于预训练代码模型采用2 层MLP 作为分类器，以判断两段代码是否为克隆代码，因此总参数量约为127.1 百万。性能通过精确率（P）、召回率（R）和 F1分数（F1）进行评估，它们的范围为 $[0, 1]$。从表 4 中，我们可以发现：

• 对于所有变体模型，与基础模型相比，训练耗时和训练参数均显著减少，同时性能没有显著变化。尤其是，冻结底部11层时，收敛耗时和训练参数分别减少了 88% 和 94%，而性能仅变化了 1-3%。

• 对于 Telly-K ($0 \le K \le 8$)，训练参数减少了32%至77%，相应地节省了约23%至51%的训练耗时。此外，性能方面，F1分数总体稳定，而精确率和召回率略有变化。

• 当（$K \ge 9$）时，不同变体的精确率和F1分数均持续下降，但召回率有所提升。然而，即使对于 Telly-11，所有评估指标仍具有较高的得分（大于 0.9），同时训练参数和相应的耗时也大幅减少。

摘要。 在克隆检测任务中，与基础模型相比，Telly-K 对于 $0 \le K \le 8$ 的性能总体稳定，但对于 $K \ge 9$ 略有变化。

4.3.3 代码摘要。 我们使用不同的 Telly-K on 进行代码摘要实验，结果如表 4 右半部分所示。由于篇幅限制，Python 上的 Rouge-L 和 CIDEr 结果以及 Ruby 上的所有实验结果均放在在线附录 [51] 中。报告的指标包括 BLEU 和 METEOR，范围为 $[0, 100]$。从表 4 中，我们可以发现：

• 对于所有变体模型，与基础模型相比，训练耗时和参数量都显著减少，而所有指标的性能变化不大。

• 对于 Telly-K ($0 \le K \le 5$)，训练参数减少了 32% 至 61%，相应地节省了约 5% 至 57% 的训练时间，同时性能保持稳定。具体而言，所有评估指标的性能变化均小于 1%。

• 对于 Telly-K ($6 \le K \le 9$)，训练参数减少了 66% 至 83%，相应地训练时间节省了 60% 至 80%，同时性能通常仅略有提升。

• 当 Telly-K ($K \ge 10$)时，从四项指标来看，性能略有下降。然而，即使冻结底部 11 层，性能也不会显著下降，同时训练参数和相应的耗时都大幅减少。

摘要。在代码摘要任务上，Telly-K 的性能在（$0 \le K \le 5$）方面保持稳定，对于 $6 \le K \le 9$略有提升，但与基础模型相比，在 $K \ge 10$ 时略有下降。

4.3.4 代码生成。我们针对不同的 Telly-K 模型进行了代码生成实验，结果见表5左半部分。评估指标 BLEU 和精确匹配（EM）的取值范围为 $[0, 100]$。从表5可以发现：

• 对于所有变体模型，与基础模型相比，训练耗时和参数量都显著减少，而除 Telly-K($K \ge 8$) 外，各项指标的性能变化不大。

• 对于 Telly-K ($0 \le K \le 5$)，它们将训练参数减少了 32% 至 61%，相应地节省了约 10% 至 55% 的训练时间，性能略有提升。具体而言，BLEU 得分提高了 0% 至 2%，精确匹配（EM）得分提高了 3% 至 10%。

• 对于 Telly-K ($6 \le K \le 9$)，训练参数减少了 66% 至 83%，相应地节省了 57% 至 65% 的训练时间。这些变体在 BLEU 指标下的性能显著下降，但在精确匹配（EM）指标下通常有所提升。这是因为 BLEU 综合计算一个样本中生成的代码片段与标准答案之间的 n 元组精确率（n=1,2,3,4），而如果两者完全相同，EM 值为 1，否则为 0。然而，对于整个数据集，只有约 18% 的代码片段能够与标准答案完全一致。其余 82% 的样本也会影响 BLEU 得分的最终结果。因此，BLEU 和 EM 的性能变化表现不同。

• 当Telly-K ($K \ge 10$)时，训练参数和相应的时间成本大幅减少，且各变体的性能在两个指标上也显著下降。

摘要。 在代码生成任务中，与基础模型相比，Telly-K 的性能对于 Telly-K ($0 \le K \le 5$)略有提升，对于 $6 \le K \le 9$则有明显变化，而在 $K \ge 10$ 时显著下降。

4.3.5 行级代码补全。我们使用所有 Telly-K 开展了行级代码补全实验，结果如表5右半部分所示。评估指标包括编辑相似度和精确匹配，取值范围为 $[0, 100]$。从表5中可以发现：

• 对于所有变体模型，与基础模型相比，训练时间耗时和参数量都大幅减少，而性能没有显著变化，Telly-11 除外。

• 对于 Telly-K ($0 \le K \le 7$)，它们将训练参数减少了 32%至72%，相应地节省了约13% 至 75%的训练时间。此外，Telly-K ($0 \le K \le 3$)的性能略有提升，而 Telly-K ($4 \le K \le 7$)的性能则略有下降。

• 对于 Telly-K（$K \ge 8$），训练参数和相应的耗时都大幅减少。变体模型在两个指标上的性能总体有所下降。尤其是当 $K = 11$ 时，性能显著下降。因此，需要对最后几层的预训练参数进行微调，以学习预测下一行代码。

摘要。在代码补全任务上，与基础模型相比，Telly-K 的性能略有提升（$0 \le K \le 10$，而当 $K = 11$ 时则显著下降。

4.3.6 各下游任务的发现。在逐项分析实验结果后，我们总结了各项任务中的总体发现如下。

• 对于所有 Telly-K，与基础模型相比，训练耗时（尤其是收敛耗时）和训练参数都显著减少。除 Telly-10 和 Telly-11 在代码生成任务上的表现，以及 Telly-11 在代码补全任务上的表现外，性能变化不大。

• 当 $0 \le K \le 5$时，Telly-K 通常可将训练参数减少30%至65%，相应地节省约10%至70%的训练时间，同时性能通常有所提升且幅度各异

表5：代码生成和行级代码补全的实验结果。#Params 是训练参数数量的缩写。M 是百万的缩写。括号中显示了相对于基础模型的变化比例。

模型	代码生成					行级代码补全				
	#参数	训练时间		性能		#参数	训练时间		性能	
		每个训练轮次	收敛	BLEU	EM		每个训练轮次	收敛	编辑相似度	EM
Base	125.93M	12分25秒	4小时08分20秒	33.82	17.4	125.93M	04分12秒	0小时37分48秒	51.92	20.40
Telly-0	85.0M(↓32%)	11分43秒(↓6%)	3小时42分37秒(↓10%)	33.88(↑0%)	18.1(↑4%)	85.0M(↓32%)	03分59秒(↓5%)	0小时31分52秒(↓16%)	52.58(↑1%)	21.07(↑3%)
Telly-1	78.0M(↓38%)	11m07s(↓10%)	3h20m06s(↓19%)	34.43(↑2%)	17.9(↑3%)	78.0M(↓38%)	03m47s(↓10%)	0h18m55s(↓50%)	52.35(↑1%)	20.87(↑2%)
Telly-2	70.9M(↓44%)	10m32s(↓15%)	2h06m24s(↓49%)	33.85(↑0%)	19.1(↑10%)	70.9M(↓44%)	03m38s(↓13%)	0h32m42s(↓13%)	52.31(↑1%)	20.93(↑3%)
Telly-3	63.8M(↓49%)	09m52s(↓21%)	2h08m16s(↓48%)	34.24(↑1%)	19.0(↑9%)	63.8M（↓49%）	03m29s（↓17%）	0h20m54s（↓45%）	51.81（↓0%）	20.73（↑2%）
Telly-4	56.7M(↓55%)	09m15s(↓26%)	1h51m00s(↓55%)	34.02(↑1%)	18.6(↑7%)	56.7M(↓55%)	03m18s(↓21%)	0h26m24s(↓30%)	51.62(↓1%)	20.27(↓1%)
Telly-5	49.6M(↓61%)	08分44秒(↓30%)	1小时53分32秒(↓54%)	34.36(↑2%)	18.6(↑7%)	49.6M(↓61%)	03分07秒(↓26%)	0小时09分21秒(↓75%)	51.66(↓0%)	20.40(0%)
Telly-6	42.5M(↓66%)	08m13s(↓34%)	1h46m49s(↓57%)	32.90(↓3%)	18.1(↑4%)	42.5M(↓66%)	02分57秒(↓30%)	0小时11分48秒(↓69%)	51.36(↓1%)	20.27(↓1%)
Telly-7	35.4M(↓72%)	07m41s(↓38%)	1h39m53s(↓60%)	32.92(↓3%)	18.0(↑3%)	35.4M(↓72%)	02m58s(↓29%)	0h14m50s(↓63%)	51.32(↓1%)	20.40(0%)
Telly-8	28.4M(↓77%)	07m08s(↓43%)	1h32m44s(↓63%)	32.12(↓5%)	17.4(0%)	28.4M(↓77%)	02m59s(↓29%)	0h14m55s(↓65%)	50.95(↓2%)	20.27(↓1%)
Telly-9	21.3M(↓83%)	06m42s(↓46%)	1h27m06s(↓65%)	31.33(↓7%)	18.1(↑4%)	21.3M(↓83%)	02m58s(↓29%)	0h14m50s(↓67%)	51.13(↓2%)	20.73(↑2%)
Telly-10	14.2M（↓89%）	06m03s（↓51%）	1h12m36s（↓71%）	30.43（↓10%）	16.6（↓5%）	14.2M（↓89%）	02m57s（↓30%）	0h11m48s（↓76%）	50.70（↓2%）	20.53（↑1%）
Telly-11	7.1M(↓94%)	05m22s(↓57%)	1h09m46s(↓72%)	27.71(↓18%)	14.3(↓18%)	7.1M(↓94%)	02m06s(↓50%)	0h06m18s(↓83%)	49.49(↓5%)	19.67(↓4%)

在各种下游任务中，相关指标提升了 1% 至 4%。因此，在降低资源消耗和提升性能方面，Telly-5 通常是这些变体中的最佳选择。

• 当 $6 \le K \le 9$时，Telly-K 通常可将训练参数减少65%至80%，相应地节省约50%至80%的训练时间，而在各种下游任务中的性能通常仅发生轻微变化。具体而言，代码搜索和代码生成的性能变化幅度为1%至5%，克隆检测和代码摘要的变化幅度为0%至1%，代码补全的变化幅度为0%至3%。因此，在以较小性能变化降低资源消耗的各个变体中，Telly-9通常是最佳选择。

• 当 $K \ge 10$时，训练参数和耗时都大幅减少。Telly-K 的性能有所下降，但并不显著，代码生成和代码补全除外。

尽管在上述实验中冻结了部分层，模型的性能仍有所提升。一种可能的解释是，参数数量的减少缓解了过拟合，使在训练集上微调的参数能够更好地泛化到测试集。然而，这一现象背后的确切机制仍未得到充分理解，需要进一步研究，例如考察训练集和测试集分布不同的更多数据集，以验证我们的推测。

摘要。 对于 Telly-K，在各种下游任务中，与基础模型相比，训练参数和耗时都大幅减少。总体而言，与基础模型相比，Telly-K 1 4 $0 \le K \le 5$ $6 \le K \le$ 的性能 9 $K \ge 10$ 提升了 % 至 %（），略有变化（），而在 时则明显下降。

5 讨论与有效性威胁

5.1 降低微调耗时的重要性及 Telly-K 的优势。

降低微调成本非常重要，尤其是耗时方面的成本，因为（1）预训练与微调范式在许多软件工程任务中日益得到采用 [2, 15, 17, 34]。降低耗时很重要，因为这通常需要很长时间对模型进行微调，尤其是在更大的数据集上进行微调。减少微调耗时还可以提高预训练代码模型的开发效率，尤其是在开发者需要赶上产品发布期限时，从而节省成本并减少意外损失。(2) GPU 通常是昂贵的计算资源。通过减少耗时，我们还可以节省 GPU 资源。此外，训练基于深度学习的模型，包括对预训练模型进行微调，每年可能会排放超过 60 万吨的 CO2 [48]。通过减少耗时，可以减少碳排放。

事实上，人们已提出许多方法 [22, 26, 29, 35, 59, 60] 来节省微调耗时。其中，知识蒸馏技术 [29, 35, 60] 颇具前景，也广受欢迎。具体而言，这些方法提出了不同的方案，将大型预训练模型压缩为较小的模型，再对较小的模型进行微调，以执行下游任务。然而，这些技术需要仔细设计和调整小模型的架构及损失函数，以从大型模型中蒸馏知识。因此，通常而言，蒸馏过程需要更多的人工设计，也更加费力。我们提出的方法 Telly-K 简单易行，并且可以通过层冻结直接降低训练成本。

5.2 如何助力未来更好地进行微调

本文并未针对不同任务总结出一种“放之四海而皆准”的方法。不过，我们对研究问题2和研究问题3的研究表明，冻结底部 7 层的 Telly-7 在效率与性能之间实现了良好的权衡。具体而言，(1) 从图3可以看出，预训练模型与微调模型底部 7 层的表示非常相似；(2) 从表 3、4 和 5 可以看出，Telly–7 显著减少了训练时间和参数量，同时在所有下游任务上保持了相近的性能。因此，我们建议在实践中使用 Telly–7。未来，我们计划研究自动化层冻结策略，以更高效地利用逐层表示。例如，我们计划设计算法，自动从不同层中选择特征并将其聚合，以执行不同的下游任务。

5.3 有效性威胁

我们已识别出本研究存在以下威胁：

编程语言。我们在五种编程语言（Python、Java、Ruby、C和 C++）上进行了实验。尽管原则上，我们所研究的模型并非专门针对某些语言而设计，但模型在不同编程语言上的表现可能有所不同。因此，需要开展更多实验来验证我们研究结果和结论的普适性。未来，我们将把研究扩展到更多编程语言。

评估数据集。我们在广泛使用的数据集上开展实验。此外，每项下游任务还有其他数据集。这些数据集在构建方法和语料库规模等方面有所不同。模型在不同数据集上的表现可能不同。因此，我们将使用更多数据集开展实验，以确认研究发现和结论的普遍性。

评估指标。本研究尽可能使用多种常用指标来评估模型性能。然而，这些指标可能存在固有局限。例如，BLEU 和 METEOR 是基于文本相似度的指标，无法衡量两个句子的语义相似度。未来，我们将采用更多指标并进行人工评估，以确认本研究的发现和结论。

预训练代码模型。受计算资源限制，本研究重点关注最先进的预训练代码模型 UniX-coder 和 GraphCodeBERT。其他预训练代码模型，如 CodeGPT [34] 和CodeT5 [63]，尚待研究。

6 相关工作

6.1 探索预训练模型

在自然语言处理领域，许多研究 [1,10, 37, 52, 53]探讨了预训练语言模型如何理解自然语言，以及对其进行微调时会发生什么。它们通常分为两类。第一类采用探测技术，研究预训练语言模型所捕获的语言属性 [52, 53]。第二类是表征相似性分析 [1, 10, 37]。这是一种与任务无关的分析方法，用于衡量两个不同表征空间之间的相似性。然而，在软件工程领域，很少有研究探讨预训练代码模型 在微调过程中会发生什么。大多数相关研究 [15, 21, 30, 54, 58] 旨在理解预训练代码模型了解哪些源代码信息。例如，Wan等人 [58] 进行结构分析，以证明预训练模型能够感知句法结构。López 等人 [21] 从预训练语言模型的隐藏表征中恢复抽象语法树。Karmakar 等人 [30] 还提出了一些新的探测任务，以研究预训练代码模型了解哪些代码信息。受编译过程和静态分析的启发，我们首先提出了涉及词法、句法、语义和结构代码属性的四项探测任务。然后，我们研究逐层预训练表征中编码了哪些代码属性，以及这些表征在微调过程中会发生什么。

6.2 加速微调过程

已有许多研究致力于加速微调过程 [22, 26, 29, 35, 59, 60]。这些研究大致可分为两类。第一类是利用知识蒸馏技术压缩大规模预训练语言模型 [29, 35, 60]。例如，Jiao 等人 [29] 提出了 TinyBERT，通过知识蒸馏 BERT，仅使用约 28%的参数来实现自然语言理解。第二种是基于适配器的微调方法 [22, 59],，其中适配器是添加在预训练模型各层之间的新型可训练模块。例如，Houlsby 等人l [22] 设计了一些适配器，与完整模型相比，微调所需参数减少了两个数量级，并且通过对预训练模型进行全参数微调，实现了相近的性能。此外，还有一些研究通过层冻结，从头开始进行高效的神经网络训练 [28, 62]。例如，Wang 等人 [62]利用参考模型中的知识，准确评估各层训练的可塑性，冻结已经收敛的层，并解冻被冻结的层以继续训练。我们的研究可能会激励研究人员提出更加高效的微调方法。

7 结论

在本文中，我们首先进行了广泛的实验研究，探索微调过程中逐层代码知识和预训练表示会发生什么变化。随后，我们基于上述发现提出了高效微调大型预训练代码模型的替代方案。实验研究表明，源代码的词汇、句法和结构属性分别主要由低层、中间层和高层捕获，而语义属性则贯穿整个模型。低层和中间层捕获的基本代码属性在微调过程中仍得以保留。此外，我们发现，对于各种下游任务，微调过程中变化最大的只有顶端两层的表示。基于上述发现，我们提出了 Telly-K，通过选择性层冻结高效微调预训练代码模型。针对五种不同下游任务开展的大量实验表明，训练参数量和耗时均可大幅降低，同时性能相近甚至更优。

此外，我们的实验研究还展示了许多关于预训练代码模型高效微调的有用发现和有前景的方向。例如，我们发现预训练代码模型将句法属性编码到中间层中。因此，当下游任务具有句法感知能力（如代码生成）时，我们可以设计一些模块，利用中间层的上下文表示。未来，我们将继续探索各种有前景的方向，并提出更加高效的微调方法。

包含源代码、数据集和在线附录的复现包可访问：https://github.com/DeepSoftwareAnalytics/Telly。

致谢

我们感谢审稿人对本工作的宝贵意见。本研究得到国家重点研发计划（编号：2017YFA0700800）和中央高校基本科研业务费（项目编号：xtr072022001）的支持。

参考文献

[1] Samira Abnar、Lisa Beinborn、Rochelle Choenni 和 Willem H.Zuidema。2019。《黑箱遇上黑箱：表征相似性与稳定性分析》】【。

神经语言模型与大脑。在 BlackboxNLP@ACL。计算语言学协会，191–203。[2] Wasi Uddin 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 NAACL-HLT。计算语言学协会，2655–2668。[3] Alfred V Aho、Monica S Lam、Ravi Sethi 和 Jeffrey D Ullman。2007年。《编译原理、技术与工具》。培生印度。[4] Haldun Akoglu。2018年。相关系数使用指南。《土耳其急诊医学杂志》18，3（2018），91–93。[5] Miltiadis Allamanis 和 Charles Sutton。2013年。使用语言建模大规模挖掘源代码仓库。载于 2013年第十届软件仓库挖掘工作会议（MSR）。IEEE，207–216。[6] Satanjeev Banerjee 和 Alon Lavie。2005年。METEOR：一种自动机器翻译评估指标，与人工判断的相关性更高。载于 IEEvalua- tion @ACL。[7] Hangbo Bao、Li Dong、Songhao Piao 和 Furu Wei。2022年。BEiT：图像变换器的 BERT 预训练。载于 ICLR。OpenReview.net。[8] Yoshua Bengio、Patrice Simard 和 Paolo Frasconi。1994年。使用梯度下降学习长期依赖关系很困难。《IEEE神经网络汇刊》5，2（1994），157–166。[9] Saikat Chakraborty、Toufique Ahmed、Yangruibo Ding、Premkumar Devanbu 和 Baishakhi Ray。2022年。NatGen：通过“自然化”源代码进行生成式预训练。（2022）。[10] Grzegorz Chrupala 和 Afra Alishahi。2019年。语言的神经表征与符号表征之间的相关性。载于 ACL（1）。计算语言学协会，2952–2962。[11] Yingnong Dang、Song Ge、Ray Huang 和 Dongmei Zhang。2011年。微软的代码克隆检测经验。载于第五届软件克隆国际研讨会论文集。63–64。[12] Jacob Devlin、Ming-Wei Chang、Kenton Lee 和 Kristina Toutanova。2019年。BERT：用于语言理解的深度双向变换器预训练。载于 NAACL-HLT（1）。计算语言学协会，4171–4186。[13] Yangruibo Ding、Luca Buratti、Saurabh Pujar、Alessandro Morari、Baishakhi Ray 和 Saikat Chakraborty。2022年。通过程序对比学习源代码的（不）相似性。载于 ACL（1）。计算语言学协会，6300–6312。[14] Lun Du、Xiaozhou Shi、Yanlin Wang、Ensheng Shi、Shi Han 和 Dongmei Zhang。2021年。单个模型够用吗？MuCoS：一种用于语义代码搜索的多模型集成学习方法。载于第30届ACM信息与知识管理国际会议论文集。2994–2998。[15] Zhangyin Feng、Daya Guo、Duyu Tang、Nan Duan、Xiaocheng Feng、Ming Gong、Linjun Shou、Bing Qin、Ting Liu、Daxin Jiang 和 Ming Zhou。2020年。CodeBERT：面向编程语言与自然语言的预训练模型。载于 EMNLP（Findings）。[16] Xiaodong Gu、Hongyu Zhang 和 Sunghun Kim。2018年。深度代码搜索。载于 ICSE。ACM，933–944。[17] Daya Guo、Shuai Lu、Nan Duan、Yanlin Wang、Ming Zhou 和 Jian Yin。2022年。UniXcoder：用于代码表示的统一跨模态预训练。载于 ACL（1）。计算语言学协会，7212–7225。[18] 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、Jian Yin、Daxin Jiang 和 Ming Zhou。2021年。GraphCodeBERT：利用数据流预训练代码表示。载于第九届国际学习表征会议（ICLR 2021），线上会议，奥地利，2021年5月3日至7日。OpenReview.net。https://openreview.net/forum?id=jLoC4ez43PZ [19] Kaiming He、Xinlei Chen、Saining Xie、Yanghao Li、Piotr Dollár 和 Ross B. Girshick。2022年。掩码自编码器是可扩展的视觉学习器。载于 CVPR。IEEE，15979–15988。[20] Kaiming He、Xiangyu Zhang、Shaoqing Ren 和 Jian Sun。2016年。用于图像识别的深度残差学习。载于 IEEE计算机视觉与模式识别会议论文集。770–778。[21] José Antonio Hernández López、Martin Weyssow、Jesús Sánchez Cuadrado 和 Houari Sahraoui。2022年。AST-Probe：从预训练语言模型的隐藏表征中恢复抽象语法树。载于第37届IEEE/ACM自动化软件工程国际会议。1–11。[22] Neil Houlsby、Andrei Giurgiu、Stanislaw Jastrzebski、Bruna Morrone、Quentin de Laroussilhe、Andrea Gesmundo、Mona Attariyan 和 Sylvain Gelly。2019年。面向自然语言处理的参数高效迁移学习。载于 ICML（《机器学习研究论文集》，第97卷）。PMLR，2790–2799。[23] Hamel Husain、Ho-Hsiang Wu、Tiferet Gazit、Miltiadis Allamanis 和 Marc Brockschmidt。2019年。CodeSearchNet挑战赛：评估语义代码搜索现状。CoRR abs/1909.09436（2019）。arXiv:1909.09436 http://arxiv.org/abs/ 1909.09436 [24] Srinivasan Iyer、Ioannis Konstas、Alvin Cheung 和 Luke Zettlemoyer。2018年。在程序上下文中将语言映射到代码。arXiv 预印本 arXiv:1808.09588（2018）。

[25] Srinivasan Iyer、Ioannis Konstas、AlvinCheung 和 LukeZettlemoyer。2018年。程序化语境中的语言到代码映射。载于 EMNLP。计算语言学协会，1643–1652。[26] Junguang Jiang、Yang Shu、Jianmin Wang 和 MingshengLong。2022年。深度学习中的迁移能力：综述。arXiv预印本 arXiv:2201.05867（2022年）。[27] Nan Jiang、Thibaud Lutellier 和Lin Tan。2021年。CURE：用于自动程序修复的代码感知神经机器翻译。载于 ICSE。IEEE，1161–1173。[28] Yimin Jiang、Yibo Zhu、Chang Lan、Bairen Yi、YongCui 和 ChuanxiongGuo。2020 年。用于加速异构 {GPU/CPU} 集群中分布式 {DNN} 训练的统一架构。载于第 14 届 USENIX操作系统设计与实现研讨会（OSDI 20）。463–479。[29] Xiaoqi Jiao、Yichun Yin、Lifeng Shang、Xin Jiang、Xiao Chen、Linlin Li、Fang Wang 和Qun Liu。2020年。TinyBERT：为自然语言理解蒸馏 BERT。载于 EMNLP（论文集）（ACL论文集，第EMNLP 2020 卷）。计算语言学协会，4163–4174。[30] AnjanKarmakar 和 Romain Robbes。2021年。预训练代码模型了解哪些代码？载于 2021 年第 36 届 IEEE/ACM自动化软件工程国际会议（ASE）。IEEE，1332–1336。[31] Nikolaus Kriegeskorte、MariekeMur和 Peter ABandettini。2008年。表征相似性分析——连接系统神经科学的各个分支。《系统神经科学前沿》（2008年），4。[32] Chin-Yew Lin。2004年。ROUGE：用于自动评估摘要的软件包。载于 ACL。[33] 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。CoRR abs/1907.11692（2019年）。[34] Shuai Lu、Daya Guo、Shuo Ren、Junjie Huang、Alexey Svyatkovskiy、Ambrosio Blanco、Colin B.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、Shao Kun Deng、ShengyuFu 和 Shujie Liu。2021年。CodeXGLUE：用于代码理解和生成的机器学习基准数据集。CoRR abs/2102.04664（2021年）。[35] WenhaoLu、Jian Jiao 和Ruofei Z hang。2020年。TwinBERT：将知识蒸馏到用于大规模检索的双结构压缩 BERT模型中。载于第 29届 ACM 信息与知识管理国际会议论文集。2645–2652。[36] Thomas JMcCabe。1976年。一种复杂度度量。《IEEE 软件工程汇刊》4（1976年），308–320。[37] Amil Merchant、Elahe Rahimtoroghi、Ellie Pavlick和 IanTenney。2020年。微调过程中 BERT 嵌入发生了什么？载于 BlackboxNLP@EMNLP。计算语言学协会，33–44。[38] AndersMøller和 MichaelI Schwartzbach。2012年。静态程序分析。笔记。2 月（2012年）。[39] Lili Mou、Ge Li、Lu Zhang、Tao Wang和 Zhi Jin。2016年。用于编程语言处理的树结构卷积神经网络。载于第三十届 AAAI 人工智能会议论文集。1287–1293。[40] Changan Niu、Chuanyi Li、Vincent Ng、Jidong Ge、Liguo Huang 和 BinLuo。2022年。SPT-Code：用于学习源代码表征的序列到序列预训练。载于 ICSE。ACM，1–13。[41] Kishore Papineni、SalimRoukos、Todd Ward 和 Wei-JingZhu。2002年。BLEU：一种用于自动评估机器翻译的方法。载于 ACL。ACL，311–318。[42] Julian Aron Prenner 和 Romain Robbes。2021年。使用 OpenAI的 Codex 进行自动程序修复：评估 QuixBugs。arXiv 预印本 arXiv:2111.03922（2021年）。[43] Colin Raffel、Noam Shazeer、Adam Roberts、Katherine Lee、Sharan Narang、Michael Matena、YanqiZhou、WeiLi 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。《机器学习研究杂志》21（2020年），140:1–140:67。[44] Mark Sanderson。2010年。Christopher D. Manning、PrabhakarRaghavan、Hinrich Schütze，《信息检索导论》，剑桥大学出版社。2008年。ISBN-13 978-0-521-86571-5，xxi+ 482 页。《自然语言工程》16，1（2010年），100–103。[45] Patrick Schober、Christa Boer和Lothar ASchwarte。2018年。相关系数：恰当的使用与解释。《麻醉与镇痛》126，5（2018年），1763–1768。[46] Ensheng Shi、Wenchao Gub、Yanlin Wang、Lun Du、Hongyu Zhang、ShiHan、DongmeiZhang 和 Hongbin Sun。2022年。利用多模态对比学习和软数据增强提升语义代码搜索。arXiv 预印本 arXiv:2204.03293（2022年）。[47] Ensheng Shi、YanlinWang、Lun Du、Junjie Chen、Shi Han、Hongyu Zhang、Dongmei Zhang和 Hongbin Sun。2022年。关于神经代码摘要评估。载于 ICSE。[48] Emma Strubell、Ananya Ganesh 和 AndrewMcCallum。2020年。现代深度学习研究中的能源与政策考量。载于 AAAI。AAAI Press，13693–13696。[49] Jeffrey Svajlenko、Judith F Islam、Iman Keivanloo、ChanchalKRoy 和 MohammadMamun Mia。2014年。迈向跨项目代码克隆的大数据整理基准。载于 2014 IEEE 软件维护与演进国际会议。IEEE，476–480。

[50] 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论文集。1433–1443。[51] Telly。2023年。复现包。*ISSTA*（2023年）。https://github.com/DeepSoftwareAnalytics/Telly [52] Ian Tenney、Dipanjan Das 和 Ellie Pavlick。2019年。BERT 重新发现了经典自然语言处理流水线。载于*ACL (1)*。计算语言学协会，4593–4601。[53] Ian Tenney、Patrick Xia、Berlin Chen、Alex Wang、Adam Poliak、R. Thomas McCoy、Najoung Kim、Benjamin Van Durme、Samuel R. Bowman、Dipanjan Das 和 Ellie Pavlick。2019年。你能从上下文中学到什么？探测上下文化词表示中的句子结构。载于*ICLR (Poster)*。OpenReview.net。[54] Sergey Troshin 和 Nadezhda Chirkova。2022年。探测源代码预训练模型。*arXiv* 预印本 *arXiv:2202.08975*（2022年）。[55] Rosalia Tufano、Simone Masiero、Antonio Mastropaolo、Luca Pascarella、Denys Poshyvanyk 和 Gabriele Bavota。2022年。利用预训练模型提升代码审查自动化。载于*ICSE*。ACM，2291–2302。[56] Ashish Vaswani、Noam Shazeer、Niki Parmar、Jakob Uszkoreit、Llion Jones、Aidan N. Gomez、Lukasz Kaiser 和 Illia Polosukhin。2017年。Attention is All you Need。载于*NIPS*。5998–6008。[57] Ramakrishna Vedantam、C. Lawrence Zitnick 和 Devi Parikh。2015年。CIDEr：基于共识的图像描述评估。载于*CVPR*。[58] Yao Wan、Wei Zhao、Hongyu Zhang、Yulei Sui、Guandong Xu 和 Hai Jin。2022年。他们捕捉到了什么？——预训练语言模型的结构分析

面向源代码。在*ICSE*。ACM，2377–2388。[59] Ruize Wang、Duyu Tang、Nan Duan、Zhongyu Wei、Xuanjing Huang、Guihong Cao、Daxin Jiang、Ming Zhou 等。2020年。K-adapter：利用适配器将知识注入预训练模型。*arXiv* 预印本 *arXiv:2002.01808*（2020年）。[60] Wenhui Wang、Furu Wei、Li Dong、Hangbo Bao、Nan Yang 和 Ming Zhou。2020年。MiniLM：面向任务无关压缩预训练Transformer的深度自注意力蒸馏。神经信息处理系统进展 33（2020年），5776–5788。[61] Yanlin Wang、Lun Du、Ensheng Shi、Yuxuan Hu、Shi Han 和 Dongmei Zhang。2020年。*Cocogum*：基于*UMLS*多关系图神经网络的上下文代码摘要。技术报告。微软，MSR-TR-2020-16。 [在线]。可访问：https://www.microsoft.com/en-us/research/publication/cocogum-contextual-code-summarization-with-multi-relational-gnn-on-umls。[62] Yiding Wang、Decang Sun、Kai Chen、Fan Lai 和 Mosharaf Chowdhury。2022年。基于知识引导的层冻结实现高效深度神经网络训练。*CoRR* abs/2201.06227（2022年）。[63] Yue Wang、Weishi Wang、Shafiq R. Joty 和 Steven C. H. Hoi。2021年。CodeT5：用于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。在*EMNLP*（*1*）。计算语言学协会，8696–8708。[64] Zhengran Zeng、Hanzhuo Tan、Haotian Zhang、Jing Li、Yuqun Zhang 和 Lingming Zhang。2022年。针对程序理解与生成的预训练模型的全面研究。在*ISSTA*。ACM，39–51。