利用强化学习和大型语言模型进行代码优化

Shukai Duan [1] Nikos Kanakaris [2] Xiongye Xiao [1] Heng Ping [1] Chenyu Zhou [1] Nesreen K. Ahmed [3] Guixiang Ma [3] Mihai Capot⊠a [3] Theodore L. Willke [3] Shahin Nazarian [1] Paul Bogdan [1]

摘要

代码优化是一项艰巨的任务，需要经验丰富的程序员具有相当高的专业水平。与新硬件架构的快速发展相比，这种水平的专业知识是不够的。为了推进整个代码优化过程，最近的方法依赖于机器学习和人工智能技术。本文引入了一种新的框架来降低代码优化的复杂性。提出的框架建立在大型语言模型(法学硕士)和强化学习(RL)的基础上，并使法学硕士能够在微调过程中接收来自其环境(即单元测试)的反馈。我们将所提出框架与现有的最先进模型进行了比较，并表明它在速度和计算使用量方面更高效，这是由于训练步骤的减少及其对具有较少参数的模型的适用性。此外，所提出框架降低了逻辑和语法错误的可能性。为了评估我们的方法，我们使用CodeT5语言模型和RRHF(一种新的强化学习算法)在PIE数据集上运行了几个实验。我们在优化质量和加速方面采用了各种评估指标。评估结果表明，所提出的框架与使用更短的训练时间和更小的预训练模型的现有模型相比，具有相似的结果。特别是，在%OPT和SP指标方面，我们比基线模型提高了5.6%和2.2。

1 介绍

从纯文本描述或之前的实现中开发软件(代码)是一项要求很高的认知任务。然而，为一个新兴的并行异构计算架构优化代码则明显更具挑战性。代码优化是将给定的程序转换为更高效的版本，同时保持相同的输入和输出(Bunel et al.， 2016)。它要么需要在编译期间利用更高水平的优化(例如-O3)，要么需要专家程序员手动重构他们的代码，使其对某些硬件更优化。随着硬件的快速进步，这两项任务都可能令人望而生畏，因为这会导致开发时间、bug修复和代码优化的增加。因此，使用基于机器学习的解决方案来完成与自动代码优化、代码生成和机器编程相关的任务已经发生了明显的转变(Gottschlich等人，2018b;Sho-jaee等人，2023;Gottschlich等人，2018a)。随着在异构设备、电路和计算系统的最新进展中，研究人员旨在利用机器学习和人工智能来启动、设计和设计一种不断发展的自主但紧凑的机器编程范式和系统，旨在为具有严格功率/能量预算的分布式移动边缘计算系统提供优化代码(Gottschlich等人，2018a)。机器编程可以帮助的方向之一是具有挑战性的代码优化任务。

随着变压器架构的出现(Vaswani et al.， 2017)，大型语言模型(法学硕士)已经成为执行自然语言处理(NLP)任务的默认技术。在其他应用程序中，使用法学硕士的突出解决方案在与软件相关的任务中显示出令人鼓舞的结果。毫无疑问，法学硕士可以执行一些与编程语言相关的任务，包括代码生成、代码优化、缺陷检测和代码完成，仅举几例(Nijkamp等人，2023b;Wang et al.， 2021)。这是如此容易实现，因为GitHub等在线存储库中有大量可用的源代码，便于训练过程(Lu et al.， 2021)。然而，即使法学硕士能够生成表面上看起来正确的代码，他们也无法在没有任何外部帮助的情况下检查其逻辑和语法有效性。因此，建议的片段

[1]Department of Electrical and Computer Engineering, Univer-sity of Southern California [2]Department of Mechanical Engineer-ing and Aeronautics, University of Patras [3]Intel Labs, Intel, USA. Correspondence to: Shukai Duan <shukaidu@usc. edu>, Nikos Kanakaris <nkanakaris@upnet.gr>.

代码并不总是能保证按预期工作(Husain等人，2019)。为此，最近在文献中提出了法学硕士和强化学习(RL)的结合(Shojaee et al.， 2023)。简而言之，包含RL技术使法学硕士能够与其环境进行交互并获得有价值的反馈。这样的反馈通常来自于单元测试的执行，在这里，给定程序的功能正确性得到了确认。

尽管法学硕士和RL的结合为编程语言的分析提供了根本性的变化和改进，但就代码优化问题而言，还没有任何重大进展。一方面，现有的大多数方法使用通用数据集进行微调，这反过来降低了模型提出给定代码高度优化版本的能力。另一方面，专注于代码优化的方法不利用RL。因此，它们无法在生成的代码片段中获得错误的反馈。

为了缓解上述问题，在这项工作中，我们提出了PerfRL，这是一个新颖的基于法学硕士的框架，专注于代码优化任务。我们的方法融合了来自法学硕士和RL的技术，从而能够利用外部反馈，比如来自单元测试的反馈。通过利用RL技术，法学硕士能够接收有关生成程序有效性的信息。这有助于整个过程(i)变得更快(即需要更少的训练)，(ii)使用更小的模型，需要更少的功率和(iii)生成更可能没有错误的优化代码。与此同时，生成的模型的性能保持不变。据我们所知，我们的方法是第一个专门从事代码生成任务的代码优化，同时，与此同时，将单元测试关于代码正确性的反馈纳入其学习过程。

广义而言，与本文工作相关的主要挑战有三个:(i)我们如何将来自单元测试的反馈纳入LLM模型的训练过程中?(ii)我们如何使一个小(紧凑)模型的表现类似于具有数十亿(或更多)参数的大型模型?(iii)我们如何训练如此小的模型，使其生成无错误的可靠代码并处理代码优化任务?

为了实现和开发我们的方法，我们使用了Python编程语言和PyTorch深度学习库。为了评估我们的方法，我们将其与一组处理特定任务的模型进行基准测试。我们的实验结果表明，与基线相比，我们的方法有显著的性能改进。所有相关代码和评估结果都可以在GitHub上公开访问。

的贡献。本文的主要新颖贡献如下:

- 我们提出了一个端到端基于法学硕士的代码优化框架，该框架能够使用强化学习(RL)技术将单元测试的反馈纳入其学习过程。
- 我们的框架是灵活的，可以与大小、复杂性和参数数量不同的法学硕士或任何RL技术一起使用。
- 我们使具有更少参数的更小的语言模型(slm)能够获得与具有数十亿参数的法学硕士相当的结果。
- 我们研究了RL技术与法学硕士相结合的应用，以提高代码优化任务的性能。我们提到所产生的LLM模型专门用于上述任务。
- 我们提出了一种新颖的方法，将来自单元测试的反馈纳入微调过程。这使模型能够学习轻松地生成无错误的代码。
- 我们使用PIE数据集对我们的方法进行了经验测试，这表明与最先进的基线相比，我们的方法具有优越性。

2 相关工作

就代码生成任务而言，法学硕士已经证明了有希望的结果。因此，文献中提出了几种针对不同编程语言的语言模型(LM) (Chen et al.， 2021)。例如，CodeT5 (Wang et al.， 2021)是一个通用语言模型，它是在CodeSearchNet数据集的扩展版本上进行预训练的(Husain et al.， 2019)。它建立在类似于T5的编码器-解码器架构上(Raffel et al.， 2020)，以学习编程和自然语言的通用文本表示。作者使用特定任务的迁移学习和多任务学习技术，将T5微调为各种下游任务(例如代码精化和代码生成)。评估结果表明，相对于CodeXGLUE基准，CodeT5的表现优于其同行(Lu et al.， 2021)。

另一个用于程序合成的大型语言模型家族是CODEGEN (Nijkamp等人，2023b;a)。这些语言模型已经用不同数量的参数进行了训练，从3.5亿到16.1B不等。在训练期间使用了三个数据集，即THEPILE, BIGQUERY和BIGPYTHON。CODEGEN模型实际上是自回归变压器，其目标是预测下一个令牌，与传统模型类似

自然语言(Vaswani等人，2017)。除了模型，作者还发布了一个多轮编程基准，帮助测量给定模型在多轮程序合成方面的能力。实验结果表明，模型的多步程序合成能力与其规模呈正相关。

最近，(Madaan et al.， 2023)提出了PIE数据集。PIE是CodeNet (Puri et al.， 2021)代码样本集合的一个子集。它由程序的轨迹组成，单个程序员从较慢版本的程序开始，并做出改变以提高其性能。PIE的作者已经使用它来评估和改进来自CODEGEN家族的多个模型变体的能力。特别是，他们对这些模型进行了微调，以建议给定代码段的更快版本。此外，他们通过使用少样本学习训练OpenAI的CODEX来评估他们的方法。他们的评估结果显示，超过25%的测试程序的加速比提高了2.5。

一组不同的现代代码生成方法建立在法学硕士和RL的利用之上。总的来说，采用RL技术的本质主要是为了确保生成程序的功能正确性(Liu et al.， 2023)。例如，CodeRL (Le et al.， 2022)将预训练模型与DRL结合起来进行程序合成，以生成满足问题规范的程序。它是CodeT5的扩展，说明了改进的学习目标，并具有更多的参数以及更好的预训练数据。在CodeRL中，训练好的语言模型被用作参与者网络。还加入了一个critic网络，旨在预测生成代码的功能正确性，并向actor提供反馈。作者还引入了一种关键采样策略，使模型能够通过考虑来自单元测试和评论家分数的反馈来重新生成程序。评估结果表明，CodeRL可以在app基准上达到最先进的性能(Hendrycks et al.， 2021)。

(Shen et al.， 2023)中的工作引入了“排序响应以校准测试和教师反馈”(RRTF)框架。它还提供了一个用于编程语言的LLM，即PanGu-Coder2。主要模型是通过使用测试用例和其他启发式偏好的反馈对候选代码片段进行排序来训练的。在HumanEval, CodeEval和LeetCode基准测试上的各种实验表明，PanGu-Coder2可以达到最先进的性能。

Shojaee等人(2023)介绍了PPOCoder，这是一个任务和模型无关的框架，可用于各种代码生成任务。PPOCoder融合了预训练语言模型和近端策略优化(Proximal Policy Optimization, PPO) (Schulman et al.， 2017)，这是一种广泛使用的深度RL技术。PPOCoder考虑了来自编译器和单元测试的反馈以及与语法相关的反馈。这促进了模型在语法和逻辑方面生成更好的代码。实验结果指出，在生成代码的语法和功能有效性方面，popcode比它的基线更有效。

虽然PPO是RL中最流行的策略梯度方法之一，但文献中已经提出了一系列替代算法。这些替代方案的目的是避免PPO的缺点(例如对超参数的敏感性)，同时也降低了整体算法的复杂度。更具体地说，偏好排序优化(PRO) (Song等人，2023)采用了“从人类反馈中强化学习”(RLHF)中的Bradley-Terry比较(Stiennon等人，2020;薛等人，2023)。它还将LLM生成的响应的概率排序与人类的偏好排序对齐。所进行的实验表明，PRO通过有效地将法学硕士与人类偏好相结合而优于其同行。最近，一种很有前途的算法RRHF被引入(Yuan et al.， 2023)。与RLHF类似，它支持法学硕士与人类偏好的对齐。作者认为，RRHF可以很容易地进行调整，并达到与Anthropic 's Helpful and无害数据集中的PPO相当的性能(Bai et al.， 2022)。

3 问题定义

我们考虑生成输入代码的优化版本的问题。更正式地说，给定一组输入程序X，任务是为每个X∈X生成一组优化程序X，⊠。优化后的程序版本应该接受与原始版本相同的输入，并产生相同的输出。

为此，对于每个x∈x，我们使用采样策略s∈{贪心，随机}，生成一组候选程序y∈y。然后我们的目标是最大化成本函数:

$$\text{cost}(x, y_{best}) = \text{eq}(R, y_{best}) + \text{perf}(y_{best}) \tag{1}$$

其中ybest∈Y是最佳候选，术语eq(R, ybest)度量具有给定输入的生成序列与单元测试输出匹配的能力，术语perf(y_{best})度量单元测试中ybest对x的性能改进情况。

同时，我们还旨在通过学习现有的训练脚本，最大限度地从输入程序的分布中生成y_{best}的概率。

$$\theta^* = \arg\max_{\theta} P(y_{\text{best}}|x;\theta) \tag{2}$$

这里，θ^*represents最优的模型参数集。由于最近的解决方案主要依赖于法学硕士，

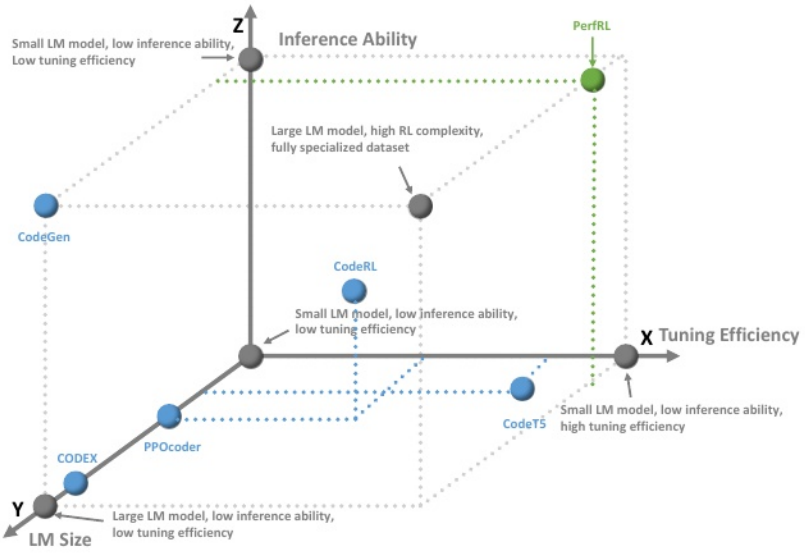

图1所示。代码优化问题从三个不同的角度来看，即LM的大小(y轴)、模型的推理能力(z轴)、调优效率(x轴)。我们的方法(绿点)使用一个完全专门化的数据集(PIE)，一个中等大小的LM模型(CodeT5)，并且具有较低的推理能力。规模较大的lm包括CodeGen家族和Codex家族的lm。

这个问题的主要挑战是法学硕士不能自然地与他们的环境互动。因此，它们可以潜在地产生给定代码的优化版本，这些版本表面上看起来正确，但包含语法或逻辑错误。此外，基于法学硕士和RL的现有解决方案并不专注于代码优化任务。因此，它们不能表现得足够好，可能会产生幻觉，即捏造不可编译的无意义代码片段。(Zhang等人，2023)。这个问题的另一个挑战是所需模型的规模不断增加。这些模型需要足够数量的计算资源，这反过来导致了能源消耗的增加。

考虑到上述挑战，我们提出了一种不同的方法来训练用于代码生成问题的LLM模型。我们没有单独使用LLM或用一些RL步骤构建通用LLM模型，而是设计了一个针对代码生成任务的框架(参见第4.3节)。图1突出了代码优化问题的不同视角，以及我们的框架与现有框架的不同之处。

如图1所示，我们可以从三个不同的角度来看待代码优化问题——这取决于RL算法的复杂性、LM的大小以及所使用的数据集与代码优化任务的相关性。例如，在我们的实验中，我们使用了一个中等规模的LM (CodeT5)、具有中等复杂性的RRHF RL算法和一个专门用于所考虑任务的数据集(PIE)。然而，所提出的框架可以促进所选LM模型的大小、RL算法的复杂性和所使用数据集的相关性的不同设置和组合。

与代码生成问题相关的一些研究问题如下:

- 我们如何减少生成代码优化版本所需的模型的大小?
- 我们如何确保生成的代码是可靠的，产生预期的结果，并且没有语法或逻辑错误?

4 建议的方法

我们提出了PerfRL，一个基于强化学习的LLM框架，用于代码性能优化。PerfRL提高了法学硕士生成优化代码的能力，这些代码可以改进程序运行时，同时在逻辑和语法上也是正确的。为此，它利用了法学硕士和RL的技术。它由三个主要部分组成:(1)LLM模型的微调(第4.1节)，(2)样本生成(第4.2节)，以及(3)强化学习监督和纠正(第4.3节)。

图2说明了所提出方法的架构。我们在本节中详细描述了每一个步骤。在训练阶段，我们首先使用专门用于代码生成任务的数据集对法学硕士进行微调。然后，我们利用微调后的模型通过采用不同的采样来为给定的输入代码生成优化的代码版本

策略，如第4.2节所述。然后，我们计算每个生成代码的奖励值和分数，并基于RL技术计算损失值(第4.3节)。然后，在推理过程中，我们使用微调的LLM模型为每一个给定的代码生成一个优化版本。

4.1 LLM模型的微调

第一步是在专门用于代码优化任务的数据集上微调LLM模型。为了提高效率和简单性，在我们的实验中(参见第5节)，我们使用了轻量级的CodeT5模型(Wang等人，2021)，但是我们的框架可以与各种llm一起运行，无论它们的大小如何。在CodeT5的初始论文中，作者使用自然语言(NL)或自然语言和编程语言(PL)的组合作为输入。特别是，他们在诸如标识符感知去噪、标识符标记、掩码标识符预测和双峰双生成等任务上对模型进行了预训练。因此，我们认为这种特殊类型的模型更有能力理解NL-PL输入。因此，我们遵循NL-PL方法将输入提供给CodeT5模型。同时，作者在(Madaan et al.， 2023)中认为，少样本采样策略有利于产生优化的输出。因此，我们将动作的要求模型提高执行性能的自然语言与输入编程语言连接起来，并将它们输入到模型中。

CodeT5模型分别期望将较慢和较快的预处理代码查询作为输入和目标。微调的目标是最小化交叉熵损失:

$$L_{ft}(\theta) = -\frac{1}{N}\sum_{i=1}^{N}\sum_{j=1}^{V}\log(p_{i,j}) \tag{3}$$

其中θ是给定LLM的参数(在我们的例子中是CodeT5)，N是标记的数量，V是标记器的词汇集，y是在真实输出序列中位置i的词汇表中第j个标记的嵌入值，$p_{i,j}$是位置i的词汇表中第j个标记的预测概率。

4.2 样本代

为了生成代码的候选样本，我们在训练、测试和验证期间采用了不同的采样策略。更具体地说，我们使用贪心采样和随机采样。

带波束搜索的贪婪采样为相同的输入生成B个不同的样本。对于序列生成的每个步骤，该模型采用样本k的当前生成的token序列，并计算对应样本的下一个token在词汇表上的概率分布。对每个样本计算top B候选词汇表及其累积概率并进行排名。选择所有累积概率最大的候选词汇中的top B序列来重复生成下一个token的过程。

$$y_t^{(1)}, \ldots, y_t^{(B)} = \text{top}_B \left\{ P(y_t | y_1^{(c)}, \ldots, y_{t-1}^{(c)}, x) \right\} \tag{4}$$

其中x是模型的输入，Y = (Y_1，⋯，y_T)，$y_t \in V$是模型$c \in [0,B-1]$的输出token。

第二种采样策略是随机采样，它使用相同的输入生成大量不同的样本。在生成过程中，我们设置了一个温度值Tem，以影响输出的多样性。给定给定的logits l的概率分布，缩放后的l'等于:

$$l = \log P(y_t | y_1, \ldots, y_{t-1}, x) \tag{5}$$

$$l' = \frac{l}{Tem} \tag{6}$$

缩放后每个token的概率为:

$$p_i = \frac{exp(l'_i)}{\sum_i exp(l'_i)} \tag{7}$$

基于p_i，我们选择top-k个token，并随机选择其中一个。我们重复这样的采样步骤t次，直到达到最大长度或结束条件。
下一步是生成代码样本。我们观察到，模型无法从大多数最初生成的代码样本中学习，因为它们有语法或逻辑错误;结果，代码样本无法通过单元测试。因此，为了在训练过程中生成代码样本，我们首先应用随机抽样，并从独立运行中选择两个候选者。然后，我们执行一次贪婪采样，以找到概率最高的候选人。最后，为了确保至少有一个正确的样本存在，我们将数据集中的目标序列与样本一起包含在列表中。然后，这4个样本在每一步都被输入到模型中。

在验证和测试(即推断)期间，我们使用带波束搜索的贪婪采样生成4个样本，并返回top-2最佳候选。对于给定的输入，我们生成两个候选者进行评估。与(Lu et al.，2021)类似，我们认为一个样本是成功的，当它与输入代码相比具有更好的执行时间。

4.3 强化学习

我们的RL步骤建立在RRHF的基础上，RRHF是一个轻量级的RL框架，用于调整带有反馈分数的llm。在

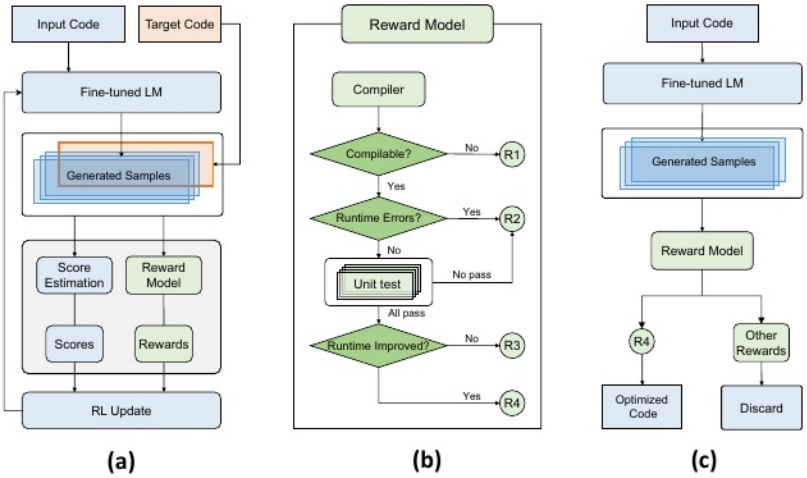

图2。PerfRL框架概述。(一)培训。我们首先使用整个训练数据集对LLM模型进行微调。然后，我们将输入代码传递到微调后的LM中，为每个输入程序生成预定义数量的优化样本。我们为每个样本分配一个得分值，并使用奖励模型计算其奖励。我们利用分数和奖励值来计算RL的L_{rank}和L_{tuning}损失值。通过结合之前的两个损失值计算出最终的损失L，并用于重新训练模型。通过这种方式，我们的框架将来自单元测试的反馈纳入其训练过程中。因此，它更有可能生成没有语法和逻辑错误的优化代码，减轻幻觉。(b)奖励模型。根据代码的状态(例如，可以编译，有运行时错误或通过所有单元测试)，奖励模型会给出不同的奖励值。(c)推理。在推理过程中，最终的LLM被用来生成候选源代码的多个样本。然后使用奖励模型对这些生成的源代码进行评估。所有没有得到R4奖励的样本都会被过滤掉。在这个淘汰过程之后剩下的源代码被认为是优化后的代码。

在每个RL步骤中，我们的目标是通过对LLM模型进行排序和微调来最大化生成高回报代码片段的概率。对于每个RL步骤，我们从训练集中采样所有数据并生成候选输出。最后，对于每个候选输出，我们计算分数、奖励和损失，以便调整模型参数θ。

4.3.1 奖励

在代码生成步骤之后，我们使用Python解释器执行每个代码示例。如果检测到错误，就会显示错误消息。如果一段代码没有语法错误，我们使用单核上的相关单元测试来测试它是否有逻辑错误。一个输入代码o的执行时间et_o是在执行过程中测量的，假设在预定义的时间段(et_o≤timeout内没有运行时错误)。与(Le et al., 2022)中提出的奖励函数类似，对于每个样本y∈y，奖励r(y)的计算如下:
⊠⊠R2R1if y(c)无法编译⊠⊠⊠⊠⊠如果$y^{(c)}$运行时错误、超时或失败r(y(c)) =任何单元测试⊠R3⊠⊠⊠⊠⊠如果y(c)通过了所有单元测试R4如果$y^{(c)}$passed和改进的运行时间(8)

正如RRHF (Yuan et al.， 2023)的论文所述，只要将更理想的结果与更高的值相关联，分配给不同代码样本的奖励值与损失的计算无关(参见4.3节)。

4.3.2 得分函数

对于给定的生成代码x，我们有一个候选序列y(c)，其中0 < c < b。对于每个y(c)，我们计算序列的预测分数为每个token的log概率除以token数量t的和:

$$p^{(c)} = \frac{\sum_t \log P(y_t^{(c)} | y_1^{(c)}, \ldots, y_{t-1}^{(c)}, x)}{||y_t^{(c)}||} \tag{9}$$

4.3.3 最小化奖励较少的输出的概率

从评价系统的执行中，我们在给定输入x的情况下，对每个y(c)得到我们的奖励r(c) = r(y(c))。我们通过以下方式最大化正确响应的损失并最小化错误响应:

$$L_{\text{rank}} = \sum_{r^{(a)} < r^{(b)}} \max(0, p^{(a)} - p^{(b)}) \tag{10}$$

4.3.4 微调损失,最大化最佳奖励候选

由于我们的方法是基于RRHF的，我们计算了与微调类似的最佳响应的交叉熵:

$$L_{\text{tuning}} = -\sum_{t} \log P(y_{best,t} | x, y_{best,<t}) \tag{11}$$

其中y_{best}在所有0 < I < k中$r(y_i)$最大。
通过这种方式，PerfRL可以不断增强其输出，即使在最佳生成的代码包含语法错误或没有最佳执行时间的情况下也是如此。由于我们只是根据输入程序的执行时间来比较候选输出，因此很有可能两个候选输出代码将获得相同的奖励。loss Ltuning根据采样策略选择y_{best}。为了培养模型独立发现优化策略的倾向，我们校准了它的学习优先级。最高的优先级被分配给从随机样本中学习，其次是对贪婪样本的偏好。目标程序被指定为最终的学习优先级。

4.3.5 损失

我们对所有输入数据进行采样，并计算损失L作为来自相同输入的样本的Lrank和Ltuning的组合。

$$L^z = \left(aL^z_{\text{rank}} + L^z_{\text{tuning}}\right) \tag{12}$$

$$z \in \text{samplefrom}(X) \tag{13}$$

其中X是来自数据集的所有输入提示，a是一个常数。

5 实验

5.1 数据集

为了微调和评估PerfRL，我们使用了来自PIE的数据集(Madaan等人，2023)。这个数据集由大约40k个Python文件，88k个c++文件和3.6k个Java文件组成。PIE捕获程序员在一段时间内为改进代码所做的渐进式更改。数据集还包含同一程序员编写的(慢的、快的)代码对。我们在与Python文件相关的数据集子集上运行我们的实验。训练集、测试集和验证集分别由大约36k、1k和2k个样本组成。每个样本都至少有一个相关的单元测试文件，该文件需要特定的输入，并具有预期的输出结果。我们通过以5秒的执行限制执行所有输入源代码来测试数据集的准确性。我们发现，72.4%的训练数据、76.4%的测试数据和70.8%的验证数据是可执行的。在我们的训练过程中，我们跳过了所有最初不是可执行的输入代码。至于测试和验证，我们使用了所有的数据，以便将我们的方法与基线模型进行比较。

5.2 设置

我们对CodeT5模型的一个实例运行微调和强化学习步骤，该模型有6000万个参数，学习率为2×10^{-5}。为了减少训练时间，我们将整个过程运行为8个RL步骤，分为3个epoch。对于模型的训练，我们在50核的Ubuntu 20.04服务器上使用了一个40GB RAM的NVIDIA A100 GPU显卡。我们的模型在大约30小时内进行了训练。每个epoch计算(i)生成序列的分数(见等式9)和(ii)来自相同输入数据的数据集的所有样本的损失。在训练阶段，我们将温度设置为1,top_k设置为50，以进行随机采样。

如前所述，在运行强化学习步骤之前，我们使用一次学习设置微调CodeT5模型。也就是说，我们将数据集的每个样本输入到模型中一次，并计算公式3中所描述的损失。我们将学习率设置为5×10^{-5}，批处理大小设置为32。结果如表1所示。

在验证过程中，我们使用贪心采样，对输入代码进行大小为4的波束搜索，以生成候选样本。从这4个样本中，我们选择累积概率最大的前2个样本。然后，我们使用奖励函数r为贪心采样轮设置奖励值。验证的编译率测量了在输入代码总数上通过编译的轮数。验证的通过率衡量的是在输入代码总数上通过所有单元测试执行的轮数。验证的优化率测量通过执行的轮数，并且在单个核心中比输入代码总数有更好的执行时间。

我们还在1000个样本上测试了我们的方法。在整个过程中，我们在RL框架之前运行测试，以测试微调模型的当前性能，在RL框架之后运行测试，以显示RL的贡献。

表1。PerfRL和基线模型的评价结果。粗体部分为PerfRL的评估结果。星号(*)表示基线模型。第一块显示了PIE论文(Madaan等人，2023年)中报告的结果。

METHOD	Model Size	Sample strategy	%OPT	SP	RTR
CODEGEN-16B	16B	greedy and 1-shot	2.2	1.55	25.05
CODEGEN-2B	2B	greedy	8.2	2.32	48.23
CODEGEN-16B	16B	greedy	14.6	1.69	51.25
CODEX	–	greedy	14.3	2.7	53.49
CodeT5 (Before RL)	60M	greedy and 0-shot	0	0	0
CodeT5 (24 Fine-tuning epoch)*	60M	greedy and 0-shot	0.5	2.27	53.42
PerfRL (8 RL steps)	60M	greedy and 0-shot	**2.8**	**4.93**	**36.93**

对于我们的实验，我们将奖励函数r的奖励值设置为R1 = 0, R2 = 1, R3 = 1.3和R4 = 2。正如在4.3.1节中已经提到的，实际的奖励值不会影响RL步骤的性能，只要我们将更高的值分配给更可取的样本。

5.3 基线

我们使用CodeT5模型的微调版本，其中24个epoch作为我们的主要基线。此外，我们将我们的结果与原始PIE论文的结果进行了比较，其中使用了CodeGen和Codex家族的模型进行实验。我们使用5.4节中描述的评估指标，将我们的方法与主要基线模型进行基准测试。

5.4 评价指标

我们使用以下评估指标，这些指标也在PIE (Madaan等人，2023)数据集的论文中定义:

- 优化百分比(%OPT):测试集上被给定方法改进的样本比例。

- Speedup (SP):执行时间的实际(绝对)改进SP(o, n) = (no)，其中o和n分别是旧的和新的执行时间。
  –

- 运行时减少(Runtime Reduction, RTR):运行时减少且语法和逻辑正确的程序在执行时间上的标准化改进，RTR(0, n) =(0–n o ×100)。我们提到
  在测试集上报告平均RTR。

为了测量每个生成程序的执行时间，我们使用所有单元测试计算累计执行时间。我们对每个实验运行三次，测量每组单元测试的平均时间，以确保执行时间的减少不受随机因素的影响。

5.5 评价结果

表1给出了所提出方法和基线模型的平均%OPT、SP和RTR分数。尽管PerfRL依赖于一个参数较少的模型，但它在贪心和1-shot版本的CodeGen- 16b和CodeGen-2B模型方面的表现是一样的，甚至更好。此外，如图3所示，通过率和优化率都与RL步骤成正比。因此，我们认为RL步骤的使用使较小的模型能够以更少的资源和耗时的方式轻松地学习源代码优化任务。更具体地说，我们训练我们的模型30小时，而不是训练大约3天的基线CodeGen，需要更强大的机器(即2 ×NVIDIA A6000用于CodeGen-2B, 4 ×NVIDIA A6000用于CodeGen- 16b)。

为了确定我们的RL步骤是否能够在训练期间通过随机抽样或贪婪抽样生成良好的候选对象，我们测量了编译、通过和优化率，如图4所示。考虑到我们将目标程序与生成的样本串联起来的采样策略，我们根据经验观察到这三个值的阈值约为20%(红色虚线)。由于这三个比率总是高于阈值，它们表示我们的模型自己生成有意义的候选项，而不是纯粹基于目标程序。

5.6 讨论

我们的框架表明，与简单的微调相比，基于强化学习的微调策略能够更有效地调整模型。特别是，对于源代码优化来说，让模型拥有自己探索搜索空间的能力至关重要。然而，功能等效的代码很难学习，如果它们的语义截然不同，LLM分词器就很难在有限的数据上理解代码的结构。我们的方法鼓励llm对输入源代码进行轻微修改。然而，我们相信具有结构信息的ML技术

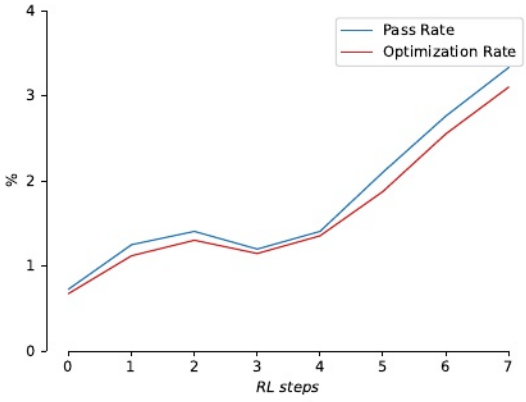

图3。在经过微调的CodeT5模型的RL步骤上验证数据的通过率和优化率。

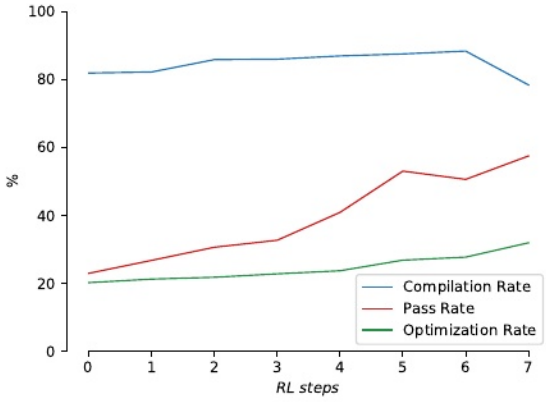

图4。使用经过微调的CodeT5模型生成的程序的每个RL步骤的编译、通过和优化率。所有的比率都是通过编译、通过或优化的生成程序的数量除以生成程序的总数来计算的。

可以提高法学硕士理解两个源代码之间复杂结构差异的能力。

6 结论

在本文中，我们提出了一种新的代码优化任务框架，称为PerfRL。我们的框架结合了来自法学硕士和RL的技术，它允许语言模型在它们的微调过程中考虑来自单元测试的反馈。为了演示框架的适用性，我们对PIE数据集上的CodeT5模型进行了微调。我们将所提出的方法与一组仅依赖简单微调的基线模型进行基准测试，而忽略了生成代码的逻辑和语法正确性。评估结果表明，通过采用我们的框架，人们可以用更小的语言模型和更少的参数达到最先进的性能，这反过来导致更低的能耗，这在边缘计算设备上是至关重要的。

未来的工作方向包括(i)集成代码的图形表示，如ProGraML (Cummins等人，2020)，(ii)图形神经网络和法学硕士的结合，以捕获代码不同部分的结构和语言特征，(iii)为RL实现更复杂的评分和奖励功能。(iv)对所建议方法在具有严格功率或能量预算的边缘计算系统中的适用性的调查(v)使用具有不同架构但具有相似参数数量的模型对我们的框架进行评估。

参考文献

Bai, Y., Jones, A., Ndousse, K., Askell, A., Chen, A., Das-Sarma, N., Drain, D., Fort, S., Ganguli, D., Henighan, T., Joseph, N., Kadavath, S., Kernion, J., Conerly, T., El-Showk, S., Elhage, N., Hatfield-Dodds, Z., Hernan-dez, D., Hume, T., Johnston, S., Kravec, S., Lovitt, L., Nanda, N., Olsson, C., Amodei, D., Brown, T., Clark, J., McCandlish, S., Olah, C., Mann, B., and Kaplan, J. Train-ing a helpful and harmless assistant with reinforcement learning from human feedback, 2022.

Bunel, R., Desmaison, A., Kumar, M. P., Torr, P. H. S., and Kohli, P. Learning to superoptimize programs. *CoRR*, abs/1611.01787, 2016. URL http://arxiv.org/abs/1611.01787.

Chen, M., Tworek, J., Jun, H., Yuan, Q., de Oliveira Pinto, H. P., Kaplan, J., Edwards, H., Burda, Y., Joseph, N., Brockman, G., Ray, A., Puri, R., Krueger, G., Petrov, M., Khlaaf, H., Sastry, G., Mishkin, P., Chan, B., Gray, S., Ryder, N., Pavlov, M., Power, A., Kaiser, L., Bavar-ian, M., Winter, C., Tillet, P., Such, F. P., Cummings, D.,

Plappert, M., Chantzis, F., Barnes, E., Herbert-Voss, A., Guss, W. H., Nichol, A., Paino, A., Tezak, N., Tang, J., Babuschkin, I., Balaji, S., Jain, S., Saunders, W., Hesse, C., Carr, A. N., Leike, J., Achiam, J., Misra, V., Morikawa, E., Radford, A., Knight, M., Brundage, M., Murati, M., Mayer, K., Welinder, P., McGrew, B., Amodei, D., McCandlish, S., Sutskever, I., and Zaremba, W. Evaluating large language models trained on code,

2021.

Cummins, C., Fisches, Z. V., Ben-Nun, T., Hoefler, T., and Leather, H. Programl: Graph-based deep learning for program optimization and analysis, 2020.

Gottschlich, J., Solar-Lezama, A., Tatbul, N., Carbin, M., Rinard, M., Barzilay, R., Amarasinghe, S., Tenenbaum, J. B., and Mattson, T. The three pillars of machine pro-gramming. In *Proceedings of the 2nd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, pp. 69–80, 2018a.

Gottschlich, J., Solar-Lezama, A., Tatbul, N., Carbin, M., Rinard, M., Barzilay, R., Amarasinghe, S., Tenenbaum, J. B., and Mattson, T. The three pillars of machine programming. In *Proceedings of the 2nd ACM SIG-PLAN International Workshop on Machine Learning and Programmin g Languages*, MAPL 2018, pp. 69–80, New York, NY, USA, 2018b. Association for Com-puting Machinery. ISBN 9781450358347. doi: 10. 1145/ 3211346. 3211355. URL https://doi. org/10. 1145/ 3211346.3211355.

Hendrycks, D., Basart, S., Kadavath, S., Mazeika, M., Arora, A., Guo, E., Burns, C., Puranik, S., He, H., Song, D., and Steinhardt, J. Measuring coding challenge competence with apps. *NeurIPS*, 2021.

Husain, H., Wu, H.-H., Gazit, T., Allamanis, M., and Brockschmidt, M. CodeSearchNet challenge: Evalu-ating the state of semantic code search. *arXiv preprint arXiv:1909.09436*, 2019.

Le, H., Wang, Y., Gotmare, A. D., Savarese, S., and Hoi, S. CodeRL: Mastering code generation through pre-trained models and deep reinforcemen t learning. In Oh, A. H., Agarwal, A., Belgrave, D., and Cho, K. (eds.), *Advances in Neural Information Processing Systems*,

2022. URL https://openreview. net/forum? id= WaGvb7OzySA.

Liu, J., Zhu, Y., Xiao, K., Fu, Q., Han, X., Yang, W., and Ye, D. Rltf: Reinforcement learning from unit test feedback,

2023.

Lu, S., Guo, D., Ren, S., Huang, J., Svyatkovskiy, A., Blanco, A., Clement, C. B., Drain, D., Jiang, D., Tang, D., Li, G., Zhou, L., Shou, L., Zhou, L., Tufano, M.,

Gong, M., Zhou, M., Duan, N., Sundaresan, N., Deng, S. K., Fu, S., and Liu, S. Codexglue: A machine learning benchmark dataset for code understanding and generation. *CoRR*, abs/2102.04664, 2021.

Madaan, A., Shypula, A., Alon, U., Hashemi, M., Ranganathan, P., Yang, Y., Neubig, G., and Yazdanbakhsh, A. Learning performance-improving code edits. *arXiv preprint arXiv:2302. 07867*, 2023.

Nijkamp, E., Hayashi, H., Xiong, C., Savarese, S., and Zhou, Y. Codegen2: Lessons for training llms on programming and natural languages. *ICLR*, 2023a.

Nijkamp, E., Pang, B., Hayashi, H., Tu, L., Wang, H., Zhou, Y., Savarese, S., and Xiong, C. Codegen: An open large language model for code with multi-turn program synthe-sis. *ICLR*, 2023b.

Puri, R., Kung, D., Janssen, G., Zhang, W., Domeniconi, G., Zolotov, V., Dolby, J., Chen, J., Choudhury, M., Decker, L., Thost, V., Buratti, L., Pujar, S., Ramji, S., Finkler, U., Malaika, S., and Reiss, F. Codenet: A large-scale ai for code dataset for learning a diversity of coding tasks, 2021.

Raffel, C., Shazeer, N., Roberts, A., Lee, K., Narang, S., Matena, M., Zhou, Y., Li, W., and Liu, P. J. Exploring the limits of transfer learning with a unified text-to-text transformer. *Journal of Machine Learning Research*, 21(140):1–67, 2020. URL http://jmlr.org/papers/v21/20-074.html.

Schulman, J., Wolski, F., Dhariwal, P., Radford, A., and Klimov, O. Proximal policy optimization algorithms, 2017.

Shen, B., Zhang, J., Chen, T., Zan, D., Geng, B., Fu, A., Zeng, M., Yu, A., Ji, J., Zhao, J., Guo, Y., and Wang, Q. Pangu-coder2: Boosting large language models for code with ranking feedback, 2023.

Shojaee, P., Jain, A., Tipirneni, S., and Reddy, C. K. Execution-based code generation using deep reinforce-ment learning, 2023.

Song, F., Yu, B., Li, M., Yu, H., Huang, F., Li, Y., and Wang, H. Preference ranking optimization for human alignment, 2023.

Stiennon, N., Ouyang, L., Wu, J., Ziegler, D. M., Lowe, R., Voss, C., Radford, A., Amodei, D., and Christiano, P. F. Learning to summarize from human feedback. *CoRR*, abs/2009. 01325, 2020. URL https://arxiv.org/abs/2009.01325.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L., and Polosukhin, I. Attention

is all you need. *CoRR*, abs/1706.03762, 2017. URL http://arxiv.org/abs/1706.03762.

Wang, Y., Wang, W., Joty, S., and Hoi, S. C. CodeT5: Identifier-aware unified pre-trained encoder-decoder mod-els for code understanding and generation. In *Pro-ceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pp. 8696–8708, On-line and Punta Cana, Dominican Republic, November 2021. Association for Computational Linguistics. doi: 10.18653/v1/2021. emnlp-main. 685. URL https://aclanthology.org/2021.emnlp-main.685.

Xue, W., An, B., Yan, S., and Xu, Z. Reinforcement learning from diverse human preferences, 2023.

Yuan, Z., Yuan, H., Tan, C., Wang, W., Huang, S., and Huang, F. Rrhf: Rank responses to align language models with human feedback without tears, 2023.

Zhang, Y., Li, Y., Cui, L., Cai, D., Liu, L., Fu, T., Huang, X., Zhao, E., Zhang, Y., Chen, Y., Wang, L., Luu, A. T., Bi, W., Shi, F., and Shi, S. Siren’s song in the ai ocean: A survey on hallucination in large language models, 2023.