LEVERAGING REINFORCEMENT LEARNING AND LARGE LANGUAGE MODELS FOR CODE OPTIMIZATION

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

ABSTRACT

Code optimization is a daunting task that requires a significant level of expertise from experienced programmers. This level of expertise is not sufficient when compared to the rapid development of new hardware architectures. Towards advancing the whole code optimization process, recent approaches rely on machine learning and artificial intelligence techniques. This paper introduces a new framework to decrease the complexity of code optimization. The proposed framework builds on large language models (LLMs) and reinforcement learning (RL) and enables LLMs to receive feedback from their environment (i.e., unit tests) during the fine-tuning process. We compare our framework with existing state-of-the-art models and show that it is more efficient with respect to speed and computational usage, as a result of the decrement in training steps and its applicability to models with fewer parameters. Additionally, our framework reduces the possibility of logical and syntactical errors. Toward evaluating our approach, we run several experiments on the PIE dataset using a CodeT5 language model and RRHF, a new reinforcement learning algorithm. We adopt a variety of evaluation metrics with regards to optimization quality, and speedup. The evaluation results demonstrate that the proposed framework has similar results in comparison with existing models using shorter training times and smaller pre-trained models. In particular, we accomplish an increase of 5.6% and 2.2 over the baseline models concerning the $\%OPT$ and SP metrics.

1 INTRODUCTION

Developing software (codes) from plain text descriptions or from prior implementations is a highly demanding cognitive task. However, it is significantly more challenging to optimize the code for an emerging parallel heterogeneous computing architecture. Code optimization is the task of converting a given program to a more efficient version while retaining the same input and output (Bunel et al., 2016). It requires either the utilization of a higher level of optimization (e.g., -O3) during compilation or expert programmers to manually refactor their code to make it more optimized for certain hardware. Both of these tasks can be daunting with the rapid advancement in hardware, which leads to an increase in development time, bug fixing, and code optimization. Thus, there has been a noticeable shift to utilizing machine learning-based solutions for tasks related to automatic code optimization, code generation, and machine programming in general (Gottschlich et al., 2018b; Shojaee et al., 2023; Gottschlich et al., 2018a). Along With

recent advancements in heterogeneous devices, circuits, and computing systems, researchers have aimed to utilize machine learning and AI to initiate, design, and engineer an evolving autonomous yet compact machine programming paradigm and system that seek to provide optimized codes for distributed mobile edge computing systems having strict power/energy budgets (Gottschlich et al., 2018a). One of the directions machine programming can help is the challenging task of code optimization.

With the advent of transformer architectures (Vaswani et al., 2017), large language models (LLMs) have emerged as the default technology to perform tasks in natural language processing (NLP). Among other applications, prominent solutions that use LLMs have shown encouraging results for software-related tasks. Undoubtedly, LLMs can perform several tasks related to programming languages, including code generation, code optimization, defect detection, and code completion, to name a few (Nijkamp et al., 2023b; Wang et al., 2021). This is so easily achievable due to the large amounts of source code available in online repositories such as GitHub, which facilitate the training process (Lu et al., 2021). However, even if LLMs are capable of producing code that superficially looks correct, they are unable to check its logical and syntactical validity without any external assistance. As a result, the suggested snippets of

---

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

code are not always guaranteed to work as expected (Husain et al., 2019). To that end, the combination of LLMs and Reinforcement Learning (RL) has been recently proposed in the literature (Shojaee et al., 2023). Briefly, the inclusion of RL techniques enables LLMs to interact with their environment and receive valuable feedback. Such feedback often comes from the execution of unit tests, where the functional correctness of a given program is confirmed.

Despite the radical changes and improvements that the combination of LLMs and RL offers to the analysis of programming languages, there has not been any significant progress as far as the problem of code optimization is concerned. On the one hand, the majority of the existing approaches use general-purpose datasets for fine-tuning, which in turn reduces the ability of a model to propose highly optimized versions of a given code. On the other hand, methods focusing on code optimization do not exploit RL. Thus, they are incapable of getting feedback for errors in the produced snippets of code.

To mitigate the issues mentioned above, in this work, we propose PerfRL, a novel LLM-based framework that concentrates on the task of code optimization. Our approach fuses techniques from LLMs and RL to enable the utilization of external feedback, such as feedback from unit tests. By leveraging RL techniques, LLMs are able to receive information about the validity of the generated program. This helps the whole process to (i) become faster (i.e. less training is required), (ii) use a smaller model that requires less power and (iii) generate optimized code that is more likely to be free from errors. At the same time, the performance of the produced model remains the same. To the best of our knowledge, our approach is the first that specializes in the task of code generation for code optimization, while, at the same time, incorporating feedback from unit tests with respect to the correctness of the code into its learning process.

Broadly speaking, there are three main challenges related to the work of this paper: (*i*) How can we incorporate feedback from unit tests into the training process of an LLM model? (*ii*) How can we make a small (compact) model perform similarly to large models with billions (or more) of parameters? (*iii*) How can we train such a small model such that it generates reliable code free from errors and deals with the code optimization task?

To implement and develop our approach, we use the Python programming language and PyTorch deep learning library. To evaluate our approach, we benchmark it against a list of models that deal with the specific task. Our experimental results demonstrate a significant performance improvement of our approach against the baselines. All related code and evaluation results are openly accessible on GitHub.

Contributions. The main novel contributions of this paper are the following:

- We propose an end-to-end LLM-based framework for code optimization, which is capable of incorporating feedback from unit tests into its learning process using reinforcement learning (RL) techniques.

- Our framework is flexible and can be used with LLMs varying in size, complexity, and number of parameters or with any RL technique.

- We enable smaller language models (SLMs) with fewer parameters to achieve results comparable to those of LLMs with billions of parameters.

- We investigate the application of RL techniques combined with LLMs to improve performance on code optimization tasks. We mention that the produced LLM model specializes in the aforementioned tasks.

- We propose a novel approach that incorporates feedback from unit tests into the fine-tuning process. This allows the model to learn to produce error-free code easily.

- We empirically test our approach using the PIE dataset, which demonstrated the superiority of our approach compared to the sate-of-the-art baselines.

2 RELATED WORK

LLMs have demonstrated promising results as far as the task of code generation is concerned. As a result, several language models (LM) for different programming languages have been proposed in the literature (Chen et al., 2021). For instance, CodeT5 (Wang et al., 2021) is a general language model, which is pre-trained on an extended version of the CodeSearchNet dataset (Husain et al., 2019). It builds on an encoder-decoder architecture similar to T5 (Raffel et al., 2020) to learn generic text representations for programming and natural languages. The authors have fine-tuned T5 to a variety of downstream tasks (e.g. code refinement and code generation) using task-specific transfer learning and multi-task learning techniques. The evaluation results have shown that CodeT5 outperformed its counterparts with respect to the CodeXGLUE benchmark (Lu et al., 2021).

Another family of large language models for program synthesis is CODEGEN (Nijkamp et al., 2023b;a). These language models have been trained with different numbers of parameters, ranging from 350M to 16.1B. Three datasets have been used during the training period, i.e. THEPILE, BIGQUERY and BIGPYTHON. The CODEGEN models are actually autoregressive transformers with the objective of predicting the next token similar to traditional models for

natural languages (Vaswani et al., 2017). Along with the models, the authors have released a multi-turn programming benchmark that assists in measuring the capacity of a given model in regard to multi-turn program synthesis. The experimental results show that the multi-step program synthesis capacity of a model is positively associated with its size.

More recently, the work in (Madaan et al., 2023) proposed the PIE dataset. PIE is a subset of the CodeNet (Puri et al., 2021) collection of code samples. It consists of trajectories of programs, where an individual programmer starts with a slower version of a program and makes changes towards improving its performance. The authors of PIE have used it to evaluate and improve the capacity of multiple variants of models from CODEGEN family. In particular, they fine-tuned these models to suggest faster versions of a given piece of code. Furthermore, they evaluated their approach by training OpenAI's CODEX using few-shot learning. Their evaluation results exhibit a speedup improvement of up to 2.5 for more than 25% of the test programs.

A different set of modern approaches to code generation builds on the utilization of LLMs and RL. In general, the essence of incorporating RL techniques is mostly to ensure the functional correctness of the produced program (Liu et al., 2023). For instance, CodeRL (Le et al., 2022) combines pre-trained models with DRL for program synthesis in order to generate a program that fulfills a problem specification. It is an extension of CodeT5 that illustrates improved learning objectives and has more parameters along with better pretraining data. In CodeRL, the trained language model is used as an actor network. A critic network is also incorporated, aiming to predict the functional correctness of the generated code and provide feedback to the actor. The authors also introduce a critical sampling strategy that enables a model to regenerate programs by taking into consideration feedback from unit tests and critic scores. The evaluation results demonstrate that CodeRL can achieve state-of-the-art performance on APPS benchmark (Hendrycks et al., 2021).

The work in (Shen et al., 2023) introduces the 'Rank Responses to align Test&Teacher Feedback' (RRTF) framework. It also presents an LLM for programming languages, namely PanGu-Coder2. The main model is trained by ranking the candidate pieces of code using feedback from test cases and other heuristic preferences. Various experiments on the HumanEval, CodeEval and LeetCode benchmarks indicate that PanGu-Coder2 can reach state-of-the-art performance.

Shojaee et al. (2023) introduced PPOCoder, a task and model agnostic framework that can be used for a variety of code generation tasks. PPOCoder fuses pre-trained language models and Proximal Policy Optimization (PPO) (Schulman et al., 2017), a widely used deep RL technique. PPOCoder takes into account feedback from the compiler and unit tests

accompanied by syntactic-related feedback. This fosters the model to generate better code in terms of syntax and logic. The experimental results point out that PPOCode is more effective than its baselines with regard to the syntactic and functional validity of the generated codes.

Although PPO is one of the most popular policy gradient methods for RL, a list of alternative algorithms has been proposed in the literature. The purpose of these alternatives is to avoid the disadvantages of PPO (e.g. sensitivity to hyperparameters), while also reducing the overall algorithmic complexity. More specifically, Preference Ranking Optimization (PRO) (Song et al., 2023) adopts the Bradley-Terry comparison from 'Reinforcement learning from human feedback' (RLHF) (Stiennon et al., 2020; Xue et al., 2023). It also aligns the probability ranking of the responses generated by an LLM with the preference ranking by humans. The conducted experiments suggest that PRO outperforms its counterparts by effectively aligning LLMs to human preferences. Lately, RRHF, a promising algorithm has been introduced (Yuan et al., 2023). In a similar fashion to RLHF, it supports the alignment of LLMs with human preferences. The authors argue that RRHF can be easily tuned and achieve a performance comparable to that of PPO in the Anthropic's Helpful and Harmless dataset (Bai et al., 2022).

3 PROBLEM DEFINITION

We consider the problem of generating optimized versions of an input code. More formally, given a set of input programs X, the task is to generate a set of optimized programs $\hat{X}$, for each $x \in X$. The optimized program versions should take the same input and produce the same output as their original versions.

To do so, for each $x \in X$, we generate a set of candidate programs $y \in Y$, using a sampling strategy $s \in \{\text{greedy}, \text{random}\}$. Then our goal is to maximize the cost function:

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

where $y_{best} \in Y$ is the best candidate, the term $\text{eq}(R, y_{best})$ measures the ability of a generated sequence with a given input to match the output of the unit test, and the term $\text{perf}(y_{best})$ measures how the performance improvement of y_{best} over x for unit tests.

At the same time, we also aim to maximize the probability of generating y_{best} from the distribution of the input program by learning from existing training scripts.

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

Here, θ^* represents the optimal set of model parameters. Since recent solutions to the problem rely mostly on LLMs,
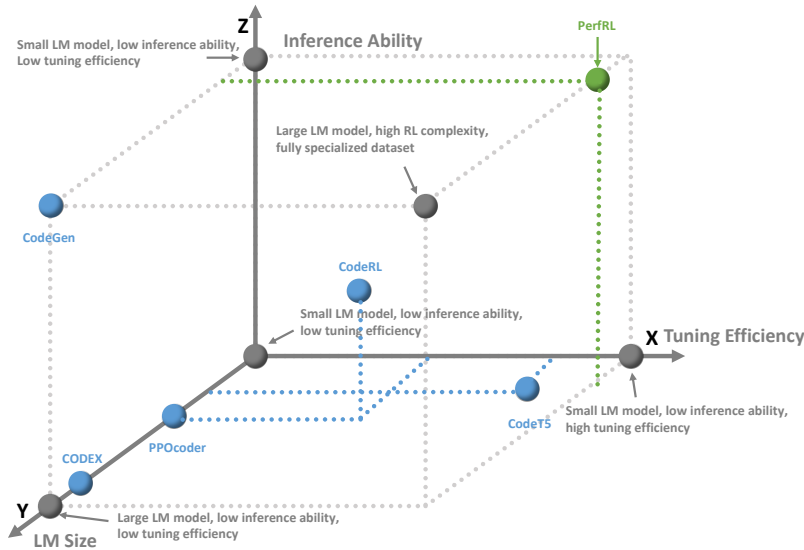
Figure 1. The problem of code optimization is seen from three different perspectives, i.e. the size of the LM (Y-axis), the inference ability of the model (Z-axis), and the tuning efficiency (X-axis). Our approach (the green dot) uses a fully specialized dataset (PIE), an LM model with a medium size (CodeT5), and has a low inference ability. LMs with a big size include those of the CodeGen family and Codex.

the main challenge of this problem is the fact that LLMs cannot naturally interact with their environment. As a result, they can potentially produce optimized versions of a given code that look correct superficially but contain either syntactical or logical errors. Additionally, existing solutions that build on LLMs and RL do not concentrate on the task of code optimization. Thus, they cannot perform sufficiently well and may suffer from hallucination, i.e. fabricating non-compilable meaningless pieces of code. (Zhang et al., 2023). A different challenge of this problem is the ever-increasing size of the required models. These models require a sufficient amount of computing resources, which in turn results in an increment of energy consumption.

Considering the challenges above, we propose a different approach to train an LLM model for the code generation problem. Instead of using solely an LLM or building a general-purpose LLM model with some RL steps, we design a framework targeting the task of code generation (see Section 4.3). Figure 1 highlights the different perspectives of the code optimization problem and how our framework differs from the existing ones.

As Figure 1 illustrates, we can perceive the problem of code optimization from three different perspectives — depending on the complexity of the RL algorithm, the size of the LM, and how relevant the used dataset with the task of code optimization. For instance, in our experiments, we use a medium-sized LM (CodeT5), the RRHF RL algorithm that has a medium complexity and a dataset (PIE) that is specialized in the considered task. However, the proposed

framework can facilitate different settings and combinations of the size of the selected LM model, the complexity of the RL algorithm and the relevance of the used dataset.

Some of the research questions associated with the problem of code generation are the following:

- How can we reduce the size of the models needed to generate optimized versions of a code?

- How can we ensure that the generated codes are reliable, produce the expected results and are free from syntactical or logical errors?

4 PROPOSED APPROACH

We propose PerfRL, a reinforcement learning-based LLM framework for code performance optimization. PerfRL advances the capability of LLMs to generate optimized code that improves program runtime, while also being logically and syntactically correct. To do so, it leverages techniques from LLMs and RL. It consists of three main components: (1) fine-tuning of the LLM model (Section 4.1), (2) sample generation, (Section 4.2), and (3) reinforcement learning supervision and correction (Section 4.3).

Figure 2 illustrates the architecture of the proposed approach. We thoroughly describe each of the steps in this section. During the training phase, we first fine-tune an LLMs using a dataset specialized in the task of code generation. Then, we utilize the fine-tuned model to generate optimized code versions for a given input code by employing different sampling

strategies, as described in Section 4.2. We then calculate a reward value and a score for each generated code and calculate a loss value based on an RL technique (Section 4.3). Then, during inference, we use the fine-tuned LLM model to generate an optimized version for each one of the given codes.

4.1 Fine-tuning of the LLM model

The first step is to fine-tune an LLM model on a dataset that is specialized in the task of code optimization. For efficiency and simplicity, in our experiments (see Section 5), we use the lightweight CodeT5 model (Wang et al., 2021), but our framework can operate with a variety of LLMs, regardless of their size. In the initial paper of CodeT5, the authors use either natural language (NL) or a combination of natural and programming language (PL) as input. Particularly, they pre-train their model on tasks such as Identifier-aware denoising, Identifier Tagging, Masked Identifier Prediction and Bimodal Dual Generation. Thus, we believe that this particular kind of models is more capable of understanding NL-PL inputs. As a result, we follow an NL-PL approach to feed the input into the CodeT5 model. Meanwhile, the authors in (Madaan et al., 2023) argue that a few-shot sampling strategy is beneficial for producing an optimized output. Therefore, we concatenate the action's natural language of asking the model to improve the execution performance with the input programming language and feed them into the model.

The CodeT5 model expects as input and target the preprocessed slower and faster code queries, respectively. The objective of fine-tuning is to minimize the cross-entropy loss:

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

where θ is the parameter of the given LLM (in our case, CodeT5), N is the number of tokens, V is the vocabulary set of the tokenizer, y is the embedded value of the j-th token in the vocabulary at position i in the true output sequence, and $p_{i,j}$ is the predicted probability for the j-th token in the vocabulary at position i.

4.2 Sample Generation

To generate candidate samples of code, we employ different sampling strategies during training, testing, and validation. More specifically, we use greedy sampling and random sampling.

Greedy sampling with beam search generates B number of distinct samples for the same input. For each step of the sequence generation, the model takes the current generated

sequence of tokens of sample k and computes the probability distribution of the next token over the vocabulary for the corresponding sample. The top B candidate vocabulary and its cumulative probability are calculated and ranked for each sample. The top B sequences among all candidates with the greatest cumulative probability are selected to repeat the process of generating the next token.

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

where x is the input of the model, $Y = (y_1, \ldots, y_T), y_t \in V$ is the output tokens from the model $c \in [0, B-1]$.

The second sampling strategy is random sampling, which generates a number of distinct samples with the same input. During the generation, we set a temperature value Tem in order to affect the diversity of the output. Given the probability distribution of the given logits l, the scaled l' equals to:

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

$$l' = \frac{l}{Tem} \qquad (6)$$

The probability of each token after scaling is:

$$p_i = \frac{exp(l_i')}{\sum_i exp(l_i')} \qquad (7)$$

Based on p_i, we select the top-k tokens and randomly pick one of them. We repeat such sampling steps for t times until reaching the max length or end conditions.

The next step is to generate the code samples. We observed that the model is unable to learn from most of the initially generated samples of code since they have a syntactical or logical error; as a result, the code samples do not pass the unit test. Thus, to generate samples of code during training, we first apply random sampling and select two candidates from independent runs. Then, we perform one greedy sampling to find the candidate with the highest probability. Finally, to ensure that at least one correct sample exists, we include the target sequence from the dataset in the list with the samples. These four samples are then fed to the model on each step.

During validation and testing (i.e. inference), we generate 4 samples using greedy sampling with beam search and return the top-2 best candidates. For a given input, we generate two candidates for evaluation. Similar to (Lu et al., 2021), we consider a sample as successful, when it has a better execution time compared to the input code.

4.3 Reinforcement Learning

Our RL step builds on RRHF, which is a lightweight RL framework for tuning LLMs with feedback scores. During
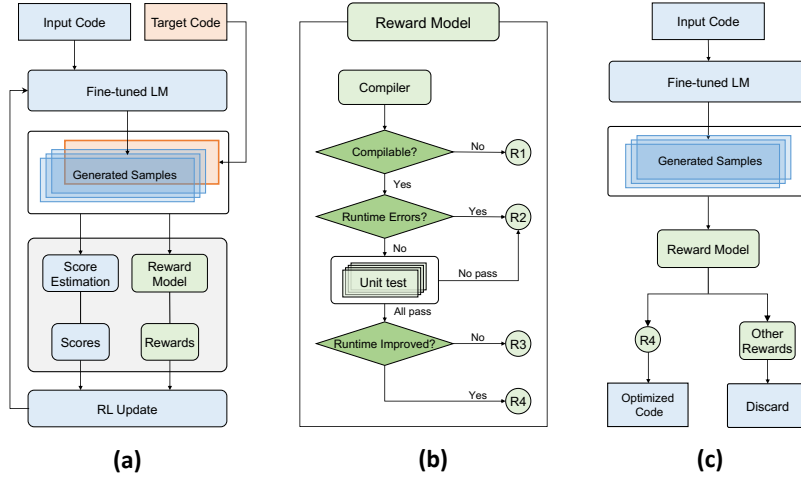
Figure 2. Overview of the PerfRL framework. **(a) Training.** We first fine-tune an LLM model using the whole training dataset. Then, we pass the input codes into the fine-tuned LM to generate a predefined number of optimized samples for each input program. We assign a score value to each sample and calculate its reward using the reward model. We utilize the score and reward values to calculate the L_{rank} and L_{tuning} loss values for RL. The final loss L is calculated by combining the two previous loss values and is used to retrain the model. In that way, our framework incorporates feedback from unit tests into its training process. Thus, it is more likely to generate optimized code that is free from syntactical and logical errors, mitigating hallucinations. **(b) Reward model.** Depending on the status of the code (e.g. can be compiled, has a run-time error or passes all the unit tests) a different reward value is given by the reward model. **(c) Inference.** During inference, the final LLM is utilized to generate multiple samples of candidate source codes. These generated source codes are then evaluated using the reward model. All samples that do not receive an R4 reward are filtered out. The source codes that remain after this elimination process are considered as the optimized code.

each RL step, our objective is to maximize the probability of generating highly rewarded pieces of code by ranking and fine-tuning the LLM model. For each RL step, we sample all the data from the training set and generate the candidate outputs. Finally, for each candidate output, we compute the score, reward and loss in order to tune the model parameter θ.

4.3.1 Reward

After the code generation step, we execute each sample of code with a Python interpreter. If an error is detected, an error message is displayed. If a piece of code does not have syntax errors, we test it for logical errors using the associated unit tests on a single core. The execution time et_o for an input code o is measured during the execution, assuming that there are no runtime errors within a predefined time period ($et_o \leq timeout$). In a similar fashion to the reward function proposed in (Le et al., 2022), for each

sample $y \in Y$, the reward $r(y)$ is calculated as follows:

$$
r(y^{(c)}) = \begin{cases} R1 & \text{if } y^{(c)} \text{cannot be compiled} \\ R2 & \text{if } y^{(c)} \text{ run-time error, timeout, or failed} \\ & \text{any unit test} \\ R3 & \text{if } y^{(c)} \text{passed all unit tests} \\ R4 & \text{if } y^{(c)} \text{passed and improved run time} \end{cases}
$$

(8)

As described in the paper of RRHF (Yuan et al., 2023), the reward values, which are assigned to the different code samples, are irrelevant to the computation of the loss, as long as a more desirable result is associated with a higher value (see Section 4.3).

4.3.2 Score Function

For a given generated code x, we have a candidate sequence $y^{(c)}$, where $0 < c < B$. For each $y^{(c)}$, we compute the predicted score of the sequence as the sum of the log probability of each token divided by number of tokens t:

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

(9)

4.3.3 Minimizing the Probability of Less Rewarded Outputs

From the execution of the evaluation system, we obtain our reward $r^{(c)} = r(y^{(c)})$ for each $y^{(c)}$ with a given input x. We maximize the loss of correct responses and minimize the wrong responses by:

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

4.3.4 Fine-tuning loss to maximize the best-rewarded candidate

Since our approach is based on RRHF, we calculate the cross-entropy for the best response similar to fine-tuning:

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

where y_{best} has the greatest $r(y_i)$ among all $0 < i < k$.

In that way, PerfRL can continuously enhance its output, even in cases where the best-generated code contains syntax errors or does not have the best execution time. As we only compare the candidate outputs based on the execution time of the input program, it is highly likely that two candidate output codes will have the same reward. The loss L_{tuning} selects the y_{best} based on the sampling strategy. To foster the model's propensity for independent discovery of optimization strategies, we have calibrated its learning priorities. The highest priority is assigned to learning from random samples, followed by a preference for greedy samples. The target program is designated as the final learning priority.

4.3.5 Loss

We sample all the input data and calculate the loss L as a combination of L_{rank} and L_{tuning} for the samples from the same input.

$$L^z = \left(aL_{\text{rank}}^z + L_{\text{tuning}}^z\right) \qquad (12)$$

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

where X is all the input prompts from the dataset and a is a constant.

5 EXPERIMENTS

5.1 Dataset

To fine-tune and evaluate PerfRL, we use the dataset from PIE (Madaan et al., 2023). This dataset consists of approximately 40k Python files, 88k C++ files, and 3.6k Java files. PIE captures the progressive changes made by a programmer

over time to improve their code. The dataset also contains (slow, fast) pairs of code written by the same programmer. We run our experiments on the subset of the dataset that concerns the Python files. The training, test and validation sets consist of approximately 36k, 1k and 2k samples respectively. Each of the samples has at least one associated unit test file that requires a specific input and has an expected output result. We test the accuracy of the dataset by executing all the input source code with a 5-second execution limit. We found that the 72.4% of the training data, 76.4% of the testing data, and 70.8% of the validation data are executable. During our training, we skip all the input code that is not originally executable in the first place. As for testing and validation, we use all the data in order to compare our approach with the baseline model.

5.2 Setup

We run the fine-tuning and reinforcement learning steps for an instance of the CodeT5 model with 60 million parameters and a learning rate of 2×10^{-5}. To reduce the training time we run the whole process for 8 RL steps for 3 epochs. For the training of the model, we use an NVIDIA A100 GPU graphics card with 40GB of RAM on an Ubuntu 20.04 server with 50 cores. Our model is trained within approximately 30 hours. Each epoch computes (i) the score of the generated sequence (see Equation 9) and (ii) the loss for all the samples of the dataset from the same input data. During the training phase, we set the temperature to 1 and top_k to 50 for random sampling.

As already mentioned, prior to running the reinforcement learning step, we fine-tune the CodeT5 model with a one-shot learning setting. That is to say, we feed each sample of the dataset into the model once and compute the loss as described in Equation 3. We set the learning rate to 5×10^{-5} and batch size to 32. The results are available in Table 1.

During the validation process, we use greedy sampling with a beam search of size 4 for the input code to generate candidate samples. From these 4 samples, we choose the top 2 ones with the greatest accumulated probability. Then, we set a reward value to the greedy sampling round using the reward function r. The validated compilation rate measures the number of rounds that pass the compilation over the total number of input codes. The validated pass rate measures the number of rounds that pass the execution of all the unit tests over the total number of input codes. The validated optimization rate measures the number of rounds that pass the execution and have a better execution time in a single core over the total number of input codes.

We also test our approach on 1000 samples. In the entire process, we run the test before the RL framework to test the current performance of the fine-tuned model and after the RL framework in order to show the contribution of RL.

Table 1. Evaluation results of PerfRL and baseline models. The bold font indicates the evaluation results of PerfRL. The asterisk (*) indicates the baseline model. The first block shows the results as reported in the paper of PIE (Madaan et al., 2023)

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**

For our experiments, we set the reward values for the reward function r to $R1 = 0$, $R2 = 1$, $R3 = 1.3$ and $R4 = 2$. As already mentioned in Section 4.3.1, the actual reward values do not affect the performance of the RL step, as long as we assign a higher value to a much preferable sample.

5.3 Baselines

We use a fine-tuned version of the CodeT5 model with 24 epochs as our main baseline. Furthermore, we compare our results with those of the original PIE paper, where models from the family of CodeGen and Codex are used for their experiments. We benchmark our approach against the main baseline model using the evaluation metrics described in Section 5.4.

5.4 Evaluation Metrics

We use the following evaluation metrics as also defined in the paper of the PIE (Madaan et al., 2023) dataset:

- **Percent Optimized (%OPT)**: The ratio of samples on the test set that are improved by a given method.

- **Speedup (SP)**: The actual (absolute) improvement in execution time $SP(o, n) = (\frac{o}{n})$, where o and n are the old and new execution times, respectively.

- **Runtime Reduction (RTR)**: The normalized improvement in execution time among the programs that have a decrease in runtime and are syntactically and logically correct, $RTR(o, n) = (\frac{o-n}{o} \times 100)$. We mention that the average RTR is reported over the test set.

To measure the execution time of each generated program, we calculate the cumulative execution time using all the unit tests. We run each experiment three times and measure the mean time on each set of unit tests in order to ensure that the reduction of the execution time is not affected by random factors.

5.5 Evaluation Results

Table 1 presents the mean %OPT, SP, and RTR scores of the proposed method as well as that of the baseline model. Although PerfRL relies on a model with fewer parameters, it manages to perform equally or even better with respect to the greedy and 1-shot versions of the CodeGen-16B and CodeGen-2B models. Furthermore, as illustrated in Figure 3, both the pass and optimization rate increase proportional to the RL steps. Thus, we argue that the utilization of an RL step enables smaller models to learn easily the task of source code optimization in a less resource and time-consuming manner. More specifically, we train our model for 30 hours as opposed to the baseline CodeGen which is trained for approximately 3 days and requires a stronger machine (i.e. $2 \times$ NVIDIA A6000 for CodeGen-2B and $4 \times$ NVIDIA A6000 for CodeGen-16B).

In order to determine whether our RL step is able to generate good candidates by random or greedy sampling during the training, we measure the compilation, pass, and optimization rate as shown in Figure 4. Considering our sampling strategy that concatenates the target program with the generated samples, we empirically observe that the threshold for these three values is approximately 20% (red dashed line). Since these three rates are always above the threshold, they denote that our model generates meaningful candidates by itself other than purely based on the target program.

5.6 Discussion

Our framework shows that an RL-based strategy for fine-tuning is able to tune a model more effectively compared to simple fine-tuning. Especially, for source code optimization, it is crucial for the model to have the ability to explore the search space by itself. However, codes that are functionally equivalent are hard to learn if their semantics are drastically different for LLM tokenizers to understand the structure of the code on limited data. Our approach encourages LLMs to make minor modifications to the input source code. However, we believe ML techniques with structure information
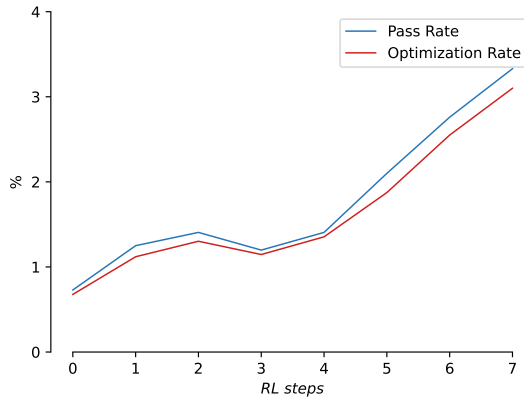
Figure 3. Pass rate and optimization rate on validation data over RL steps on the fine-tuned CodeT5 model.
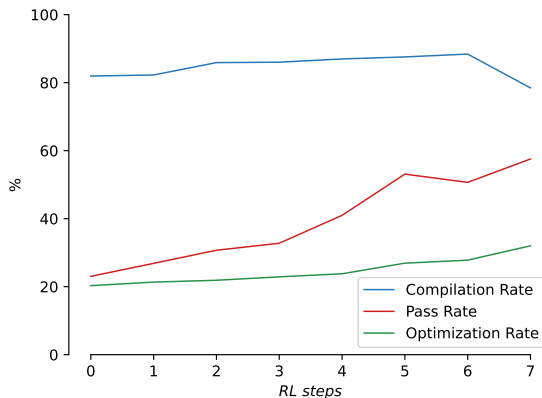

Figure 4. Compilation, pass, and optimization rate per RL step for the generated programs using the fine-tuned CodeT5 model. All the rates are calculated by the number of compiled, passed, or optimized generated programs over the total number of the generated programs.

of code can improve the ability of LLMs to understand the complicated structure difference between two source codes.

6 CONCLUSION

In this paper, we propose a novel framework for the task of code optimization, called PerfRL. Our framework combines techniques from LLMs and RL, and it allows language models to take into consideration feedback from unit tests during their fine-tuning process. To demonstrate the applicability of our framework, we fine-tuned the CodeT5 model on the PIE dataset. We benchmark the proposed approach against a list of baseline models that rely solely on simple fine-tuning, while ignoring logical and syntactical correctness of the generated code. The evaluation results demonstrate that by adopting our framework, one can reach state-of-art performance using a smaller language model with fewer parameters, which in turn results in lower energy consumption, something that is critical on edge computing devices.

Future work directions include (i) the integration of graph representations for code such as ProGraML (Cummins et al., 2020), (ii) the combination of graph neural networks and LLMs to capture both structural and language characteristics of the different parts of code, (iii) the implementation of more sophisticated scoring and reward functions for RL, (iv) the investigation of the applicability of the suggested approach to edge computing systems with strict power or energy budgets (v) the evaluation of our framework using models with different architectures but with a similar number of parameters.

REFERENCES

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., Hernandez, 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. Training 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., Bavarian, 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 programming. 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 SIGPLAN International Workshop on Machine Learning and Programming Languages*, MAPL 2018, pp. 69–80, New York, NY, USA, 2018b. Association for Computing 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: Evaluating 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 pretrained models and deep reinforcement 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 synthesis. *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 reinforcement 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 models for code understanding and generation. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pp. 8696–8708, Online 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.