LLaMoCo: Instruction Tuning of Large Language Models for Optimization Code Generation

Zeyuan Ma, Yue-Jiao Gong, *Senior Member, IEEE*, Hongshu Guo, Jiacheng Chen, Yining Ma, Zhiguang Cao and Jun Zhang, *Fellow, IEEE*

***Abstract*—Recently, combining the strength of large language models (LLMs) and Evolutionary Computation (EC) has shown promising results for addressing optimization problems. It typically involves either iterative next-step solution seeking or directly prompting LLMs to generate critical optimization codes. However, these methods often suffer from low computational efficiency, high sensitivity to prompt design, and a lack of domain-specific knowledge. We introduce LLaMoCo, the first instruction-tuning framework designed to adapt LLMs for solving optimization problems in a code-to-code manner. LLaMoCo features a comprehensive instruction set that includes code-style problem descriptions as input prompts and robust optimization codes from expert EC optimizers as target outputs. We then develop a novel two-phase learning strategy with a contrastive learning-based warm-up to enhance convergence during instruction tuning. Extensive experiments demonstrate that a CodeGen (350M) model tuned by our LLaMoCo yields a powerful domain-specific model for generating high-performance optimizers, achieving superior performance compared to GPT-4 family and other competitors on both synthetic and realistic problem sets.**

***Index Terms*—Large Language Model, Evolutionary Computation, Black-Box Optimization, Code Generation.**

I. Introduction

OPTIMIZATION plays a key role in both scientific and engineering scenarios [1], [2]. Evolutionary Computation (EC) is widely recognized as an effective gradient-free optimization paradigm for addressing various optimization problems [3]. So far, most optimization problems from scientific or engineering scenarios are solved by hand, requiring expert knowledge to 1) formulate the problem and 2) solve the problem by selecting and/or configuring an optimizer. This in turn hinders the widespread of optimization techniques such as EC. Nowadays, Large Language Models (LLMs) are posing a profound impact on human society [4], [5] through their remarkable natural language understanding and ability to solve complex tasks [5], [6]. Optimization is inherently related to language understanding and LLMs [7], as solving an optimization problem requires us to first describe and comprehend it, and then implement a solution using a programming language. This raises a key research question: *Can LLMs tackle challenging optimization problems that are difficult for humans to address?* This paper presents a comprehensive exploration of this question.

In the literature, several works have explored the possibilities of using LLMs to solve optimization problems. A common and straightforward way is to let LLMs act as EC optimizer, which is iteratively prompted to generate better solutions through a multi-turn conversation process [8]–[10], sometimes incorporating the concept of in-context learning. This solution-to-solution process involves prompting the LLMs with initial or current best-so-far solutions and iteratively requesting improved solutions. While showing certain effectiveness, they can have several limitations: 1) the scale of target optimization tasks (e.g., in terms of the number of decision variables, historical solutions and newly generated solutions) is constrained by the context window length of LLMs; 2) the iterative process typically involves hundreds rounds of conversations, consuming significant resources or API callings; and 3) due to LLMs' sensitivity to prompt design, it is nontrivial to provide consistent prompts that ensure ideal outputs.

An alternative way is directly prompting LLMs to generate optimization programs [7], [13]. It can be more efficient than the solution-to-solution methods for two reasons: 1) only a few rounds of conversation are needed to generate codes; and 2) the prompts and generated codes do not include solution information, making it compatible with the problem scales. However, careful prompt crafting is crucial to ensure logical coherence. For example, OptiMUS [7] integrates hints about the target optimizer into the prompts, requiring a deep understanding of optimization domain knowledge. Additionally, LLMs pre-trained on a broad corpus often fail to generate customized optimizers tailored to a specific optimization problem instance due to the *lack of domain-specific expert knowledge* [14]. Existing code LLMs, such as Codex [15], StarCoder [16], CodeGen [17] and Code Llama [18], are trained on vast, general-domain code corpora, focusing on the general-purpose programming tasks in software development.

In this paper, we propose **LLaMoCo**, a novel framework that adapts general-purpose **L**arge **La**nguage **M**odels for **o**ptimization **Co**de generation. Unlike methods based solely on prompt engineering, LLaMoCo instruction-tunes general (code) LLMs as domain-specific models to generate well-

This research is supported in part by National Natural Science Foundation of China (Grant No. 62276100), in part by the Guangdong Provincial Natural Science Foundation for Outstanding Youth Team Project (Grant No. 2024B1515040010), in part by Guangzhou Science and Technology Elite Talent Leading Program for Basic and Applied Basic Research (Grant No. SL2024A04J01361), and in part by the Fundamental Research Funds for the Central Universities (Grant No. 2025ZYGXZR027). This research is supported by the National Research Foundation Singapore under the AI Singapore Programme (AISG Award No: AISG3-RPGV-2025-017).

Zeyuan Ma, Yue-Jiao Gong, Hongshu Guo and Jiacheng Chen are with the School of Computer Science and Engineering, South China University of Technology, China (E-mail: gongyuejiao@gmail.com)

Yining Ma is with Massachusetts Institute of Technology, United States. (E-mail: yiningma@mit.edu).

Zhiguang Cao is with Singapore Management University, Singapore. (E-mail: zgcao@smu.edu.sg).

Jun Zhang is with Nankai University, Tianjin, China and Hanyang University, Seoul, South Korea. (E-mail: junzhang@ieee.org).

Corresponding author: Yue-Jiao Gong and Yining Ma.

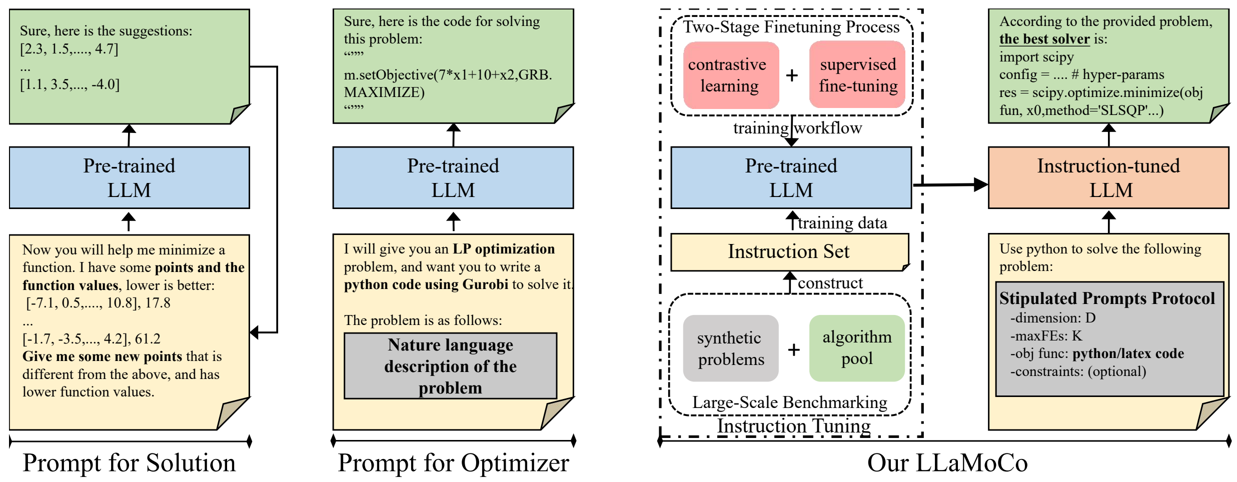

Sure, here is the suggestions:
[2.3, 1.5,...., 4.7]
...
[1.1, 3.5,..., -4.0]
Pre-trained LLM
Now you will help me minimize a function. I have some points and the function values, lower is better:
[-7.1, 0.5,...., 10.8], 17.8
...
[-1.7, -3.5,..., 4.2], 61.2
Give me some new points that is different from the above, and has lower function values.
Prompt for Solution
Sure, here is the code for solving this problem:
""""
m.setObjective(7*x1+10+x2,GRB.MAXIMIZE)
""""
Pre-trained LLM
I will give you an LP optimization problem, and want you to write a python code using Gurobi to solve it.
The problem is as follows:
Nature language description of the problem
Prompt for Optimizer
Two-Stage Finetuning Process
contrastive learning
supervised fine-tuning
training workflow
Pre-trained LLM
training data
Instruction Set
construct
synthetic problems
algorithm pool
Large-Scale Benchmarking
Instruction Tuning
According to the provided problem, the best solver is:
import scipy
config = ... # hyper-params
res = scipy.optimize.minimize(obj fun, x0,method='SLSQP'...)
Instruction-tuned LLM
Use python to solve the following problem:
Stipulated Prompts Protocol
-dimension: D
-maxFEs: K
-obj func: python/latex code
-constraints: (optional)
Our LLaMoCo

Fig. 1. Conceptual overview of *LLMs for optimization*. **Left**: optimization through iteratively prompting for better solutions (e.g., [8]). **Middle**: optimization through prompting LLMs for optimizer configuration or code generation (e.g., [7], [11], [12]). **Right**: our LLaMoCo learns a domain-specific optimization model that generates complete, instance-level optimizers' code from formatted prompts, tuned with a comprehensive instruction set.

performing optimizers (e.g., EC optimizers) tailored at the instance level. LLaMoCo offers several unique advantages. Figure 1 depicts the difference between our LLaMoCo and existing approaches. First, it generates optimization programs in a single round, making it more efficient for large-scale problems. Second, we standardize problem description prompts into structured codes. This approach not only minimizes prompt engineering effort but also enhances the models' ability to learn underlying patterns for optimization problem solving. As a consequence, LLaMoCo is able to generate optimizers that are highly competitive with those produced by leading general-purpose LLMs, achieving superior optimization performance and strong zero-shot generalization. However, building such a specialized framework presents fundamental challenges that are central to our research motivation. Two key issues must be solved:

- The Knowledge Gap: How can we gather and structure the vast, specialized knowledge of optimization problems and high-performing solvers into a format suitable for instruction tuning? To address this, we develop a systematic popeline to generate the instruction set. First, a novel synthesizing procedure automatically generates sufficient and diverse synthetic problem instances. Next, we design a universal question-answer template, combined with coding pattern-based augmentation strategy, to scale a well-formatted instruction set. Finally, we obtain a large-scale set of code-to-code examples, each consists of a problem description in Python or LaTeX and the corresponding high-performing optimizer implementations, validated through extensive benchmarking and hyper-parameter tuning.
- The Tuning Inefficiency: How can we fine-tune a general model on this new data effectively, without requiring prohibitive amounts of training? To solve this, we propose an efficient contrastive warm up learning strategy, which accurately addresses the semantic alignment issue in the training data. Combining the contrastive warm up with the normal supervised fine-tuning process creates an effective & efficient two-phase training workflow.

Extensive experiments show that instruction tuning a small LLM like CodeGen-350M [17] on domain-specific tasks significantly outperforms even very large models like GPT-4 [19]. Further ablation studies provide in-depth analysis on the two-phase adaptation strategy, training data sensitivity, and zero-shot generalization performance. More importantly, as the first exploration in fine-tuning LLMs for optimization, we further investigate several key aspects in the training such as the model's scaling law, generalization potential and knowledge gap in mainstream LLMs. We believe these valuable observations could provide guidance for future works in this filed.

Our contributions are four folds: 1) we introduce LLaMoCo, the first framework for adapting general LLMs to generate competitive optimization programs; 2) we establish a code-to-code instruction set tailored for the optimization domain; 3) we propose an effective two-phase instruction-tuning strategy based on contrastive warm-up learning; and 4) our LLaMoCo exhibits superior zero-shot generalization, efficiency, and robustness to a wide range of related baselines.

II. Related Works

A. Instruction Tuning LLMs

Pre-trained LLMs can be refined by parameter updates on specific tasks through a fine-tuning process. We introduce two prominent fine-tuning strategies: Instruction Tuning (IT) [20] and Alignment Tuning (AT) [21], [22], each serving distinct purposes. We adopt IT in this paper as the major fine-tuning strategy. Generally, IT involves fine-tuning pre-trained LLMs using a moderate collection of formatted task instances [23]. The fine-tuning process typically includes two steps: 1) prepare instruction-formatted training examples by associating a task description with each task instance, which aids LLMs in understanding tasks [24]; and 2) leverage the prepared instruction set to fine-tune LLMs using a sequence-to-sequence supervised loss [25]. By incorporating a well-established task-specific instruction set, IT can effectively inject domain-specific knowledge into general LLMs. It enables the transfer of LLMs to specific experts in domains like medicine [26], law [27] and finance [28]. Our LLaMoCo is the first instruction-tuning framework for adapting general LLMs as an efficient and effective optimization tool, which addresses the unsatisfactory optimization performance due to the limited domain-specific knowledge of general LLMs.

B. LLMs for code generation

Generating code from natural language descriptions is exciting and complex [29], [30]. Although general-purpose LLMs such as GPT [31], Llama 2 [32] and Mistral [33] show competitive performance on the widely used LLM benchmarks including HumanEval [30], MBPP [34] and DS-1000 [35], their performance on a particular task may still be limited. Recent efforts have focused on developing LLMs specifically tailored for code generation. These models can be trained exclusively on code, such as AlphaCode [36] and StarCoder [16], fine-tuned from general LLMs, like Codex [15] and Code Llama [18], or prompted from pre-trained LLMs such as FunSearch [11]. Notably, Codex shows that a 12B LLM can solve 72.31% of complex programming tasks posed by humans. This success has led to the emergence of various Code LLMs, such as CodeGen [17] that factorizes a potentially long specification into multiple steps to enhance program synthesis, and Code Llama that extends Llama 2 models through a cascade of fine-tuning steps. Other models such as Phi-2 [37], InCoder [38] and CodeGeeX [30] have also gained great attention. One potential technical bottleneck that might limits their domain-specific ability is that: The extremely large scale training corpus, while might includes some optimization programming cases, is still limited considering the portion of optimization-related scripts in the whole corpus. Diluted knowledge intensity pushes these code LLMs to focus more on general programming tasks. Such observation has been witnessed in latest literature [39], [40]. Given such strong motivation, we propose LLaMoCo as the first work to explore how to enhance general LLMs (including general code LLMs) with expert-level optimization knowledge.

C. LLMs as optimizers

Unlike tasks such as language understanding, optimization problems are difficult for humans to solve without efficient algorithms, challenging LLM's reasoning and generalization abilities. Recent research has explored *prompting LLMs for solutions* of the given numerical optimization [8], [41], [42], optimizer discovery [13], [43], [44], or prompt optimization [45] scenarios, which is based on prompt engineering and in-context learning [46]. Typically, these methods involve a set of candidate solutions for improvement, where LLMs receive prompts with these solutions and their objective values and then generate improved solutions iteratively until a termination condition is met. However, such a paradigm challenges the expertise of the general LLMs for optimization, which is less developed during their pre-training. To address this, the latest studies innovatively proposed prompting LLMs to behave like black-box optimizers, that is, instructing LLMs to perform mutation, crossover operations and elitism strategy [10], [47]–[49] on the candidate solutions. An eye-catching work of this line is EvoPrompting [48], which is surprisingly capable of finding neural network architectures with state-of-the-art performance. However, these approaches have limitations in efficiency due to the need for extensive iterations.

In contrast, several studies explore other possibility: *prompting LLMs directly for pieces of executable optimization programs*. An emerging research area is LLMs for Algorithm Design [43] in the EC domain, that is, applying LLMs at the algorithm design level to search and refine algorithm programs in an EC fashion. By evolving the programs of constructive methods, the finally obtained optimization program could beat certain human-crafted heuristics. OptiMUS [7], on the other hand, integrates hints about the target optimizer into the prompts to create and test new optimizers for numerical optimization problems. Following the idea of Funsearch [11] and EoH [43], the concept of using LLMs for iterative (meta-) heuristic program search becomes more and more popular. Huang et al. [50] opensourced a comprehensive prompting strategy group, where they emphasized the importance of repair process in LLM generated program. Forniés-Tabuenca et al. [51] introduced reflection-based self-refinement mechanism into the interative prompt trajectory to ensure better optimization program can be attained. Liu et al. proposed a general development platform LLM4AD [52] to help practitioners in the related domain to learn, develop and evaluate. LLaMEA [53] identifies a key limitation in existing work: the reliance on the LLM-based textual mutations. They therefore combine an evolutionary algorithm-style loop with the LLM mutations, which could generate context-aware algorithm scripts in any length. Zhong et al. [54] proposed asking LLM within one conversation through a CRISPE principle. The optimizer generated by their method show several differences with existing EC optimizers, showing the design capability of LLMs.

More recently, researchers are increasingly recognizing that directly prompting general-purpose LLMs is insufficient for complex optimization tasks. The limited expert knowledge within these models necessitates fine-tuning to enhance problem solving abilities, yet relevant works remains scarce. While prior studies have explored instruction-tuning for operations research problems [55], [56], low-cost LLM-aided multi-objective optimization [57], and the use of LLMs as surrogate models in EAs [58], these efforts have been limited in scope. They have focused on discrete domains and/or component-level assistance, rather than the end-to-end generation of complete optimizers for continuous domain. This further underscores the novelty of LLaMoCo, the first framework designed to tune LLMs for this purpose, addressing both data preparation and training enhancement. There are also several valuable position papers [59]–[61] offer a broader perspective.

III. LLaMoCo

LLaMoCo is the first instruction-tuning framework enabling post-training general-purpose LLMs to generate complete instance-level optimizers. Achieving such code-to-code flexibility presents several significant challenges. First, data preparation requires innovative solutions for representing optimization problems, efficiently generating sufficient problem instances, and gathering high-quality optimization knowledge.

Next, instruction-tuning demands careful consideration of the unique semantic alignment issues and potential data imbalances within the optimization domain. Finally, since optimization code generation inherently combines the complexities of both optimization and code generation tasks, objectively evaluating methods in this domain presents unique challenges. In the subsequent sections, we detail the tailored designs to address these challenges.

A. Construction of Instruction Set

1) Synthetic instance generation: An optimization problem can be mathematically formulated as follows:

$$\begin{aligned} \min_{x} \quad & f(x), \\ \text{s.t.} \quad & g_i(x) \leq 0, \quad i = 1, \dots, M_g, \\ & h_j(x) = 0, \quad j = 1, \dots, M_h, \end{aligned} \tag{1}$$

where $f(x)$ is the objective function, $g_i(\cdot)$ and $h_j(\cdot)$ denote M_g inequality constraints and M_h equality constraints respectively. Without loss of generality, we assume a minimization problem where the optimal solution x^* attains the minimum objective value, adhering to all specified constraints.

The first concern is generating a sufficient number of high-quality and diverse problem instances for instruction tuning [24], [62]. As it is impractical to gather all types of real-world optimization problems, we generate synthetic instances that represent various problem landscapes. Specifically, we collect a basic function set $\digamma$ with various optimization problems and a basic constraint set Ω with various constraints from the well-known benchmarks [63]–[65]. Following [66], we synthesize a new objective function from K basic functions in $\digamma$ through two different paradigms as given by Eq. (2): 1) *Composition*: a linear combination of the K basic functions over the entire decision space, with each w_i uniformly sampled from $[0, 1]$. 2) *Hybrid*: The decision vector x is randomly decomposed into K segments (s_1 to s_K). Each basic function operates on one segment, and the final objective function is their summation.

$$\begin{aligned} \text{Composition: } f(x) &= \sum_{i=1}^{K} w_i \times f_i(x). \\ \text{Hybrid: } f(x) &= \sum_{i=1}^{K} f_i(x[s_i]). \end{aligned} \tag{2}$$

We then process each problem instance in three steps: 1) indicate the problem dimension D, the search bounds for each dimension (e.g., $-10 \leq x_i \leq 10$), and the number of basic functions K; 2) if $K = 1$, randomly select a basic function in $\digamma$ as $f(x)$, otherwise, we apply *Composition/Hybrid* paradigm to synthesize $f(x)$; and 3) randomly select a group of constraints $\{\{g_i\}, \{h_j\}\}$ in Ω. Note that step 3) is optional, as some optimization problems may not have constraints.

Composition and Hybrid complement each other to produce diverse functions in both decision and objective spaces. Hybrid combines sub-functions by assigning them to different decision variable sub-components, resulting in a decomposable (modular) structure. Composition, on the other hand, is mostly non-decomposable. It blends the objective spaces of multiple basic functions, carefully weighted and shifted, to create a smooth global landscape with diverse properties around different local optima. Together, they facilitate our primary goal in LLaMoCo: fine-tuning LLMs with problem solving knowledge as diversified as possible.

In this work, we generate 3k problem instances without constraints, denoted as P_{nc}, and another 3k problem instances with constraints, denoted as P_{c}. The complete set P is the union of P_{nc} and P_{c}, consisting of 6000 instances. These instances showcase different characteristics of global landscapes, including unimodal or multimodal, separable or non-separable, and symmetrical or asymmetrical. They also exhibit various local landscape properties, such as distinct properties around different local optima, continuous everywhere yet differentiable nowhere, and optima situated in flattened areas. This guarantees that the generated instances comprehensively mirror various realistic problems. It is worthy to note that the scope of this work is focused on single-objective optimization problems. Our primary goal is to investigate the fundamental challenges and core methodologies of adapting general (code) LLMs into expert optimization agents. Nevertheless, the proposed framework is inherently general and extensible to other domains, such as multi-objective and multi-task optimization, which we leave for future work.

2) Knowledge gathering: In our study, the term 'knowledge' refers to expertise in handling optimization problems, including selecting a well-performing optimizer and configuring its hyperparameters. To this end, we conduct exhaustive benchmarking to determine one effective optimizer for each instance $p \in P$. Concretely, we filter a wide range of optimizers from the literature [67], [68], competitions [64], [66], [69], and benchmarks [65], selecting 23 optimizers that span various algorithm families, including EC optimizers (e.g., GA [70]–[72], DE [73]–[76], PSO [77]–[79] and ES [80]–[83]), Bayesian Optimization [84], [85], Local Search strategies [86]–[88], and Numerical Optimization methods [89]–[92]. To determine the most effective optimizer for each instance p, we employ a two-step process.

For each optimizer $\Lambda_k \in \Lambda$, we span a grid-search configuration space C_k based on its tunable hyper-parameters, which is listed in Appendix A, Table II. Take DEAP-DE [93] as an example, it has three hyper-parameters and each of them has 5 optional values (pre-defined by us). We hence span the C_k of DEAP-DE as a configuration space comprising $5 \times 5 \times 5 = 125$ configurations, each denoted as C_k^j. We now establish the target of our benchmarking process:

$$\underset{\Lambda_k \in \Lambda, C_k^j \in C_k}{\arg\max} \ \mathbb{E}\left[eval(p, \Lambda_k, C_k^j)\right],$$

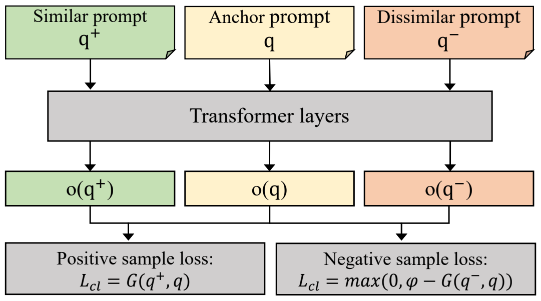
where p denotes the tested problem instance, $eval$ denotes the final optimization performance by calling Λ_k with configuration C_k^j to solve p, and $\mathbb{E}$ denotes the expectation of the optimization performance, which is unbiased-estimated by 5 independent runs in this work. For constrained problems, we benchmark Λ_c, while for unconstrained problems we benchmark Λ_{nc}. We note that the benchmarking process for each problem instance may encounter execution failures, e.g., some optimizers in Λ_c can not handle equality constraints, some optimizers in Λ_{nc} are incompatible with non-convex problems, BO optimizers are extremely time-consuming on high-dimensional problems. When failures occur, the corresponding $eval(p, \Lambda_k, C_k^j)$ is set to 0. After benchmarking Λ on P, we provide a configured optimizer $a(\Lambda_k, C_k^j)$, and the corresponding code implementation as the desired optimizer for each p. In Appendix E.1, we provide the validation of the quality of the found desired optimizers.

3) Instruction set construction: So far we have obtained a problem set P of 6000 optimization instances and their best-performing optimizer code. However, how to describe each instance $p \in P$ in a programming language is a challenge: while programming languages such as Python or LaTeX are specified, users formulate the same problem with diverse code due to their programming habits. This motivates us to augment each instance in P to diverse problem descriptions written in either Python or LaTeX. Training on such augmented data could help LLaMoCo output consistent optimization code even if its inputs are different description versions of the same problem instance.

To achieve this, we have conducted a survey among computer science students, and let them independently write problem descriptions for instances in P by both Python and LaTeX. The collected scripts are then fed into a general LLM for similarity check and coding pattern recognition, which results in several major programming patterns. These found patterns are then further undergone a human value consistency validation among a group of senior researchers in EC domain. By using these patterns, we create $4 \sim 6$ Python or LaTeX problem descriptions for each instance. A key advantage of such augment process is that it is explicitly designed to address noisy inputs. Using such training data with diverse coding habits facilitates generalization and robustness of LLaMoCo on realistic noisy inputs from diverse users. Such advantage will be validated in subsequent ablation studies. We provide a more detailed discussion in Appendix C.

After the augmentation, for each problem instance $p \in P$, we first insert each description of it into the prompt template to attain $4 \sim 6$ text prompts $\{q_{1,p}, q_{2,p}, ...\}$. We then insert the selected best-performing optimization code for p into the answer template to attain a text answer a_p. At last, for each p, we construct $4 \sim 6$ prompt-answer pairs $\{(q_{1,p}, a_p), (q_{2,p}, a_p), ...\}$. By repeatedly constructing prompt-answer pairs for all instances in P, we finally construct an instruction tuning set $\mathbb{I}$ comprising 32570 prompt-answer pairs. We note that such data scale conforms to common practices in existing instruction-tuning literature [94]–[97]. We also provide scaling law experiment in Section IV-D to further support this design choice.

4) Discussion: A key consideration in our study the trade-off between the quality and scalability of the instruction set [98]. As the first work that explores how to fine-tuning LLMs for end-to-end optimization workflow, the primary focus of this study is to ensure an effective and successful supervised fine-tuning process, where the quality of training data is especially important [99]. This leads to our decision to construct the algorithm pool and the synthetic problem set manually. By introducing certain expert knowledge of competitive optimizers and diverse optimization problem instances, the training effectiveness is ensured. Meanwhile, we also engineered significant automation into the pipeline, including the entire benchmarking loop, the prompt-answer pairing process, in order to establish a pricinpled and scalable framework.



Similar prompt q^+
Anchor prompt q
Dissimilar prompt q^-
Transformer layers
$o(q^+)$
$o(q)$
$o(q^-)$
Positive sample loss: $L_{cl} = G(q^+, q)$
Negative sample loss: $L_{cl} = max(0, \varphi - G(q^-, q))$

Fig. 2. The loss calculation in the proposed contrastive warm-up .

B. Two-Phase Instruction Tuning

1) Contrastive warm-up: Given the constructed instruction set $\mathbb{I}$, although we could naively apply regular LM loss to train an LLM to fit prompt-answer pairs in $\mathbb{I}$, the training convergence and stability suffer from two cases: a) there are 6000 problem instances involved, and they can be categorized into 23 classes, where the class label is the optimizer achieving best performance on each instance. This means that there is a chance two different problems hold very similar answers. b) for two different problems, the difference in their programming language description might be very small. For example, consider two problems with the same objective function one with a constraint and the other without, the difference in their code descriptions is only one line. However, the answer for the one without constraint should be the DE algorithm and for the other should be SLSQP. The above two cases would confuse the LLM during the fine-tuning.

Building on the practice of using contrastive learning for semantic alignment in code understanding [100], LLaMoCo introduces a tailored contrastive warm-up process before instruction tuning, offering a refined approach to the aforementioned challenge. As shown in Figure 2, to construct a mini-batch of training data, we first randomly select a prompt-answer pair (q, a) from $\mathbb{I}$, where q is regarded as the anchor prompt. For the positive sample, we randomly select a data pair (q^+, a) with the same optimizer type (we have 23 types). For the negative sample, we randomly select a data pair (q^-, a') with a different optimizer type. Then these three samples are applied to calculate the positive and negative sample loss, respectively. Specifically, for the decoder-only LLMs adopted for code generation tasks in this paper, we activate the Transformer layers [101] and regard the output embedding of the final self-attention block as a latent representation for a prompt. In LLaMoCo, we measure the distance between two prompts q and q', denoted as $G(q, q')$, by considering the cosine similarity between their latent representations $\overrightarrow{o}(q)$ and

$\overrightarrow{o}(q')$:

$$G(q,q') = \frac{1}{2}\left(1 - \frac{\vec{o}(q)\cdot\vec{o}(q')}{\|\vec{o}(q)\|\,\|\vec{o}(q')\|}\right). \tag{3}$$

The distance $G(q,q') \in [0,1]$. The contrastive loss of q and q', denoted as $L_{\text{cl}}(q,q')$, is calculated as:

$$L_{\text{cl}} = \begin{cases} G(q,q'), & q' = q^{+}; \\ \max(0, \varphi - G(q,q')), & q' = q^{-}, \end{cases} \tag{4}$$

where φ is a margin parameter. By minimizing L_{cl}, we could efficiently pull together the representations of two prompts which share the same desired optimizer yet have different forms, and vice versa. This contrastive phase is economic since we only consume a small number of epochs to warm up the fine-tuning of LLMs by L_{cl} and then instruction-tune the LLMs with the regular LM loss for next-token prediction [102]. We validate the effectiveness of this contrastive learning phase in Section IV-C.

2) Balanced data sampling: The instruction set $\mathbb{I}$ exhibits certain imbalance in the distribution of data. Notably, we observe that several optimizers dominate on thousands of problem instances, while the others only outperform on a few problem instances. Dealing with imbalanced data poses a challenge during the process of fine-tuning models [14], [103]. To address the issue, we follow the example-proportional mixing strategy [104] to re-balance the data distribution in $\mathbb{I}$. Each data pair (q,a) is sampled with a probability ρ as:

$$\rho(q,a) = \frac{1}{N_a \times N_{q,a}}, \tag{5}$$

where N_a denotes the number of optimizers in the gathered algorithm pool, $N_{q,a}$ denotes the number of instances whose desired optimizer is a. In this way, the number of sampled pairs dominated by each optimizer is approximately equal in each training epoch. Note that we apply this strategy in both the contrastive warm-up phase and the instruction tuning phase. The approach aids in avoiding biased training of the LLMs and enables them to effectively learn the knowledge from minority instances.

IV. Results and Discussions

A. Experimental Setup

1) Fundamental models & training settings: We adopt CodeGen-Mono (350M), Phi-2 (2.7B) and Code Llama (7B) as backbone models and fine-tune them on our instruction set. The reasons are two-fold: 1) these models show robust programming language reasoning and code generation ability, serving as a good starting point for the code-to-code scenario in our work; 2) the relatively small model size helps to reduce computational resources required for training and deploying. For generating the task set P, the range of the problem dimension is $[2, 50]$, and the number of components K is randomly chosen from $[1, 5]$. We randomly split the instruction set $\mathbb{I}$ into a training set $\mathbb{I}_{\text{train}}$ with 30k input-output pairs and a test set $\mathbb{I}_{\text{eval}}$ with the rest examples. We leave other details in Appendix B.

2) Baselines: We include two solution-to-solution approaches, OPRO [8] and LMEA [10], which prompt pre-trained LLMs (e.g., GPT-4 Turbo) repeatedly to generate and improve solutions for the given problems. Compared to OPRO, LMEA additionally engineered its prompt with an explicit indication of using some evolutionary operators to let LLMs act as an evolutionary optimizer for performance boost. We also include six general LLMs for code generation, namely CodeGen-Mono-350M [17], Phi-2-2.7B [37], Code Llama-7B [18], Llama 2-70B [32], DeepSeedMath-Instruct-7B [105] and GPT-4 Turbo [19], Codex-12B [15]. We prompt these three general LLMs with the same format as in our instruction set $\mathbb{I}$ to generate an optimizer for each problem instance. Note that we do not include the LLMs for algorithm design works such as EoH [43] into the comparison, since works in this line primarily address the combinatorial optimization problems but our LLaMoCo revolves around numerical optimization problems. The configurations of the baselines are set by default according to the corresponding references, and listed in Appendix B. Note that we do not include Algorithm Selection methods [106], [107] for comparison since LLaMoCo generates complete optimizer source codes that not only specify the selected algorithm but also include the necessary implementation details, which is beyond the scope of standard algorithm selection. Besides, LLaMoCo performs hyper-parameter tuning as part of the optimization code generation process, providing a level of configurability that algorithm selection methods cannot achieve.

3) Performance metrics: Compared to typical code generation task, optimization program generation task need to be considered under more systematic evaluation standards beyond code generation accuracy, e.g., optimization accuracy and runtime complexity. To this end, we propose four metrics to ensure rigorous evaluation:

a) **Error rate (Err.)** The robustness of the generated optimization program is a very important index for quality of service (QoS). We measure the robustness by the proportion of error programs generated by an LLM, named as error rate. For each instance, we use the optimization program generated by an LLM (ours or the others) to optimize that instance for 5 independent runs. We count the number of the generated programs which encounter compilation error or runtime error when being executed, denoted as N_{err} (every single run on each instance is counted). Then the error rate of an approach on the tested instances is calculated as $\frac{N_{err}}{5\times N_p}$.

b) **Recovery cost (Rec.)** While an optimization program may encounter compilation error or runtime error, we observe from our experiments that a certain proportion of the error programs could be repaired and recovered. We provide a metric named recovery cost to measure the efforts required to repair the generated programs. Concretely, during the test time, if the optimization program generated by an LLM was non-functional, we would give the tested LLM an additional turn of conversation to refine the errors of the generated optimization code. Concretely, we construct a prompt: for an optimization program a_j, we denote the number of lines in it as $L^{(j)}$, and the number of lines that need to be repaired as $L_{err}^{(j)}$. Then the recovery cost for a_j is $r_j = \frac{L_{err}^{(j)}}{L^{(j)}}$, and the recovery cost considering all N_{err} error programs is calculated as $\frac{\sum_{j=1}^{N_{err}} r_j}{N_{err}}$. For the case that the generated optimization code is still erroneous after the self-refine process, we set the performance of that LLM on that optimization problem as 0.

c) **Optimization performance (Perf.)** We measure the optimization performance of an approach by a min-max normalized objective value descent. Concretely, we first estimate an optimal objective value f_i^* for i-th problem instance, which can be easily obtained from our benchmarking process (achieved best objective value). For the given approach, we denote the performance on the i-th problem instance in j-th run as a min-max normalized term $w_{i,j} = 1 - \frac{f_{i,j}^* - f_i^*}{f_{i,j}^0 - f_i^*}$, where $f_{i,j}^0$ is the best objective value of the solutions initialized by the optimizer on solving the i-th problem instance in j-th run, and $f_{i,j}^*$ is the corresponding best objective the optimizer finds. We have to note that if the optimization code generated is still non-functional after the repairing process above, we assign a performance value of 0 for that run. At last, the overall average optimization performance of the given approach on the N_p instances can be calculated as follows: $\frac{\sum_{i=1}^{N_p}\sum_{j=1}^{5} w_{i,j}}{5 \times N_p}$.

d) **Computational overhead (Comp.)** Measuring the computational overhead by the wall-time complexity of an LLM-based approach is impractical since some of the LLMs only provide API for users. The network communication budget through calling the APIs would bias the ground results. We instead count the average number of tokens (input+output) consumed by an approach for solving a problem instance over the test runs.

These four metrics could provide a comprehensive evaluation of existing baselines and our LLaMoCo in aspects of code generation robustness, performance and runtime complexity. For example, a higher performance score indicates LLaMoCo holds expertise to design desired optimizers for diverse problems, while a lower error rate indicates it is capable of coding as good as expert-level researchers and engineers.

B. Performance Analysis

We use LLaMoCo-S(mall), -M(edium) and -L(arge) to denote the fine-tuned CodeGen-Mono (350M), Phi-2 (2.7B) and Code Llama (7B) models on $\mathbb{I}_{\text{train}}$, respectively.

1) Performance on test sets.: First, we evaluate the performance of our fine-tuned LLMs and the competitors on three test sets, $\mathbb{I}_{\text{eval}}/P_{\text{c}}$, $\mathbb{I}_{\text{eval}}/P_{\text{nc}}$, and $\mathbb{I}_{\text{eval}}$ that represent the unconstrained task set, constrained task set, and the complete set mixing unconstrained and constrained tasks, respectively, each with 5 independent runs. The numeric results and corresponding optimization curves are reported in Table I and Figure 3 respectively, which show that:

a) The LLMs fine-tuned by our LLaMoCo framework consistently achieve superior performance, which validates

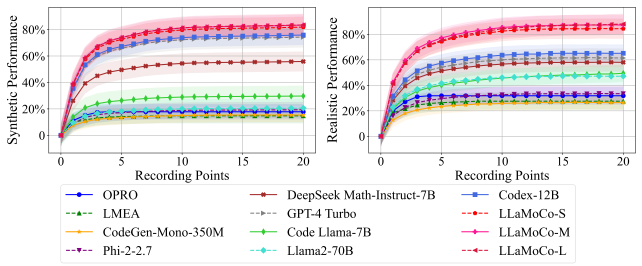


Fig. 3. The performance curves in $\mathbb{I}_{eval}$ (left) and $\mathbb{I}_{real}$ (right).

that instruction tuning the general LLMs with moderate expert-level knowledge would gain substantial performance reinforcement in optimization. For example, LLaMoCo-L fine-tuned on the Code Llama (7B) demonstrate a performance boost from 29.717% to 81.843% on $\mathbb{I}_{\text{eval}}$.

b) Although LLaMoCo-S is fine-tuned from a relatively small fundamental model, it achieves competitive performance to those of LLaMoCo-M and LLaMoCo-L. This reveals a potential marginal effect in instruction tuning, since the data scale should match the model capacity. See Table IV for a detailed experiment where we provide an initial exploration on LLaMoCo's scaling law.

c) The solution-to-solution approaches OPRO and LMEA achieve unsatisfactory performance on our complex optimization task sets. Considering the tremendous tokens these approaches consume to solve one optimization problem through iteratively prompting solutions, both the efficacy and efficiency (as shown in the 'Perf.' and 'Comp.' rows of Table I) of them require further improvement.

d) Among the six 'prompt for optimizer' models we compared, the GPT-4 Turbo dominates the others, which shows the power of a general-purpose LLM with high capacity. Nevertheless, it still underperforms our domain-specific LLaMoCo. Our models effectively reduce the error rates and the required recovery efforts for generating the codes of an optimizer through the instruction tuning. Meanwhile, note that the Code Llama (7B) model achieves better overall performance than the Llama 2 (70B) model in our experiments. The above observations validate that, although LLMs with larger capacity may show strong performance for solving general tasks, a smaller model could be sufficient to be fine-tuned as a domain-specific task solver.

e) Without specifically trained on our proposed $\mathbb{I}_{\text{train}}$, all general LLM baselines show high error rates when generating the optimizer program. This further outlines the advantages of instruction-tuning the general LLMs with LLaMoCo (with at most 6.13% error rate for our three pre-trained models). We have to note that recent few-shot prompting researches [108], [109] indicate that the error rate could be decreased by providing the general LLMs with moderate examples as hints. To verify if our LLaMoCo is more effective than few-shot prompting strategy, we also conduct a comparison study between our LLaMoCo models and the three general LLMs enhanced by the few-shot prompting strategy and present the comparison results in table II. The results show that few-shot prompting strategy does introduce performance improvement for the baselines. In particular, by showing the baselines with

TABLE I
RESULTS OF DIFFERENT APPROACHES IN TERMS OF **CODE ERROR RATE (ERR.)**, **CODE RECOVERY COST (REC.)**, **OPTIMIZATION PERFORMANCE (PERF.)**, AND **COMPUTATIONAL OVERHEAD (COMP.)** ON THE UNCONSTRAINED PROBLEMS ($\mathbb{I}_{\text{eval}}/P_{\text{c}}$), CONSTRAINED PROBLEMS ($\mathbb{I}_{\text{eval}}/P_{\text{nc}}$), BOTH CONSTRAINED AND UNCONSTRAINED PROBLEMS ($\mathbb{I}_{\text{eval}}$), AND REALISTIC PROBLEMS $\mathbb{I}_{\text{real}}$, WHERE "-" DENOTES THAT THE APPROACH DOES NOT GENERATE CODE (IT FOLLOWS A SOLUTION-TO-SOLUTION PARADIGM).

Testset	Metrics	Prompt for Solution		Prompt for Optimizer							Our LLaMoCo		
		OPRO	LMEA	CodeGen-Mono-350M	Phi-2-2.7B	DeepSeekMath-Instruct-7B	GPT-4 Turbo	Code Llama-7B	Llama2-70B	Codex-12B	LLaMoCo-S	LLaMoCo-M	LLaMoCo-L
$\mathbb{I}_{\text{eval}}/P_{\text{c}}$	Err. ↓	-	-	99.864%	97.413%	71.564%	43.333%	98.184%	99.673%	40.315%	5.437%	**4.414%**	4.697%
	Rec. ↓	-	-	80.234%	72.242%	13.483%	9.942%	67.857%	62.232%	9.217%	**9.684%**	10.101%	9.947%
	Perf. ↑	29.499% ±8.154%	20.350% ±7.223%	12.341% ±5.155%	17.313% ±4.145%	58.568% ±7.849%	71.783% ±8.197%	14.089% ±5.461%	18.922% ±6.484%	74.341% ±7.119%	85.360% ±7.987%	**86.412%** ±**8.594%**	85.810% ±8.669%
	Comp. ↓	115k	249k	1.9k	2.1k	2.1k	3.4k	1.7k	**1.5k**	3.5k	2.3k	2.3k	2.3k
$\mathbb{I}_{\text{eval}}/P_{\text{nc}}$	Err. ↓	-	-	99.413%	92.234%	67.488%	39.944%	90.474%	99.521%	39.173%	**5.697%**	6.130%	5.977%
	Rec. ↓	-	-	78.341%	54.156%	12.145%	16.463%	44.938%	49.202%	15.221%	11.861%	**10.443%**	10.584%
	Perf. ↑	4.514% ±2.698%	7.541% ±3.457%	20.314% ±5.021%	41.342% ±6.466%	53.477% ±6.597%	75.678% ±9.218%	46.968% ±8.136%	22.460% ±7.667%	76.262% ±7.247%	77.576% ±7.492%	79.718% ±7.124%	**83.404%** ±**8.148%**
	Comp. ↓	115k	249k	2.1k	2.2k	2.1k	3.5k	**2.0k**	**2.0k**	3.5k	2.5k	2.5k	2.5k
$\mathbb{I}_{\text{eval}}$	Err. ↓	-	-	99.421%	94.314%	69.371%	41.667%	95.156%	99.617%	39.679%	5.580%	**5.434%**	5.509%
	Rec. ↓	-	-	79.371%	63.314%	12.518%	13.072%	57.001%	55.717%	13.016%	10.826%	**10.349%**	10.461%
	Perf. ↑	17.821% ±5.604%	14.762% ±5.463%	15.341% ±5.093%	19.345% ±5.792%	55.847% ±7.339%	74.248% ±7.683%	29.717% ±7.648%	20.579% ±7.005%	75.830% ±7.338%	81.843% ±7.720%	83.369% ±7.853%	**83.451%** ±**8.360%**
	Comp. ↓	115k	249k	2.0k	2.1k	2.1k	3.5k	1.9k	**1.7k**	3.5k	2.4k	2.4k	2.4k

TABLE II
COMPARISON BETWEEN OUR LLAMOCO MODELS AND THE FEW-SHOT PROMPTING ENHANCEMENT OF THE GENERAL LLM BASELINES ON THE TEST SET $\mathbb{I}_{\text{eval}}$.

Testset	Metrics	Prompt for Optimizer (few-shot)			Our LLaMoCo
		GPT-4 Turbo (few-shot)	Code Llama-7B (few-shot)	Llama2-70B (few-shot)	LLaMoCo-L
$\mathbb{I}_{\text{eval}}$	Err. ↓	10.546%	15.235%	10.235%	**5.509%**
	Rec. ↓	8.423%	29.445%	**7.456%**	10.461%
	Perf. ↑	76.568%	45.775%	37.456%	**83.451%**
	Comp. ↓	7.0k	6.1k	6.5k	**2.4k**

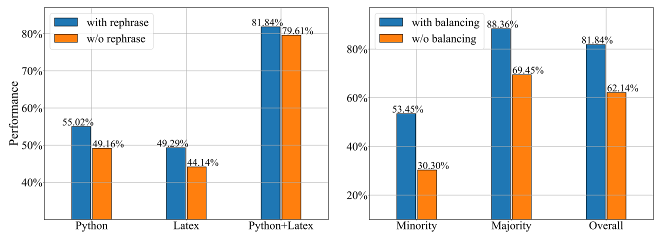


Fig. 4. **Left**: Effectiveness of the diversity enhancement strategy. **Right**: Effectiveness of the balanced data sampling strategy.

some prompts as hints, the Error rate and the Recovery cost are significantly reduced. Although the error rates of the three general LLM baselines are significantly reduced, the error rates are still over 10%. Besides, few-shot prompting for the general LLMs introduces additional computational overhead since it consumes more tokens to provide hint examples for the LLMs. Besides certain expertise is needed for selecting the examples. In contrast, our LLaMoCo-L model achieves lower error rate and better performance, while consuming less tokens.

2) Zero-shot performance on realistic problems: To validate the generalization of LLaMoCo on intricate real-world scenarios, we compare models fine-tuned by our LLaMoCo and the other baselines on a wide range of realistic instances. Concretely, we select 16 realistic optimization tasks with diverse optimization challenges such as optimization with complex constraints, rugged objective landscapes, expensive evaluation, ill-conditioned objective landscapes, multimodality etc. We present the results in Table III, where we provide the optimization performances of all baselines and our LLaMoCo. The results demonstrate the robust generalization performance of LLaMoCo for optimization problems in our daily life. This generalization roots from the stipulated problem description we proposed in this paper: every problem can be defined by its fundamental components: a) the objective function, b) the constraints, c) the search range of decision variables, and d) the function evaluation budget. Therefore either the synthetic problems or the realistic problems are represented by programming language hence share the semantic consistency, which helps the generalizability of LLaMoCo. Besides, a potential difference between synthetic and realistic problems is not structural issue, instead, is that realistic problem may hold more constraints and requires to be solved in small evaluation budget. To make LLaMoCo generalizable for such cases, the 6000 synthetic instances constructed by us include 3000 constrained ones, to ensure the generalization ability of the trained LLaMoCo models. Except the above 16 realistic scenarios, we further validate the zero-shot performance of LLaMoCo on a realistic problem collection proposed by [126], which covers 57 diverse real-world optimization scenarios that represents diverse optimization challenges. The results is provided in Table III, Appendix E, which show consistent observation that our LLaMoCo flexibly generates optimization program for those realistic problems, with promising optimization performance.

3) Qualitative analysis of generation errors: While Table I and II quantitatively demonstrate LLaMoCo's significantly lower error rates, a qualitative analysis of the generated outputs provides deeper insights into why our model is more robust. Through manual inspection of the experimental results, we identified distinct error patterns of different model categories:

- General LLMs: the general LLMs like GPT series and DeepSeek series show different extends of generation error for optimization domain. We could observe three major error types: a) API hallucination: although these LLMs know some specific library to call, however, they have hallucination of interfaces that never exist in the library; b) syntax errors: this case happens more in Llama series models, where they might generate totally wrong optimization procedure, resulting in syntax error; c) Hyper-parameter misinterpretation: these LLMs

TABLE III
PERFORMANCE COMPARISON BETWEEN LLAMOCO AND OTHER BASELINES ON REALISTIC PROBLEMS. THESE PROBLEMS COVER VARIOUS DOMAINS SUCH AS ENGINEERING, CONTINUOUS CONTROL, AUTOML, AND SCIENTIFIC DISCOVERY, SHOWING DIVERSE OPTIMIZATION CHALLENGES.

Testset	Prompt for Solution		Prompt for Optimizer							Our LLaMoCo		
	OPRO	LMEA	CodeGen-Mono-350M	Phi-2-2.7B	DeepSeekMath-Instruct-7B	GPT-4 Turbo	Code Llama-7B	Llama2-70B	Codex-12B	LLaMoCo-S	LLaMoCo-M	LLaMoCo-L
Haverly's Pooling [110]	58.567% ±7.974%	46.852% ±6.694%	43.729% ±6.747%	50.345% ±7.100%	62.782% ±8.448%	63.487% ±8.679%	51.425% ±7.894%	49.488% ±7.908%	64.144% ±8.541%	83.152% ±8.399%	80.468% ±8.104%	**86.758% ±8.392%**
Multi-product batch plant [111]	37.153% ±7.854%	40.256% ±7.015%	33.154% ±6.548%	40.782% ±7.015%	57.121% ±8.168%	67.158% ±8.970%	53.187% ±8.134%	51.156% ±8.318%	67.034% ±8.885%	75.145% ±9.231%	**78.851% ±9.344%**	76.148% ±9.017%
Robot gripper [112]	26.488% ±6.948%	34.786% ±7.197%	43.155% ±7.014%	49.486% ±7.487%	59.478% ±7.456%	58.586% ±7.492%	55.406% ±7.948%	52.045% ±8.137%	59.922% ±7.596%	61.798% ±8.874%	**64.587% ±8.997%**	62.891% ±8.657%
Wind Farm Layout [113]	31.499% ±7.152%	29.457% ±6.948%	19.456% ±5.047%	34.498% ±6.905%	53.145% ±7.872%	58.489% ±8.193%	46.487% ±7.866%	44.267% ±7.771%	58.950% ±8.011%	75.412% ±8.697%	75.654% ±8.547%	**77.364% ±8.621%**
SOPWM for 3-level inverters [114]	34.125% ±7.541%	36.445% ±7.144%	40.564% ±7.397%	46.557% ±7.842%	66.784% ±8.667%	76.447% ±9.175%	70.887% ±8.474%	68.456% ±8.199%	75.558% ±8.753%	81.364% ±9.547%	82.457% ±9.316%	**82.669% ±9.345%**
Protein Docking [115]	28.315% ±7.021%	17.342% ±5.310%	18.348% ±5.169%	40.348% ±6.781%	61.783% ±7.894%	74.284% ±8.369%	39.341% ±6.101%	37.481% ±6.236%	75.535% ±8.479%	80.734% ±9.642%	81.044% ±9.510%	**81.532% ±9.410%**
HPO [116]	58.567% ±8.445%	46.852% ±7.235%	43.729% ±7.615%	50.345% ±8.479%	62.782% ±8.909%	63.487% ±9.078%	51.425% ±7.483%	49.488% ±7.492%	65.462% ±8.972%	**83.657% ±8.778%**	80.468% ±8.483%	83.152% ±8.511%
Neuroevolution [117]	18.243% ±5.151%	19.423% ±5.147%	24.354% ±5.467%	26.487% ±5.464%	34.364% ±6.011%	37.145% ±6.232%	32.451% ±6.429%	35.478% ±6.597%	37.005% ±6.121%	57.364% ±7.973%	**58.145% ±7.815%**	57.341% ±7.842%
UAV Path Planning [118]	37.845% ±5.451%	39.464% ±5.643%	47.784% ±6.393%	49.987% ±6.525%	69.348% ±8.481%	68.882% ±8.451%	42.456% ±5.958%	41.014% ±5.451%	69.054% ±8.317%	80.153% ±8.451%	81.553% ±8.421%	**84.462% ±8.542%**
Heat exchanger network design [119]	43.151% ±5.981%	47.597% ±6.225%	59.460% ±7.262%	58.010% ±7.697%	68.124% ±8.243%	68.691% ±8.109%	44.166% ±6.080%	39.396% ±4.984%	68.143% ±8.255%	82.192% ±8.334%	82.111% ±8.314%	**83.690% ±8.559%**
Reactor network design [120]	38.933% ±5.663%	39.659% ±5.804%	56.300% ±7.339%	59.877% ±7.742%	66.614% ±8.224%	67.038% ±8.620%	50.505% ±6.944%	49.665% ±5.972%	66.929% ±8.610%	80.341% ±8.465%	**84.980% ±8.217%**	83.364% ±8.531%
Industrial refrigeration system [121]	30.642% ±4.997%	33.314% ±5.621%	44.984% ±6.482%	46.252% ±6.581%	58.805% ±7.676%	55.039% ±7.612%	39.951% ±5.643%	41.628% ±5.513%	56.608% ±7.661%	**79.504% ±8.014%**	77.261% ±8.051%	77.998% ±7.669%
Multiple disk clutch brake [122]	40.499% ±6.642%	40.587% ±5.908%	55.456% ±7.064%	58.128% ±7.891%	63.975% ±7.999%	64.221% ±7.905%	44.442% ±5.874%	42.360% ±5.984%	64.732% ±7.876%	77.247% ±7.858%	78.691% ±7.847%	**79.945% ±7.648%**
Rolling element bearing [123]	32.246% ±4.958%	32.369% ±4.882%	43.694% ±6.880%	45.694% ±6.548%	55.555% ±7.361%	56.094% ±6.590%	38.442% ±5.667%	37.310% ±5.048%	56.312% ±6.481%	63.154% ±7.104%	63.057% ±7.254%	**65.972% ±7.144%**
Gear train design [124]	35.648% ±5.135%	37.557% ±5.267%	46.949% ±6.801%	48.642% ±6.661%	59.694% ±7.661%	60.159% ±6.898%	48.642% ±6.198%	48.212% ±5.659%	60.606% ±6.724%	78.681% ±7.951%	79.298% ±7.940%	**80.482% ±7.680%**
Beef cattle feed ration [125]	36.541% ±5.770%	38.448% ±5.268%	50.694% ±7.153%	55.535% ±7.064%	63.269% ±8.015%	62.566% ±6.991%	47.238% ±6.275%	46.456% ±5.789%	63.121% ±6.784%	76.548% ±7.894%	77.941% ±7.882%	**79.960% ±7.821%**

TABLE IV
OPTIMIZATION PERFORMANCE COMPARISON ACROSS DIFFERENT MODEL SIZES AND DATASET SIZES.

model/data	1k	5k	15k	30k
350M	47.260%	66.661%	80.306%	81.843%
1B	46.799%	67.829%	81.783%	82.541%
3B	47.131%	68.492%	82.501%	83.315%
7B	45.645%	70.147%	82.966%	**83.513%**

TABLE V
THE AVERAGED OPTIMIZATION PERFORMANCE NORMALIZED BY OUR ORIGINAL GRANULARITY'S PERFORMANCE AND AVERAGED SEARCHING WALL TIME FOR ONE PROBLEM INSTANCE OF DIFFERENT GRID SEARCH GRANULARITIES.

	half	our setting	double
Performance	71.793%	1	102.344%
wall time	6s	211s	6379s

struggle in figuring out the formats and data types of the hyper-parameters such as the searching range vector and function evaluations. They might neglect these information from the prompt and use a default setting they believe correct. All above aspects reveal the necessity of LLaMoCo's instruction tuning to get general LLMs prepared as optimization experts.

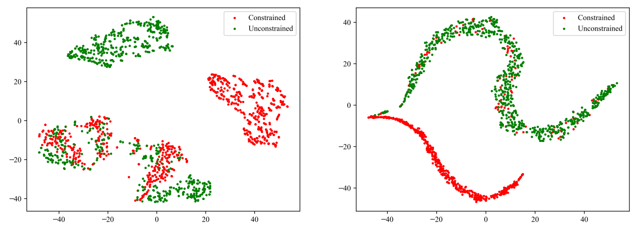
- LLaMoCo: the primary error case in LLaMoCo is that for a very small portion of problems where the constraint functions are much simpler than the objective functions, such problem structures might misleads LLaMoCo to generate an non-constrained optimizer for them, which causes running time error. Overall, LLaMoCo is capable of accurately generating zero error codes, thanks to the robust instruction tuning.

4) Scaling law in LLaMoCo: To in-depth explore LLaMoCo's scaling law, we chose the base model of our LLaMoCo-S, which is the CodeGen family (350M 1B, 3B, 7B), for the experiment. Further, we construct four training sets with different sizes (1k, 5k, 15k, 30k). The LLaMoCo-S model in this paper corresponds to 350M CodeGen model trained on 30k data. We presents the optimization performance of these 16 trained models on the test set $\mathbb{I}_{eval}$ in Table IV. The results above provide several key observations: a) when data size is very small (1k), increasing model size would not obtain any performance gain, which possibly indicates overfitting. b) for all model sizes, increasing the dataset size could introduce performance gain consistently. c) a marginal effect can be observed: for each model size, increasing data from 1k to 15k produces much more performance gain than increasing data from 15k to 30k. d) In summary, both the model size and the dataset size could influence the final performance of LLaMoCo. A dataset with 15k training samples could secure the performance, even for the smallest 350M model. These scaling law results also outline the rationale behind our 30k data scale choice for instruction-tuning LLaMoCo, increasing data scale further may not introduce significant performance improvement due to the model capacity.

TABLE VI
COMPARISON BETWEEN LLAMOCO AND ALGORITHM SELECTION METHODS.

	SBS	PIAS	AS-LLM	SBS +SMAC3	PIAS +SMAC3	AS-LLM +SMAC3	LLaMoCo-S
$\mathbb{I}_{\text{eval}}$	69.475% ±6.882%	62.341% ±6.847%	55.157% ±6.369%	75.339% ±7.659%	71.974% ±7.214%	59.782% ±6.484%	**81.843% ±7.720%**
$\mathbb{I}_{\text{real}}$	73.158% ±7.342%	68.397% ±7.164%	61.461% ±6.871%	79.239% ±7.913%	75.985% ±7.554%	64.942% ±6.876%	**88.135% ±7.894%**
Per-instance Runtime	1.9s	2.1s	6.1s	12.1s	12.1s	14.5s	5.9s

5) Algorithm selection performance: A similar scope to LLaMoCo is algorithm selection (AS) [106]. While both our work and AS are capable of selecting desired optimizers for given problem instances, key differences exist: 1) Compared to AS, LLaMoCo selects an optimizer *and* tunes its hyper-parameters simultaneous by outputting the executable optimization codes, aiming to operating as a synergy of AS and algorithm configuration (AC). 2) LLaMoCo unifies semantic



(a) Without contrastive learning. (b) With contrastive learning.

Fig. 5. t-SNE Visualization of the hidden features in LLaMoCo.

understanding of diverse optimization problems (constraint-free, constrained etc.), mitigating the feature extraction challenges in traditional AS. 3) Our work facilitate end-to-end optimization problem solving. It not only performs AS + AC at the methodological level, but also directly outputs executable program at the application level, which is more promising in realistic scenarios.

We show our LLaMoCo's advantage against automatic algorithm selection methods in Table VI. In this experiment, we include three representative algorithm selection baselines: a) **SBS**: single-best-solver in our constructed algorithm pool that achieves the best averaged optimization performance on the test set. For the unconstrained test set $\mathbb{I}_{eval}/P_c$, we checked the benchmarking results and found SBS is BIPOP-CMA-ES [82]; For constrained set $\mathbb{I}_{eval}/P_{nc}$, SBS is SLSQP [89]. b) **PIAS**: per-instance-algorithm-selection approach in [127], we use the xgboost classifier version. This baseline trains a classifier that maps a problem's feature vector to a label that indicates selected optimizer from the algorithm pool. c) **AS-LLM**: a newly proposed LLM-based algorithm selection framework [128], which leverages the textual embedding of the problem prompt as the decision vector, and them maps the decision vector to an optimizer in the algorithm pool. We also include three baselines that use strong HPO library SMAC3 [129] to further fine-tune the three algorithm selection models. We report the average normalized performances of the baselines and LLaMoCo-S in Table VI, with error bar annotated below. We also report in this table the average time cost for the baselines to solve a problem instance.

Generally, we observe that algorithm selection, when being incorporated with hyper-parameter tuning techniques, could achieve better optimization performance. Our LLaMoCo model, being instruction-tuned to generate both algorithm code and the hyper-parameter settings, could achieve preferred optimization performance, while consuming much less time than works that select algorithm first and then use HPO library for parameter tuning. The efficiency advantage stems from the end-to-end, one-shot generation of optimization programs, which handles optimizer selection and the corresponding hyper-parameter optimization simultaneously. In contrast, for AS + SMAC3 baselines, they require iterative hyper-parameter optimization, which significantly increasing their running time per instance.

C. Ablation study

1) Diversity enhancement: To improve the generalization of the fine-tuned LLMs in LLaMoCo, we enrich the task descriptions for each problem instance by augmenting the description of its objective function and constraints with Python or LaTeX codes of different writing styles. We illustrate the effect of this procedure in the left of Figure 4 by showing the optimization performance of six LLaMoCo-S models trained on pure Python, pure LaTeX and Python+LaTeX data, with or without the diversity enhancement by rephrasing. The results show that providing multi-lingual descriptions of optimization problems significantly boosts the generalization performance, while rephrasing each description with multiple writing styles further enhances the final training results.

2) Balanced data sampling: In LLaMoCo, we address the imbalanced data distribution (caused by dominate optimizers) through performing example-proportional sampling on $\mathbb{I}_{\text{train}}$. To investigate its effectiveness, we train two LLaMoCo-S models on $\mathbb{I}_{\text{train}}$, with or without the data balancing strategy, respectively. The optimization performance of the two models is presented in the right of Figure 4, by separately considering the majority instances (which request the dominating optimizers), the minority instances (which request the others), and the overall instances of $\mathbb{I}_{\text{eval}}$. The results consistently show that keeping a balanced training data distribution significantly boosts performance.

3) Grid search granularity: Recall that as we described in Section III-A2, we obtain the desired optimizer for each problem instance in $\mathbb{I}_{train}$ by benchmarking the optimizers in the algorithm pool with grid-search to locate best-performing configurations of them. We conduct two additional benchmarking processes, with half and double granularity of our original grid-search granularity. For example, if a hyper-parameter holds four optional values in our setting: [0.2, 0.4, 0.6, 0.8], half granularity denotes [0.2, 0.8], double granularity denotes [0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8]. We present the averaged optimization performance of the most effective optimizers on our problem set searched by these two granularities, normalized by our original granularity's performance, as well as the averaged searching wall time for one problem instance in Table V. The results reveal that the searching wall time increases exponentially since there are 4-5 hyper-parameters in an optimizer. However, the performance improvement obtained by spending so much computational resources is only 2.344%. This validates the appropriateness of our grid search granularity choice.

4) Contrastive warm-up: The contrastive warm-up phase in our proposed two-phase instruction tuning strategy (see Section III-B) aims to reduce the cross-modal ambiguity by aligning the latent representations of different prompts that share the same desired optimizer (vice versa). We in this section validate the effectiveness of the contrastive warm-up on improving the training efficiency and stability. We first plot in Figure 5 the t-SNE representations of the hidden features of the tested problem prompts output by LLaMoCo before LLaMoCo outputs subsequent optimizer programs. We show the decision bound between constrained problems and unconstrained ones. The results highlight that, without our proposed contrastive warm up, some constrained problems and unconstrained ones cannot be separated, leading to feature-level confusion that causes generation bias in LLaMoCo.

TABLE VII
COMPARISON BETWEEN LLAMOCO AND THE OPENAI MODELS, IN TERMS OF **CODE ERROR RATE (ERR.)**, **CODE RECOVERY COST (REC.)**, **OPTIMIZATION PERFORMANCE (PERF.)**, AND **COMPUTATIONAL OVERHEAD (COMP.)** ON THE TEST SET $\mathbb{I}_{\text{eval}}$.

		GPT-4 Turbo	GPT-4o	o1-mini	o1-preview	GPT-4 vector search	LLaMoCo-S	LLaMoCo-M	LLaMoCo-L
$\mathbb{I}_{\text{eval}}$	Err. $\downarrow$	41.667%	33.771%	**3.355%**	4.107%	9.336%	5.580%	5.434%	5.509%
	Rec. $\downarrow$	13.072%	14.405%	**10.299%**	10.641%	12.853%	10.826%	10.349%	10.461%
	Perf. $\uparrow$	74.248%	75.193%	80.269%	79.945%	79.944%	81.843%	**86.412%**	85.810%
	Comp. $\downarrow$	3.5k	3.6k	4.1k	4.1k	7.1k	**2.4k**	**2.4k**	**2.4k**

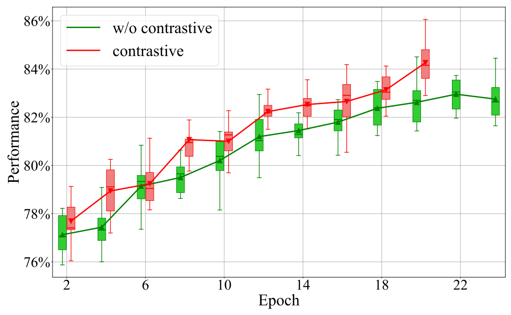


Fig. 6. The performance gain curves of LLaMoCo models on the test sets when trained with/without contrastive learning.

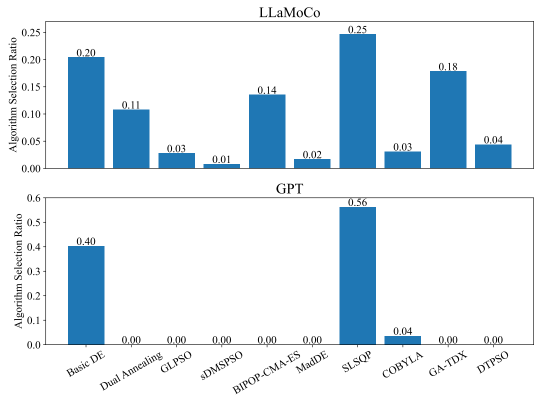


Fig. 7. The diversity of generated optimizers in LLaMoCo and GPT cases.

We further illustrate the performance gain curves with or without the contrastive warm-up during the instruction tuning in Figure 6, where we present the case of training effectiveness of LLaMoCo-L on $\mathbb{I}_{\text{eval}}$ with or without contrastive learning. The results show that incorporating such a contrastive warm-up strategy consistently aids in accelerating the convergence of the subsequent instruction tuning. Refer to Appendix E.3 for all results on all LLaMoCo models and test problem sets.

D. Open-Ended Discussion

1) Is GPT-4 an optimization expert?: Considering the competitive performance of GPT-4, as shown in Table I, we delve into whether GPT-4 can be deemed as a genuine optimization expert. Upon viewing the optimization codes generated by GPT-4 for both test and realistic problem sets, a noteworthy pattern emerges. We provide in Figure 7 statistical histogram on the optimizer selection distribution of LLaMoCo (averaged across -L, -M and -S) and GPT series models (averaged on all GPT models involved in our experiments). By instruction tuning on our constructed knowledge set, LLaMoCo could generate desired optimizer according to concrete problem characteristics. In contrast, GPT series models, can only generate basic DE or SLSQP for all tested problems. While SLSQP and basic DE are classical solvers included in our chosen advanced optimizers, our benchmarking results identify that on a proportion of tested problems, it underperforms the others, e.g., BIPOP-CMA-ES [82]. This inflexibility exactly demonstrates that although the training corpus of GPT includes some widely used optimization scripts such as DE, SLSQP, such limited knowledge makes these general LLMs unable to customize optimizer for specific problems.

To investigate further, we have conducted testing on a series of latest GPT models: GPT-4o, GPT o1-mini and GPT o1-preview. In particular, we also add a GPT-4 vector search baseline [130] which uses a vector search method to provide in-context enhancement for the GPT-4 model. We present the performance comparison in Table VII. From the results, we can observe that: a) o1 v.s. GPT-4o: indeed, o1 models achieve significantly lower coding errors than 4o model, demonstrating their robust coding enhancement. b) o1-mini v.s. LLaMoCo: on the one hand, the error rate of o1-mini is lower than our LLaMoCo, which originates from the black-box training of o1-mini on extremely large coding tasks. On the other hand, LLaMoCo, trained on a very small model, could achieve more optimization performance gain, with fewer tokens consumed. Besides, we look into the source codes generated by o1-mini as we have done for GPT-4 Turbo model in Section IV-D. It turns out that o1-mini also leans toward generating a particular optimizer, DE algorithm, for almost all tested problems. This further underscores the core motivation of LLaMoCo, which is exploring how to inject domain-specific knowledge into LLMs to adapt them for specific tasks. c) By providing GPT-4 Turbo with an example prompt-answer pair which is similar to the tested prompt, the error rate of the generated optimizer code significantly declines. d) However, such a prompting strategy consumes doubled tokens than directly prompting GPT-4 Turbo, which is inefficient considering our LLaMoCo only requires 2.4k tokens to achieve superior optimization performance. This underscores the importance of our LLaMoCo for adapting LLMs to solve optimization problems.

2) Hallucination in LLMs: Through checking the errors occur in the optimization codes generated by baseline LLMs, we recognize several major hallucination phenomena: a) Incorrect interface calling: The LLM might call an optimization library correctly, however, it mistakenly believes some functional interfaces were in the library (actually no such interfaces in the package). b) Reckless hyper-parameter setting: The LLM might holds very limited knowledge on how to set proper parameters for a solver to achieve ideal performance on given problem instance, resulting in random and reckless parameter tuning. c) Confusion on constraint: The LLM might mistakenly call an optimization library that specializes at unconstrained problems for solving constrained ones, due to its limited knowledge on mapping between problem types and solver types.

We now elaborate our contributions on addressing these unique hallucination phenomena in optimization domain, which are in two-folds: a) Large scale benchmarking: For each instance in the 6000 synthetic problems we synthetized, we find the best optimizer from the algorithm pool, with accurate interface calling and further grid hyper-parameter testing to locate best parameter setting. By doing so, we ensure high quality training data which could significantly reduce the hallucination of "Incorrect Interface Calling" and "Reckless Hyper-parameter Setting". b) Contrastive pre-training: Through introducing a contrastive warm up phase before the normal instruction tuning process, we reduced feature-level confusion of the LLM for diverse problem types, e.g., constrained or not. We also show a hallucination example of an advanced code LLM Codex [15] in Appendix E.2.

3) Human-level Interpretability: To interpreting LLaMoCo's behavior patterns, we provide following two aspects as intuitive explanation of LLaMoCo's decision philosophy: a) Adaptive algorithm selection: As Figure 7 shows, by instruction tuning on our constructed knowledge set, LLaMoCo could generate desired optimizers according to concrete problem characteristics. b) Precise problem recognition: We further validate the recognition ability of our LLaMoCo and other GPT baselines for different problem types, e.g., constrained and unconstrained problems. We compute a recognition accuracy metric r_acc by checking if the optimizer program generated by a baseline could be used for solving the given problem's type (constrained or unconstrained). On average, LLaMoCo models achieve 98.3% r_acc, while the GPT baselines only achieve 70.6% r_acc. It is quite clear to see that through instruction-tuning, LLaMoCo models could recognize the problem type correctly hence generate proper optimizers for given problems, which results in the superior optimization performance.

4) Extensibility: We provide a brief discussion on the extensibility of LLaMoCo here. Our primary goal was to investigate the fundamental challenges and core methodologies of adapting general (code) LLMs into expert optimization agents. We believe that establishing a principled approach for this process is a more pressing and foundational research contribution at this stage than merely extending the application to numerous optimization domains such as multi-objective optimization. Nevertheless, the instruction-tuning workflow proposed in this paper does not hold any specific designs restricted within single-objective optimization. The proposed code templates, data construction process and instruction-tuning strategy could be extended to other optimization domain with minimal development efforts.

V. Conclusion

We introduce LLaMoCo, a novel instruction-tuning framework to adapt general LLMs to function as high-performing, end-to-end systems for solving optimization problems. To achieve this, we first customize stipulated Python/LaTex prompt and answer templates for a universal representation on diverse optimization problems and optimization algorithms. We then meticulously construct an instruction set with more than 30k demonstration examples through a rigorous and large scale benchmarking, resulting in a comprehensive knowledge database. To inject the collected optimization knowledge into LLaMoCo, we employ a novel two-phase instruction tuning strategy to fine-tune a series of LLMs, where a contrastive learning-based warmup effectively reduces representation alignment issue from noisy user inputs. Objective evaluation is addressed by comprehensive experiments with four novel evaluation metrics. Key findings include: a) LLaMoCo models consistently outperform LLM-based baselines in both performance and coding coherence; b) through instruction-tuning with diverse synthetic instances, LLaMoCo show promising zero-shot performance over unseen, diverse and complex realistic scenarios. Notably, we observe that a relatively small LLM is sufficient to be tuned as an highly effective optimization code generator superior to GPT-4; c) ablation studies underscore that the each tailored design in LLaMoCo consistently contributes to the final model's performance; d) LLaMoCo demonstrates a greater ability to customize diverse optimization programs for different problems than advance general LLMs, suggesting an emergence of generalization through our instruction-tuning approach.

As a pioneering work, LLaMoCo holds certain limitations. On the one hand, current LLaMoCo models focus on single-objective optimization problems, serving as a proof of concept for the framework's efficacy in instruction-tuning. An immediate and promising direction for future work is to extend the framework to accommodate more complex, out-of-distribution problem types, such as multi-objective and multi-task optimization problems. On the other hand, beyond the current reliance on a labeled dataset for fine-tuning, we envision two potential enhancements to this paradigm: a) integrate alignment tuning techniques (e.g., DPO or RLHF), which not only promises to refine the model's performance with expert optimization knowledge but also ensures that the generated programs are more aligned with human preference; and b) incorporate LLaMoCo with a multi-turn code optimization framework, which allows the model to progressively refine its generated solutions, likely improves both the code correctness and the optimization performance. In summary, we hope this preliminary exploratory research would inspire the community to further to explore the potential of fine-tuning LLMs as the next generation of powerful, versatile optimizers.

References

[1] G. G. Wang and S. Shan, "Review of metamodeling techniques in support of engineering design optimization," in *International Design Engineering Technical Conferences and Computers and Information in Engineering Conference*, 2006.

[2] P. E. Gill, W. Murray, and M. H. Wright, *Practical optimization*, 2019.

[3] P. A. Vikhar, "Evolutionary algorithms: A critical review and its future prospects," in *ICGTSPICC*, 2016.

[4] L. Floridi and M. Chiriatti, "Gpt-3: Its nature, scope, limits, and consequences," *Minds Mach.*, 2020.

[5] B. D. Lund and T. Wang, "Chatting about chatgpt: how may ai and gpt impact academia and libraries?" *LHTN*, 2023.

[6] S. S. Biswas, "Role of chat gpt in public health," *Ann. Biomed. Eng.*, 2023.

[7] A. AhmadiTeshnizi, W. Gao, and M. Udell, "Optimus: Optimization modeling using mip solvers and large language models," *arXiv preprint*, 2023.

[8] C. Yang, X. Wang, Y. Lu, H. Liu, Q. V. Le, D. Zhou, and X. Chen, "Large language models as optimizers," *arXiv preprint*, 2023.

[9] P.-F. Guo, Y.-H. Chen, Y.-D. Tsai, and S.-D. Lin, "Towards optimizing with large language models," *arXiv preprint*, 2023.

[10] S. Liu, C. Chen, X. Qu, K. Tang, and Y.-S. Ong, "Large language models as evolutionary optimizers," *arXiv preprint*, 2023.

[11] B. Romera-Paredes, M. Barekatain, A. Novikov, M. Balog, M. P. Kumar, E. Dupont, F. J. Ruiz, J. S. Ellenberg, P. Wang, O. Fawzi *et al.*, "Mathematical discoveries from program search with large language models," *Nature*, 2024.

[12] R. T. Lange, Y. Tian, and Y. Tang, "Large language models as evolution strategies," *arXiv preprint*, 2024.

[13] M. Pluhacek, A. Kazikova, T. Kadavy, A. Viktorin, and R. Senkerik, "Leveraging large language models for the generation of novel metaheuristic optimization algorithms," in *GECCO*, 2023.

[14] W. X. Zhao, K. Zhou, J. Li, T. Tang, X. Wang, Y. Hou, Y. Min, B. Zhang, J. Zhang, Z. Dong *et al.*, "A survey of large language models," *arXiv preprint*, 2023.

[15] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. D. O. Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman *et al.*, "Evaluating large language models trained on code," *arXiv preprint*, 2021.

[16] R. Li, L. B. Allal, Y. Zi, N. Muennighoff, D. Kocetkov, C. Mou, M. Marone, C. Akiki, J. Li, J. Chim *et al.*, "Starcoder: may the source be with you!" *arXiv preprint*, 2023.

[17] E. Nijkamp, B. Pang, H. Hayashi, L. Tu, H. Wang, Y. Zhou, S. Savarese, and C. Xiong, "Codegen: An open large language model for code with multi-turn program synthesis," in *ICLR*, 2023.

[18] B. Roziere, J. Gehring, F. Gloeckle, S. Sootla, I. Gat, X. E. Tan, Y. Adi, J. Liu, T. Remez, J. Rapin *et al.*, "Code llama: Open foundation models for code," *arXiv preprint*, 2023.

[19] J. Achiam, S. Adler, S. Agarwal, L. Ahmad, I. Akkaya, F. L. Aleman, D. Almeida, J. Altenschmidt, S. Altman, S. Anadkat *et al.*, "Gpt-4 technical report," *arXiv preprint*, 2023.

[20] L. Ouyang, J. Wu, X. Jiang, D. Almeida, C. Wainwright, P. Mishkin, C. Zhang, S. Agarwal, K. Slama, A. Ray *et al.*, "Training language models to follow instructions with human feedback," *NIPS*, 2022.

[21] P. F. Christiano, J. Leike, T. Brown, M. Martic, S. Legg, and D. Amodei, "Deep reinforcement learning from human preferences," *NIPS*, 2017.

[22] D. M. Ziegler, N. Stiennon, J. Wu, T. B. Brown, A. Radford, D. Amodei, P. Christiano, and G. Irving, "Fine-tuning language models from human preferences," *arXiv preprint*, 2019.

[23] J. Wei, M. Bosma, V. Zhao, K. Guu, A. W. Yu, B. Lester, N. Du, A. M. Dai, and Q. V. Le, "Finetuned language models are zero-shot learners," in *ICLR*, 2022.

[24] V. Sanh, A. Webson, C. Raffel, S. Bach, L. Sutawika, Z. Alyafeai, A. Chaffin, A. Stiegler, A. Raja, M. Dey, M. S. Bari, C. Xu, U. Thakker, S. S. Sharma, E. Szczechla, T. Kim, G. Chhablani, N. Nayak, D. Datta, J. Chang, M. T.-J. Jiang, H. Wang, M. Manica, S. Shen, Z. X. Yong, H. Pandey, R. Bawden, T. Wang, T. Neeraj, J. Rozen, A. Sharma, A. Santilli, T. Fevry, J. A. Fries, R. Teehan, T. L. Scao, S. Biderman, L. Gao, T. Wolf, and A. M. Rush, "Multitask prompted training enables zero-shot task generalization," in *ICLR*, 2022.

[25] H. Gupta, S. A. Sawant, S. Mishra, M. Nakamura, A. Mitra, S. Mashetty, and C. Baral, "Instruction tuned models are quick learners," *arXiv preprint*, 2023.

[26] K. Singhal, S. Azizi, T. Tu, S. S. Mahdavi, J. Wei, H. W. Chung, N. Scales, A. Tanwani, H. Cole-Lewis, S. Pfohl *et al.*, "Large language models encode clinical knowledge," *Nature*, 2023.

[27] Q. Huang, M. Tao, Z. An, C. Zhang, C. Jiang, Z. Chen, Z. Wu, and Y. Feng, "Lawyer llama technical report," *arXiv preprint*, 2023.

[28] J. Zhang, R. Xie, Y. Hou, W. X. Zhao, L. Lin, and J.-R. Wen, "Recommendation as instruction following: A large language model empowered recommendation approach," *arXiv preprint*, 2023.

[29] D. Zan, B. Chen, F. Zhang, D. Lu, B. Wu, B. Guan, W. Yongji, and J.-G. Lou, "Large language models meet nl2code: A survey," in *ACL*, 2023.

[30] Q. Zheng, X. Xia, X. Zou, Y. Dong, S. Wang, Y. Xue, L. Shen, Z. Wang, A. Wang, Y. Li *et al.*, "Codegeex: A pre-trained model for code generation with multilingual benchmarking on humaneval-x," in *ACM SIGKDD*, 2023.

[31] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell *et al.*, "Language models are few-shot learners," *NIPS*, 2020.

[32] H. Touvron, L. Martin, K. Stone, P. Albert, A. Almahairi, Y. Babaei, N. Bashlykov, S. Batra, P. Bhargava, S. Bhosale *et al.*, "Llama 2: Open foundation and fine-tuned chat models," *arXiv preprint*, 2023.

[33] A. Q. Jiang, A. Sablayrolles, A. Roux, A. Mensch, B. Savary, C. Bamford, D. S. Chaplot, D. d. l. Casas, E. B. Hanna, F. Bressand *et al.*, "Mixtral of experts," *arXiv preprint*, 2024.

[34] J. Austin, A. Odena, M. Nye, M. Bosma, H. Michalewski, D. Dohan, E. Jiang, C. Cai, M. Terry, Q. Le *et al.*, "Program synthesis with large language models," *arXiv preprint*, 2021.

[35] Y. Lai, C. Li, Y. Wang, T. Zhang, R. Zhong, L. Zettlemoyer, W.-t. Yih, D. Fried, S. Wang, and T. Yu, "Ds-1000: A natural and reliable benchmark for data science code generation," in *ICML*, 2023.

[36] Y. Li, D. Choi, J. Chung, N. Kushman, J. Schrittwieser, R. Leblond, T. Eccles, J. Keeling, F. Gimeno, A. Dal Lago *et al.*, "Competition-level code generation with alphacode," *Science*, 2022.

[37] M. Javaheripi, S. Bubeck, M. Abdin, J. Aneja, S. Bubeck, C. C. T. Mendes, W. Chen, A. Del Giorno, R. Eldan, S. Gopi *et al.*, "Phi-2: The surprising power of small language models," 2023.

[38] D. Fried, A. Aghajanyan, J. Lin, S. Wang, E. Wallace, F. Shi, R. Zhong, S. Yih, L. Zettlemoyer, and M. Lewis, "Incoder: A generative model for code infilling and synthesis," in *ICLR*, 2023.

[39] Z. Zhang, C. Wang, Y. Wang, E. Shi, Y. Ma, W. Zhong, J. Chen, M. Mao, and Z. Zheng, "Llm hallucinations in practical code generation: Phenomena, mechanism, and mitigation," *Proceedings of the ACM on Software Engineering*, 2025.

[40] X. Gu, M. Chen, Y. Lin, Y. Hu, H. Zhang, C. Wan, Z. Wei, Y. Xu, and J. Wang, "On the effectiveness of large language models in domain-specific code generation," *ACM Transactions on Software Engineering and Methodology*, 2025.

[41] Z. Wang, S. Liu, J. Chen, and K. C. Tan, "Large language model-aided evolutionary search for constrained multiobjective optimization," *arXiv preprint*, 2024.

[42] F. Liu, X. Lin, Z. Wang, S. Yao, X. Tong, M. Yuan, and Q. Zhang, "Large language model for multi-objective evolutionary optimization," *arXiv preprint*, 2023.

[43] F. Liu, X. Tong, M. Yuan, X. Lin, F. Luo, Z. Wang, Z. Lu, and Q. Zhang, "Evolution of heuristics: Towards efficient automatic algorithm design using large language model," in *ICML*, 2024. [Online]. Available: https://arxiv.org/abs/2401.02051

[44] M. R. Zhang, N. Desai, J. Bae, J. Lorraine, and J. Ba, "Using large language models for hyperparameter optimization," in *NIPS*, 2023.

[45] Q. Guo, R. Wang, J. Guo, B. Li, K. Song, X. Tan, G. Liu, J. Bian, and Y. Yang, "Connecting large language models with evolutionary algorithms yields powerful prompt optimizers," *arXiv preprint*, 2023.

[46] S. Min, X. Lyu, A. Holtzman, M. Artetxe, M. Lewis, H. Hajishirzi, and L. Zettlemoyer, "Rethinking the role of demonstrations: What makes in-context learning work?" *arXiv preprint*, 2022.

[47] J. Lehman, J. Gordon, S. Jain, K. Ndousse, C. Yeh, and K. O. Stanley, "Evolution through large models," in *Handbook of Evolutionary Machine Learning*, 2023.

[48] A. Chen, D. M. Dohan, and D. R. So, "Evoprompting: Language models for code-level neural architecture search," *arXiv preprint*, 2023.

[49] S. Brahmachary, S. M. Joshi, A. Panda, K. Koneripalli, A. K. Sagotra, H. Patel, A. Sharma, A. D. Jagtap, and K. Kalyanaraman, "Large language model-based evolutionary optimizer: Reasoning with elitism," *arXiv preprint*, 2024.

[50] Y. Huang, S. Wu, W. Zhang, J. Wu, L. Feng, and K. C. Tan, "Autonomous multi-objective optimization using large language model," *IEEE Transactions on Evolutionary Computation*, 2025.

[51] D. Forniés-Tabuenca, A. Uribe, U. Otamendi, A. Artetxe, J. C. Rivera, and O. L. de Lacalle, "Remoh: A reflective evolution of multi-objective heuristics approach via large language models," *arXiv preprint arXiv:2506.07759*, 2025.

[52] F. Liu, R. Zhang, Z. Xie, R. Sun, K. Li, X. Lin, Z. Wang, Z. Lu, and Q. Zhang, "Llm4ad: A platform for algorithm design with large language model," *arXiv preprint arXiv:2412.17287*, 2024.

[53] N. van Stein and T. Bäck, "Llamea: A large language model evolutionary algorithm for automatically generating metaheuristics," *IEEE Transactions on Evolutionary Computation*, 2024.

[54] R. Zhong, Y. Xu, C. Zhang, and J. Yu, "Leveraging large language model to generate a novel metaheuristic algorithm with crispe framework," *Cluster Computing*, 2024.

[55] F. Liu, R. Zhang, X. Lin, Z. Lu, and Q. Zhang, "Fine-tuning large language model for automated algorithm design," *arXiv preprint arXiv:2507.10614*, 2025.

[56] C. Huang, Z. Tang, S. Hu, R. Jiang, X. Zheng, D. Ge, B. Wang, and Z. Wang, "Orlm: A customizable framework in training large models for automated optimization modeling," *Operations Research*, 2025.

[57] W. Liu, L. Chen, and Z. Tang, "Large language model aided multi-objective evolutionary algorithm: a low-cost adaptive approach," *arXiv preprint arXiv:2410.02301*, 2024.

[58] H. Hao, X. Zhang, and A. Zhou, "Large language models as surrogate models in evolutionary algorithms: A preliminary study," *Swarm and Evolutionary Computation*, 2024.

[59] C. He, Y. Tian, and Z. Lu, "Artificial evolutionary intelligence (aei): evolutionary computation evolves with large language models: C. he et al." *Journal of Membrane Computing*, 2025.

[60] H. Yu and J. Liu, "Deep insights into automated optimization with large language models and evolutionary algorithms," *arXiv preprint arXiv:2410.20848*, 2024.

[61] W. Chao, J. Zhao, L. Jiao, L. Li, F. Liu, and S. Yang, "When large language models meet evolutionary algorithms," *arXiv preprint arXiv:2401.10510*, 2024.

[62] C. Zhou, P. Liu, P. Xu, S. Iyer, J. Sun, Y. Mao, X. Ma, A. Efrat, P. Yu, L. Yu *et al.*, "Lima: Less is more for alignment," *arXiv preprint*, 2023.

[63] S. P. Boyd and L. Vandenberghe, *Convex optimization*, 2004.

[64] G. Wu, R. Mallipeddi, and P. N. Suganthan, "Problem definitions and evaluation criteria for the cec 2017 competition on constrained real-parameter optimization," *National University of Defense Technology, Changsha, Hunan, PR China and Kyungpook National University, Daegu, South Korea and Nanyang Technological University, Singapore, Technical Report*, 2017.

[65] H. Guo, Z. Ma, J. Chen, Z. Li, G. Peng, Y.-J. Gong, Y. Ma, and Z. Cao, "Metabox: A benchmark platform for meta-black-box optimization with reinforcement learning," in *NIPS*, 2023.

[66] A. W. Mohamed, A. A. Hadi, A. K. Mohamed, P. Agrawal, A. Kumar, and P. N. Suganthan, "Problem definitions and evaluation criteria for the cec 2021 on single objective bound constrained numerical optimization," in *CEC*, 2021.

[67] J. Stork, A. E. Eiben, and T. Bartz-Beielstein, "A new taxonomy of global optimization algorithms," *Nat. Comput.*, 2022.

[68] Z.-H. Zhan, L. Shi, K. C. Tan, and J. Zhang, "A survey on evolutionary computation for complex continuous optimization," *Artif. Intell. Rev.*, 2022.

[69] R. Turner, D. Eriksson, M. McCourt, J. Kiili, E. Laaksonen, Z. Xu, and I. Guyon, "Bayesian optimization is superior to random search for machine learning hyperparameter tuning: Analysis of the black-box optimization challenge 2020," in *NIPS*, 2021.

[70] J. H. Holland, "Adaptation in natural and artificial systems: an introductory analysis with applications to biology, control, and artificial intelligence," 1992.

[71] J. Clune, D. Misevic, C. Ofria, R. E. Lenski, S. F. Elena, and R. Sanjuán, "Natural selection fails to optimize mutation rates for long-term adaptation on rugged fitness landscapes," *PLoS Comput. Biol.*, 2008.

[72] F. Wang, G. Xu, and M. Wang, "An improved genetic algorithm for constrained optimization problems," *IEEE Access*, 2023.

[73] R. Storn and K. Price, "Differential evolution-a simple and efficient heuristic for global optimization over continuous spaces," *J. Glob. Optim.*, 1997.

[74] T. Xu, J. He, and C. Shang, "Helper and equivalent objectives: Efficient approach for constrained optimization," *TC*, 2020.

[75] S. Biswas, D. Saha, S. De, A. D. Cobb, S. Das, and B. A. Jalaian, "Improving differential evolution through bayesian hyperparameter optimization," in *CEC*, 2021.

[76] C. Ye, C. Li, Y. Li, Y. Sun, W. Yang, M. Bai, X. Zhu, J. Hu, T. Chi, H. Zhu *et al.*, "Differential evolution with alternation between steady monopoly and transient competition of mutation strategies," *Swarm Evol. Comput.*, 2023.

[77] J. Kennedy and R. Eberhart, "Particle swarm optimization," in *ICNN*, 1995.

[78] Y.-J. Gong, J.-J. Li, Y. Zhou, Y. Li, H. S.-H. Chung, Y.-H. Shi, and J. Zhang, "Genetic learning particle swarm optimization," *TC*, 2015.

[79] D. Wu and G. G. Wang, "Employing reinforcement learning to enhance particle swarm optimization methods," *Eng. Optim.*, 2022.

[80] N. Hansen and A. Ostermeier, "Completely derandomized self-adaptation in evolution strategies," *Evol. Comput.*, 2001.

[81] R. Ros and N. Hansen, "A simple modification in cma-es achieving linear time and space complexity," in *PPSN*, 2008.

[82] N. Hansen, "Benchmarking a bi-population cma-es on the bbob-2009 function testbed," in *GECCO*, 2009.

[83] X. He, Z. Zheng, and Y. Zhou, "Mmes: Mixture model-based evolution strategy for large-scale optimization," *TEVC*, 2020.

[84] J. Snoek, H. Larochelle, and R. P. Adams, "Practical bayesian optimization of machine learning algorithms," in *NIPS*, 2012.

[85] L. Wang, R. Fonseca, and Y. Tian, "Learning search space partition for black-box optimization using monte carlo tree search," *NIPS*, 2020.

[86] S. Kirkpatrick, C. D. Gelatt Jr, and M. P. Vecchi, "Optimization by simulated annealing," *Science*, 1983.

[87] P. J. Van Laarhoven, E. H. Aarts, P. J. van Laarhoven, and E. H. Aarts, *Simulated annealing*, 1987.

[88] Y. Xiang, D. Sun, W. Fan, and X. Gong, "Generalized simulated annealing algorithm and its application to the thomson model," *Physics Letters A*, 1997.

[89] D. Kraft, "A software package for sequential quadratic programming," *Forschungsbericht- Deutsche Forschungs- und Versuchsanstalt fur Luft- und Raumfahrt*, 1988.

[90] A. R. Conn, N. I. Gould, and P. L. Toint, *Trust region methods*, 2000.

[91] M. J. Powell, "A view of algorithms for optimization without derivatives," *Mathematics Today-Bulletin of the Institute of Mathematics and its Applications*, 2007.

[92] R. Bollapragada, J. Nocedal, D. Mudigere, H.-J. Shi, and P. T. P. Tang, "A progressive batching l-bfgs method for machine learning," in *ICML*, 2018.

[93] F.-A. Fortin, F.-M. De Rainville, M.-A. G. Gardner, M. Parizeau, and C. Gagné, "Deap: Evolutionary algorithms made easy," *JMLR*, 2012.

[94] F. Liu, R. Zhang, X. Lin, Z. Lu, and Q. Zhang, "Fine-tuning large language model for automated algorithm design," *arXiv preprint arXiv:2507.10614*, 2025.

[95] L. Zhu, X. Wang, and X. Wang, "Judgelm: Fine-tuned large language models are scalable judges," *arXiv preprint arXiv:2310.17631*, 2023.

[96] X. Zhang, C. Tian, X. Yang, L. Chen, Z. Li, and L. R. Petzold, "Alpacare: Instruction-tuned large language models for medical application," *arXiv preprint arXiv:2310.14558*, 2023.

[97] K. Yang, A. Swope, A. Gu, R. Chalamala, P. Song, S. Yu, S. Godil, R. J. Prenger, and A. Anandkumar, "Leandojo: Theorem proving with retrieval-augmented language models," *Advances in Neural Information Processing Systems*, vol. 36, pp. 21 573–21 612, 2023.

[98] L. Long, R. Wang, R. Xiao, J. Zhao, X. Ding, G. Chen, and H. Wang, "On llms-driven synthetic data generation, curation, and evaluation: A survey," *arXiv preprint*, 2024.

[99] M. Shu, J. Wang, C. Zhu, J. Geiping, C. Xiao, and T. Goldstein, "On the exploitability of instruction tuning," *NeuIPS*, 2023.

[100] D. Guo, S. Lu, N. Duan, Y. Wang, M. Zhou, and J. Yin, "Unixcoder: Unified cross-modal pre-training for code representation," *arXiv preprint*, 2022.

[101] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," *NIPS*, 2017.

[102] T. Wolf, L. Debut, V. Sanh, J. Chaumond, C. Delangue, A. Moi, P. Cistac, T. Rault, R. Louf, M. Funtowicz *et al.*, "Huggingface's transformers: State-of-the-art natural language processing," *arXiv preprint*, 2019.

[103] G. E. Batista, R. C. Prati, and M. C. Monard, "A study of the behavior of several methods for balancing machine learning training data," *ACM SIGKDD*, 2004.

[104] C. Raffel, N. Shazeer, A. Roberts, K. Lee, S. Narang, M. Matena, Y. Zhou, W. Li, and P. J. Liu, "Exploring the limits of transfer learning with a unified text-to-text transformer," *JMLR*, 2020.

[105] Z. Shao, P. Wang, Q. Zhu, R. Xu, J. Song, M. Zhang, Y. Li, Y. Wu, and D. Guo, "Deepseekmath: Pushing the limits of mathematical reasoning in open language models," *arXiv preprint*, 2024.

[106] P. Kerschke, H. H. Hoos, F. Neumann, and H. Trautmann, "Automated algorithm selection: Survey and perspectives," *ECJ*, 2019.

[107] H. Guo, Y. Ma, Z. Ma, J. Chen, X. Zhang, Z. Cao, J. Zhang, and Y.-J. Gong, "Deep reinforcement learning for dynamic algorithm selection: A proof-of-principle study on differential evolution," *TSMC*, 2024.

[108] A. Madaan, S. Zhou, U. Alon, Y. Yang, and G. Neubig, "Language models of code are few-shot commonsense learners," *arXiv preprint*, 2022.

[109] P. Bareiß, B. Souza, M. d'Amorim, and M. Pradel, "Code generation tools (almost) for free? a study of few-shot, pre-trained language models on code," *arXiv preprint*, 2022.

[110] C. A. Floudas and P. M. Pardalos, *A collection of test problems for constrained global optimization algorithms*, 1990.

[111] I. E. Grossmann and R. W. Sargent, "Optimum design of multipurpose chemical plants," *Ind. Eng. Chem. Proc. Design Devel.*, 1979.

[112] A. Osyczka, S. Krenich, and K. Karas, "Optimum design of robot grippers using genetic algorithms," in *WCSMO*, 1999.
[113] Y. Wang, H. Liu, H. Long, Z. Zhang, and S. Yang, "Differential evolution with a new encoding mechanism for optimizing wind farm layout," *TII*, 2017.
[114] A. K. Rathore, J. Holtz, and T. Boller, "Synchronous optimal pulsewidth modulation for low-switching-frequency control of medium-voltage multilevel inverters," *TIE*, 2010.
[115] H. Hwang, T. Vreven, J. Janin, and Z. Weng, "Protein–protein docking benchmark version 4.0," *Proteins:Struct., Funct., Bioinf.*, 2010.
[116] S. P. Arango, H. S. Jomaa, M. Wistuba, and J. Grabocka, "Hpo-b: A large-scale reproducible benchmark for black-box hpo based on openml," *arXiv preprint*, 2021.
[117] F. P. Such, V. Madhavan, E. Conti, J. Lehman, K. O. Stanley, and J. Clune, "Deep neuroevolution: Genetic algorithms are a competitive alternative for training deep neural networks for reinforcement learning," *arXiv preprint*, 2017.
[118] M. A. Shehadeh and J. Kudela, "Benchmarking global optimization techniques for unmanned aerial vehicle path planning," *Expert Systems with Applications*, 2025.
[119] C. A. Floudas, A. R. Ciric, and I. E. Grossmann, "Automatic synthesis of optimum heat exchanger network configurations," *AIChE Journal*, 1986.
[120] H. S. Ryoo and N. V. Sahinidis, "Global optimization of nonconvex nlps and minlps with applications in process design," *Computers & Chemical Engineering*, 1995.
[121] N. Andrei and N. Andrei, *Nonlinear optimization applications using the GAMS technology*, 2013.
[122] G. Steven, "Evolutionary algorithms for single and multicriteria design optimization," *Structural and Multidisciplinary Optimization*, 2002.
[123] S. Gupta, R. Tiwari, and S. B. Nair, "Multi-objective design optimisation of rolling bearings using genetic algorithms," *Mechanism and Machine Theory*, 2007.
[124] E. Sandgren, "Nonlinear integer and discrete programming in mechanical design," in *International design engineering technical conferences and computers and information in engineering conference*. American Society of Mechanical Engineers, 1988.
[125] D. D. Uyeh, R. Mallipeddi, T. Pamulapati, T. Park, J. Kim, S. Woo, and Y. Ha, "Interactive livestock feed ration optimization using evolutionary algorithms," *Computers and Electronics in Agriculture*, 2018.
[126] A. Kumar, G. Wu, M. Z. Ali, R. Mallipeddi, P. N. Suganthan, and S. Das, "A test-suite of non-convex constrained optimization problems from the real-world and some baseline results," *Swarm Evol. Comput.*, 2020.
[127] A. Kostovska, A. Jankovic, D. Vermetten, S. Džeroski, T. Eftimov, and C. Doerr, "Comparing algorithm selection approaches on black-box optimization problems," in *GECCO*, 2023.
[128] X. Wu, Y. Zhong, J. Wu, B. Jiang, and K. C. Tan, "Large language model-enhanced algorithm selection: towards comprehensive algorithm representation," in *AAAI*, 2024.
[129] M. Lindauer, K. Eggensperger, M. Feurer, A. Biedenkapp, D. Deng, C. Benjamins, T. Ruhkopf, R. Sass, and F. Hutter, "Smac3: A versatile bayesian optimization package for hyperparameter optimization," *JMLR*, 2022.
[130] OpenAI, *Knowledge retrieval documentations*, 2023. [Online]. Available: https://platform.openai.com/docs/assistants/tools/knowledge-retrieval

Zeyuan Ma received the B.Eng. degree in the school of Computer Science and Engineering, South China University of Technology, Guangzhou, China, in 2022. He is currently pursuing the Ph.D. degree with the School of Computer Science and Engineering, South China University of Technology, China. He is working at the intersection of Machine Learning (ML) and Optimization. In particular, his research interests include Deep Reinforcement Learning (DRL), Black-Box Optimization (BBO), Meta-Black-Box Optimization (MetaBBO). He has (co-) authored over 20 publications, most of which are published in top-tier conferences and journals such as ICML, ICLR, NeurIPS, AAAI and IEEE TEVC. He also actively serves as reviewers for these advanced conferences and journals. Recently, he organized LEAD 2025 Workshop, and has been invited as program committee member of Future Computing 2026 and editorial board member of Journal of Intelligent and Sustainable Systems.

Yue-Jiao Gong (S'10-M'15-SM'19) received the B.S. and Ph.D. degrees in Computer Science from Sun Yat-sen University, China, in 2010 and 2014, respectively. She is currently a Full Professor at the School of Computer Science and Engineering, South China University of Technology, China. Her research interests include optimization methods based on swarm intelligence, deep learning, reinforcement learning, and their applications in smart cities and intelligent transportation. She has published over 100 papers, including more than 60 in ACMIEEE TRANSACTIONS and over 50 at renowned conferences such as NeurIPS, ICLR, and GECCO. Dr. Gong was awarded the Pearl River Young Scholar by the Guangdong Education Department in 2017 and the Guangdong Natural Science Funds for Distinguished Young Scholars in 2022. She was named to the Stanford World's Top 2% Scientists list in the artificial intelligence field. She is currently an Associate Editor of IEEE Transactions on Evolutionary Computation and ACM Transactions on Evolutionary Learning and Optimization, and has served as an Area Chair or Senior PC member for a few leading conferences.

Hongshu Guo received the B.Eng. degree in the school of Computer Science and Engineering, South China University of Technology, Guangzhou, China, in 2022. He is currently pursuing the Ph.D. degree with the School of Computer Science and Engineering, South China University of Technology, China. His research interests include deep reinforcement learning and evolutionary computing.

Jiacheng Chen is a PhD student in the Department of Computer Science and Engineering at The Chinese University of Hong Kong (CUHK), advised by Prof. Yu Cheng and Prof. Weiyang Liu. He obtained his Bachelor of Engineering degree from the School of Computer Science and Technology, South China University of Technology (SCUT), where he was mentored by Prof. Yue-Jiao Gong. His research interests primarily focus on reasoning in natural language processing (NLP) and reinforcement learning.

Yining Ma received the Ph.D. degree in Industrial Systems Engineering from the National University of Singapore, Singapore, in 2024, and the B.E. degree in Computer Science from the South China University of Technology, Guangzhou, China, in 2019, respectively. He is currently a Postdoctoral Associate at the Laboratory for Information and Decision Systems (LIDS), Massachusetts Institute of Technology (MIT), USA. Prior to that, he was a Research Fellow at the College of Computing and Data Science (CCDS), Nanyang Technological University, Singapore. His research focuses on the intersection of artificial intelligence (AI) and optimization, also known as the field of learning to optimize (L2Opt), and extends to broader AI domains such as LLM, multi-agent systems, and computational intelligence.

Zhiguang Cao received the B.Eng. degree in automation from Guangdong University of Technology, Guangzhou, China, the M.Sc. degree in signal processing from Nanyang Technological University, Singapore, and the Ph.D. degree from the Interdisciplinary Graduate School, Nanyang Technological University. He was previously a Research Fellow with the Energy Research Institute @ NTU (ERI@N), a Research Assistant Professor with the Department of Industrial Systems Engineering and Management, National University of Singapore, and a Scientist with the Agency for Science, Technology and Research (A*STAR), Singapore. He is currently an Assistant Professor with the School of Computing and Information Systems, Singapore Management University. His research focuses on learning to optimize (L2Opt) and AI-driven combinatorial optimization.

Jun Zhang (FIEEE'17) received his PhD degree in Electrical Engineering from the City University of Hong Kong in 2002. Professor Zhang's research activities are mainly in the areas of Computational Intelligence(CI). Based on his research in Evolutionary Computation(EC) and its applications, Professor Zhang Jun has published more than 600 peer-reviewed research papers, of which more than 200 have been published in IEEE Transactions. Professor Zhang is Clarivate Highly Cited Researcher rank in the top 1% for field in Computer Science, and was awarded the "Outstanding Young Scientist Fund" by NSFC in 2011, and was appointed as a "Changjiang Chair Professor" in 2013. He currently serves as Associate Editor of IEEE Transactions on Artificial Intelligence and IEEE Transactions on Cybernetics.