FunBO: Discovering Acquisition Functions for Bayesian Optimization with FunSearch

Anonymous authors
Paper under double-blind review

Abstract

The sample efficiency of Bayesian optimization algorithms depends on carefully crafted acquisition functions (AFs) guiding the sequential collection of function evaluations. The best-performing AF can vary significantly across optimization problems, often requiring ad-hoc and problem-specific choices. This work tackles the challenge of designing novel AFs that perform well across a variety of experimental settings. Based on FunSearch, a recent work using Large Language Models (LLMs) for discovery in mathematical sciences, we propose FunBO, an LLM-based method that can be used to learn new AFs written in computer code by leveraging access to a limited number of evaluations for a set of objective functions. We provide the analytic expression of all discovered AFs and evaluate them on various global optimization benchmarks and hyperparameter optimization tasks. We show how FunBO identifies AFs that generalize well in *and* out of the training distribution of functions, thus outperforming established general-purpose AFs and achieving competitive performance against AFs that are customized to specific function types and are learned via transfer-learning algorithms.

1 Introduction

Bayesian optimization (BO) (Jones et al., 1998; Mockus, 1974) is a powerful methodology for optimizing complex and expensive-to-evaluate black-box functions which emerge in many scientific disciplines. BO has been used across a large variety of applications ranging from hyperparameter tuning in machine learning (Bergstra et al., 2011; Snoek et al., 2012; Cho et al., 2020) to designing policies in robotics (Calandra et al., 2016) and recommending new molecules in drug design (Korovina et al., 2020). Two main components lie at the heart of any BO algorithm: a surrogate model and an acquisition function (AF). The surrogate model expresses assumptions about the objective function, e.g., its smoothness, and it is often given by a Gaussian Process (GP) (Rasmussen & Williams, 2006). Based on the surrogate model, the AF determines the sequential collection of function evaluations by assigning a score to potential observation locations. BO's success heavily depends on the AF's ability to efficiently balance exploitation (i.e. assigning a high score to locations that are likely to yield optimal function values) and exploration (i.e. assigning a high score to regions with higher uncertainty about the objective function in order to inform future decisions), thus leading to the identification of the optimum with the minimum number of evaluations.

Existing AFs aim to provide either general-purpose optimization strategies or approaches tailored to specific objective types. For example, Expected Improvement (EI) (Mockus, 1974), Upper Confidence Bound (UCB) (Lai & Robbins, 1985) and Probability of Improvement (PofI) (Kushner, 1964) are all widely adopted *general-purpose* AFs that can be used out-of-the-box across BO algorithms and objective functions. The performance of these AFs varies significantly across different types of black-box functions, making the AF choice an ad-hoc, empirically driven, decision. There exists an extensive literature on alternative AFs outperforming EI, UCB and PofI, for instance entropy-based (Wang & Jegelka, 2017) or knowledge-gradient (Frazier et al., 2008) optimizers, see Garnett (2023, Chapter 7) for a review. However, while these functions are often interpretable as they can be written as the expectation of a utility function, they are generally hard to implement and expensive to evaluate, partly defeating the purpose of replacing the expensive original optimization with the optimization of a much cheaper and faster to evaluate AF. In other to avoid the limitations of current AFs, several works have proposed self-adjusting the hyper-parameters of known AFs in a data driven way throughout the optimization process (Benjamins et al., 2023; Ding et al., 2022) or combining different AFs in a portfolio and selecting them via an online multi-armed bandit strategy (Hoffman et al., 2011). Other prior works (Hsieh et al., 2021; Volpp et al., 2020; Wistuba & Grabocka, 2021) have instead proposed representing AFs via neural networks thus bypassing the need for an analytical representation and and learning new AFs tailored to specific objectives by transferring information from a set of related functions with a given training distribution via, e.g., reinforcement learning or transformers. While such learned AFs can outperform general-purpose AFs, their generalization performance to objectives outside of the training distribution is often poor (see experimental section and discussion on generalization behaviour in Volpp et al. (2020)). More recently, the concurrent work of Yao et al. (2024) investigated representing AFs in code for specific optimization settings where the experimentation budget is limited. Defining methodologies that *automatically* identify new AFs capable of outperforming general-purpose *and* function-specific alternatives, both in and out of the training distribution, remains a significant and unaddressed challenge. In this work we tackle this challenge by considering AFs represented in computer code. Learning new AFs expressed in code presents three main difficulties: (i) the vast space of all possible programs makes exhaustive search infeasible, (ii) efficiently exploring a constrained space of possible programs requires scalable methods and (iii) there is no clear criteria for ensuring the validity and effectiveness of generated AFs.

Contributions. We overcome these difficulties by formulating the problem of learning novel AFs written in computer code as an algorithm discovery problem and address it by extending FunSearch (Romera-Paredes et al., 2023), a recently proposed algorithm that uses LLMs to solve open problems in mathematical sciences. In particular, we introduce FunBO, a novel method that explores the large space of AFs written in computer code by taking an initial AF as input and, with a limited number of evaluations for a set of objective functions, iteratively modifying it to improve the performance of the resulting BO algorithm. We focus on Python programs but develop an algorithm that can be readily applied to other languages supported by FunSearch, such as C++. Unlike existing algorithms, FunBO outputs code snippets corresponding to improved AFs, which can be inspected to (i) identify differences with respect to known AFs, (ii) investigate the reasons for their observed performance, thereby enforcing *interpretability*, and (iii) be easily deployed in practice without additional infrastructure overhead. We extensively test FunBO on a range of optimization problems including standard global optimization benchmarks and hyperparameter optimization (HPO) tasks. For each experiment, we report the explicit functional form of the discovered AFs and show that they generalize well to the optimization of functions both in and out of the training distribution, outperforming general-purpose AFs while comparing favorably to function-specific ones. To the best of our knowledge, this is the first work exploring AFs represented in computer code, thus demonstrating a novel approach to harness the power of LLMs for sampling policy design.

2 PRELIMINARIES

We consider an expensive-to-evaluate black-box function $f : \mathcal{X} \rightarrow \mathbb{R}$ over the input space $\mathcal{X} \subseteq \mathbb{R}^d$ for which we aim at identifying the global minimum $\boldsymbol{x}^* = \arg\min_{\boldsymbol{x}\in\mathcal{X}} f(\boldsymbol{x})$. We assume access to a set of auxiliary black-box and expensive-to-evaluate objective functions, $\mathcal{G} = \{g_j\}_{j=1}^J$, with $g_j : \mathcal{X}_j \rightarrow \mathbb{R}, \mathcal{X}_j \subseteq \mathbb{R}^{d_j}$ for $j = 1, \ldots, J$, from which we can obtain a set of evaluations.

Bayesian optimization. BO seeks to identify $\boldsymbol{x}^*$ with the smallest number T of sequential evaluations of f given N initial observations $\mathcal{D} = \{\boldsymbol{x}_i, y_i\}_{i=1}^N$, with $y_i = f(\boldsymbol{x}_i)$.[1] BO relies on a probabilistic surrogate model for f which in this work is set to a GP with prior distribution over any batch of input points $\boldsymbol{X} = \{\boldsymbol{x}_1, \ldots, \boldsymbol{x}_N\}$ given by $p(f|\boldsymbol{X}) = \mathcal{N}(m(\boldsymbol{X}), K_\theta(\boldsymbol{X}, \boldsymbol{X}'))$ with prior mean $m(\boldsymbol{X})$ and kernel $K_\theta(\boldsymbol{X}, \boldsymbol{X}')$ with hyperparameters θ. The posterior distribution $p(f|\mathcal{D})$ is available in closed form via standard GP updates. At every step t in the optimization process, BO selects the next evaluation location by optimizing an AF $\alpha(\cdot|\mathcal{D}_t) : \mathcal{X} \rightarrow \mathbb{R}$, given the current posterior distribution $p(f|\mathcal{D}_t)$, with $\mathcal{D}_t$ denoting the function evaluations collected up to trial t (including $\mathcal{D}$). A commonly used AF is the Expected Improvement (EI), which is defined as $\alpha_{\text{EI}}(\boldsymbol{x}|\mathcal{D}_t) = (y^* - m(\boldsymbol{x}|\mathcal{D}_t))\Phi(z) + \sigma(\boldsymbol{x}|\mathcal{D}_t)\phi(z)$, where y^* denotes the best function value observed in $\mathcal{D}_t$, also called incumbent, $z = (y^* - m(\boldsymbol{x}|\mathcal{D}_t))/\sigma(\boldsymbol{x}|\mathcal{D}_t)$, ϕ and Φ are the standard Normal density and distribution functions, and $m(\boldsymbol{x}|\mathcal{D}_t)$ and $\sigma(\boldsymbol{x}|\mathcal{D}_t)$ are the GP posterior mean and standard deviation computed at $\boldsymbol{x} \in \mathcal{X}$. Other general-purpose AFs proposed in the literature are: UCB ($\alpha_{\text{UCB}}(\boldsymbol{x}|\mathcal{D}_t) = m(\boldsymbol{x}|\mathcal{D}_t) - \beta\sigma(\boldsymbol{x}|\mathcal{D}_t)$

[1] We focus on noiseless observations but the method can be equivalently applied to noisy outcomes.

with hyperparameter β), PofI ($\alpha_{\text{PofI}}(\boldsymbol{x}|\mathcal{D}_t) = \Phi((y^* - m(\boldsymbol{x}|\mathcal{D}_t))/\sigma(\boldsymbol{x}|\mathcal{D}_t))$) and the posterior mean $\alpha_{\text{MEAN}}(\boldsymbol{x}|\mathcal{D}_t) = m(\boldsymbol{x}|\mathcal{D}_t)$ (denoted by MEAN hereinafter).[2]

Unlike general-purpose AFs, several works have proposed increasing the efficiency of BO for a specific optimization problem, say the optimization of f, by either adaptively selecting and/or adjusting known AFs in a data-driven manner (Benjamins et al., 2023) or by *learning* problem-specific AFs (Hsieh et al., 2021; Volpp et al., 2020; Wistuba & Grabocka, 2021). The learned AFs are trained on the set $\mathcal{G}$, whose functions are assumed to be drawn from the same distribution or function class associated to f, reflecting a meta-learning setup. "Function class" here refers to a set of functions with a shared structure and obtained by, e.g., applying scaling and translation transformations to their input and output values or evaluating the loss function of the same machine learning model, e.g., AdaBoost, on different data sets. For instance, Wistuba et al. (2018) learns an AF that is a weighted superposition of EIs by exploiting access to a sufficiently large dataset for functions in $\mathcal{G}$. Volpp et al. (2020) considered settings where the observations for functions in $\mathcal{G}$ are limited and proposed MetaBO, a reinforcement learning based algorithm that learns a specialized neural AF, i.e., a neural network representing the AF. The neural AF takes as inputs a set of potential locations (with a given d), the posterior mean and variance at those points, the trial t and the budget T and is trained using a proximal policy optimization algorithm (Schulman et al., 2017). Similarly, Hsieh et al. (2021) proposed FSAF, an AF obtained via few-shot adaptation of a learned AF using a small number of function instances in $\mathcal{G}$. Note that, while general-purpose AFs are used to optimize objectives across function classes, learned AFs aim at achieving high performance for the single function class to which f and $\mathcal{G}$ belong.

FunSearch. FunSearch (Romera-Paredes et al., 2023) is a recently proposed evolutionary algorithm for *search*ing in the *fun*ctional space by combining a pre-trained LLM used for generating new computer programs with an efficient evaluator, which guards against hallucinations and scores fitness. An example problem that FunSearch tackles is the online bin packing problem (Coffman et al., 1984), where a set of items of various sizes arriving online needs to be packed into the smallest possible number of fixed sized bins. A set of heuristics have been designed for deciding which bin to assign an incoming item to, e.g., "first fit." FunSearch aims at discovering new heuristics that improve on existing ones by taking as inputs: (i) the computer code of an `evolve` function $h(\cdot)$ representing the initial heuristic to be improved by the LLM, e.g., "first fit" and (ii) an `evaluate` function $e(h, \cdot)$, also written in computer code, specifying the problem at hand (also called "problem specification") and scoring each $h(\cdot)$ according to a predefined performance metric, e.g., the number of bins used in $h(\cdot)$. The inputs of both $h(\cdot)$ (denoted by h hereinafter) and $e(h, \cdot)$ (denoted by e hereinafter), are problem specific. A description of h's inputs is provided in the function's docstring[3] together with an explanation of how the function itself is used within e. Given these initial components, FunSearch prompts an LLM to propose an improved h, scores the proposals on a set of inputs, e.g., on different bin-packing instances, and adds them to a programs database. The programs database stores correct h functions[4] together with their respective scores. In order to encourage diversity of programs and enable exploration of different solutions, a population-based approach inspired by genetic algorithms (Tanese, 1989) is adopted for the programs database (DB). At a subsequent step, functions in the database are sampled to create a new prompt, LLM's proposals are scored and stored again. The process repeats for $\tau = 1, \dots, \mathcal{T}$ until a time budget $\mathcal{T}$ is reached and the heuristic with the highest score on a set of inputs is returned.

3 FUNBO

FunBO is a FunSearch-based method for discovering novel AFs that increase BO efficiency by exploiting the set of auxiliary objectives $\mathcal{G}$. In particular, FunBO (i) uses the same prompt and DB structure as FunSearch, but (ii) proposes a new problem specification by viewing the learning of AFs as a algorithm discovery problem, and (iii) introduces a novel initialization and evaluation pipeline that is used within the FunSearch structure. FunBO does not make assumptions about similarities between f and $\mathcal{G}$, nor assumes access to a large dataset for each function in $\mathcal{G}$. Therefore, FunBO can be used to

[2]We focus on AFs that can be evaluated in closed form given the posterior parameters of a GP surrogate model and exclude those whose computation involve approximations, e.g., Monte-Carlo sampling.

[3]We focus on Python programs.

[4]The definition of a correct function is also problem specific. For instance, a program can be considered correct if it compiles.

Inputs: $\mathcal{G}_{\text{Tr}}$, $\mathcal{G}_{\text{V}}$, N_{DB}, B, $\mathcal{T}$
Setup: Initialize h (Fig. 2, Top), e (Fig. 9-10) and DB with N_{DB} islands. Assign h to each island.
while $\tau < \mathcal{T}$ **do**
1. Sample two programs from DB and create prompt (Fig. 2)
2. Get a batch of B samples from the LLM
3. For each correct h^{τ} in the batch compute $s_{h^{\tau}}(\mathcal{G}_{\text{Tr}})$
4. Add correct h^{τ} to DB and update it (see Appendix B)
5. Update step $\tau = \tau + 1$

end
Output: Return h in DB with score in the top 20th percentile for $\mathcal{G}_{\text{Tr}}$ and highest score on $\mathcal{G}_{\text{V}}$.

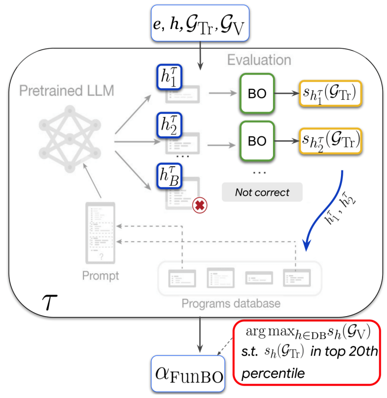

Figure 1: *Left*: The FunBO algorithm. *Right*: Graphical representation of FunBO. The different FunBO component w.r.t. FunSearch (Romera-Paredes et al., 2023, Fig. 1) are highlighted in color.

discover both general-purpose and function-specific AFs as well as to adapt AFs via few-shots. FunBO leverages the LLMs' ability to generate executable code to make the search for novel AFs automatic and scalable, potentially leveraging the extensive LLMs' knowledge of BO and AFs while delivering more interpretable AFs than those represented by neural networks. Furthermore, while FunSearch was only applied to problems that required evolving functions with simple inputs (integers, floats or short tuples; with only one application taking as input a single array), FunBO explores a significantly more complex function space where programs take as inputs multiple arrays. This demonstrates how the same formulation can be applied to problems of increasing complexity as long as an appropriate scoring mechanism is identified.

Method overview. FunBO sequentially prompts an LLM to improve an **initial AF** expressed in code so as to enhance the performance of the corresponding BO algorithm when optimizing objectives in $\mathcal{G}$. At every step τ of FunBO, an LLM's **prompt** is created by including the code for two AF instances generated and stored in a **programs database** (DB) at previous iterations. With this prompt, a number (B) of alternative AFs are sampled from the LLM and are evaluated based on their average performance on a subset $\mathcal{G}_{\text{Tr}} \subseteq \mathcal{G}$, which acts as training dataset. The **evaluation** process for an AF, say h^{τ} at step τ, on $\mathcal{G}_{\text{Tr}}$ gives a numeric score $s_{h^{\tau}}(\mathcal{G}_{\text{Tr}})$ that is used to store programs in DB and sample them for subsequent prompts. The "process" of prompt creation, LLM sampling, and AF scoring and storing repeats until time budget $\mathcal{T}$ is reached. Out of the top performing[5] AFs on $\mathcal{G}_{\text{Tr}}$, the algorithm returns the AF performing the best, on average, in the optimization of $\mathcal{G}_{\text{V}} = \mathcal{G} \backslash \mathcal{G}_{\text{Tr}}$, which acts as a validation dataset. When no validation functions are used ($\mathcal{G} = \mathcal{G}_{\text{Tr}}$), the AF with the highest average performance on $\mathcal{G}_{\text{Tr}}$ is returned. Each FunBO component highlighted in bold is described below in more details, along with the complete algorithm and graphical representation in Fig. 1. We denote the AF returned by FunBO as α_{FunBO}.

Initial AF. FunBO's initial program h determines the input variables that can be used to generate alternative AFs while imposing a prior on the programs the LLM will generate at successive steps. For these reasons it is important for guiding the search process effectively. We consider `acquisition_function` in Fig. 2 (top) which takes the functional form of the EI and has as inputs the union of the inputs given to EI, UCB and PofI. The AF returns an integer representing the index of the point in a vector of potential locations that should be selected for the next function evaluation. All programs generated by the LLM share the same inputs and output, but vary in their implementation, which defines different optimization strategies, see for instance the AF generated for one of our experiments in Fig. 3 (left).[6]

Prompt. At every algorithm iteration, a prompt is constructed by sampling two AFs, h_i and h_j, previously generated and stored in DB. h_i and h_j are sampled from DB in a way that favours higher scoring and shorter programs (see paragraph below for more details) and are sorted in the prompt

[5] In this work we consider the programs with score in the top 20th percentile.

[6] We explored using a random selection of initial points as an alternative to EI. However, this approach did not yield good results as using a random selection was incentivizing the generation of functions with a stochastic output, for which convergence results are not reproducible.

```
def acquisition_function(predictive_mean, predictive_var, incumbent, beta=1.0):
  """Returns the index of the point to collect ... (Full docstring in Fig. 8)."""
  z = (incumbent - predictive_mean) / np.sqrt(predictive_var)
  predictive_std = np.sqrt(predictive_var)
  vals = (incumbent - predictive_mean) * stats.norm.cdf(z) + predictive_std * stats.norm.pdf(z)
  return np.argmax(vals)
```

```
"""Improve Bayesian Optimization by discovering a new acquisition function."""

def acquisition_function_v0(predictive_mean, predictive_var, incumbent, beta=1.0):
    """Returns the index of the point to collect ... (Full docstring in Fig. 8)"""
    # Code for lowest-scoring sampled AF.
    return ...

def acquisition_function_v1(predictive_mean, predictive_var, incumbent, beta=1.0):
    """Improved version of `acquisition_function_v0`."""
    # Code for highest-scoring sampled AF.
    return ...

def acquisition_function_v2(predictive_mean, predictive_var, incumbent, beta=1.0):
    """Improved version of the previous `acquisition_function`."""
```

Figure 2: *Top*: FunBO's initial AF takes the functional form of EI with inputs given by the posterior parameter of the GP at a set of potential sample locations, the incumbent and a parameter $\beta = 1$. *Bottom*: FunBO prompt includes two previously generated AFs which are sampled from DB and are sorted in ascending order based on the score achieved on $\mathcal{G}_{\text{Tr}}$. The LLM generates a third AF, `acquisition_function_v2`, representing an improved version of the highest scoring program.

in ascending order based on their scores $s_{h_i}(\mathcal{G}_{\text{Tr}})$ and $s_{h_j}(\mathcal{G}_{\text{Tr}})$, see the prompt skeleton[7] in Fig. 2 (bottom). The LLM is then asked to generate a new AF representing an improved version of the last, higher scoring, program.

Evaluation. As expected, the evaluation protocol is critical for the discovery of appropriate AFs. Our novel evaluation setup, unlike the one used in FunSearch, entails performing a full BO loop to evaluate program fitness. In particular, each function generated by the LLM is (i) checked to verify it is correct, i.e., it compiles and returns a numerical output; (ii) scored based on the average performance of a BO algorithm using h^τ as an AF on $\mathcal{G}_{\text{Tr}}$. Evaluation is performed by running a full BO loop with h^τ for each function $g_j \in \mathcal{G}_{\text{Tr}}$ and computing a score that contains two terms: a term that rewards AFs finding values close to the true optimum, and a term that rewards AFs finding the optimum in fewer evaluations (often called trials). Specifically, we use the score:

$$s_{h^\tau}(\mathcal{G}_{\text{Tr}}) = \frac{1}{|\mathcal{G}_{\text{Tr}}|}\sum_{j=1}^{J}\left[\left(1 - \frac{g_j(\boldsymbol{x}^*_{j,h^\tau}) - y^*_j}{g_j(\boldsymbol{x}^{t=0}_j) - y^*_j}\right) + \left(1 - \frac{T_{h^\tau}}{T}\right)\right] \tag{1}$$

where, for each g_j, y^*_j is the known true optimum, $\boldsymbol{x}^{t=0}_j$ gives the optimal input value at $t = 0$ which is assumed to be different from the true one, $\boldsymbol{x}^*_{j,h^\tau}$ is the found optimal input value with h^τ and T_{h^τ} gives the number of trials out of T that h^τ selected before reaching y^*_j (if the optimum was not found, then $T_{h^\tau} = T$ to indicate that all available trials have been used). The first term in the square brackets of Eq. (1) quantifies the discrepancy between the function values at the returned optimum and the true optimum. This term becomes zero when $\boldsymbol{x}^*_{j,h^\tau}$ equals $\boldsymbol{x}^{t=0}_j$, indicating a failure to explore the search space. Conversely, if h^τ successfully identifies the true optimum, such that $g_j(\boldsymbol{x}^*_{j,h^\tau}) = y^*_j$, this term reaches its maximum value of one. The second term in Eq. (1) captures how quickly h^τ identifies y^*_j. When $T_{h^\tau} = T$, indicating the algorithm has not converged, this term becomes zero, and the score is solely determined by the discrepancy between the discovered and true optimum. If,

[7] Note that, when $\tau = 1$, only the initial program will be available in DB thus the prompt in Fig. 2 will be simplified by removing `acquisition_function_v1` and replacing `v_2` with `v_1`.

```
def acquisition_function(predictive_mean
    predictive_var, incumbent, beta=1.0):
  """Returns the index of the point to collect.
  predictive_std = np.sqrt(predictive_var)
  diff_mean_std = (incumbent - predictive_mean
  + beta * predictive_std)
  z = diff_mean_std / predictive_std
  vals = (diff_mean_std * stats.norm.cdf(z)
      + predictive_std * stats.norm.pdf(z))
  return np.argmax(vals)
```

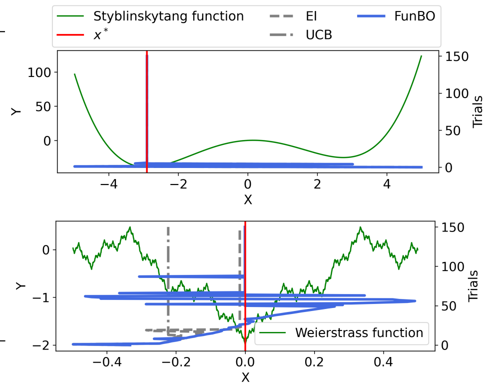

Figure 3: OOD-Bench. *Left:* Code for α_{FunBO}. *Right*: Different AFs trading-off exploration and exploitation for two one-dimensional objective functions (green lines). Blue and gray trajectories track the points queried by α_{FunBO}, EI and UCB over 150 steps (right y-axis). All AFs behave similarly for Styblinski-Tang (top, note that trajectories are overlapping), converging to the true optimizer (red vertical line) in fewer than 25 trials. Instead, for Weierstrass (bottom), EI and UCB get stuck after a few trials while α_{FunBO} continues to explore, eventually converging to the ground truth optimum.

instead, the algorithm reaches the global optimum, this term represents the proportion of trials, out of the total budget T, needed to do so. As an alternative scoring mechanism, we considered: (i) a binary score giving 0 or 1 based on the convergence of the optimization problem to the global optimum, and (ii) the negative normalized cumulative regret. We found (i) to not provide enough signal during the exploration phase. A scoring mechanism that captures small improvements in the proposed AF is needed to steer the LLM toward promising regions of the function space. At the same time, we did not find (ii) to provide significant advantages over the currently adopted scoring mechanism. BO algorithms with simple Code for the evaluation process is presented in Appendix A.

Programs database. Similar to FunSearch, scored AFs are added to DB, which keeps a population of correct programs following an island model (Tanese, 1989). DB is initialized with a number N_{DB} of islands that evolve independently. Sampling of h_i and h_j from DB is done by first uniformly sampling an island and, within that island, sampling programs by favouring those that are shorter and higher scoring. A new program generated when using h_i and h_j in the prompt is added to the same island and, within that, to a cluster of programs performing similarly on $\mathcal{G}_{\text{Tr}}$, see Appendix B for more details.

4 Experiments

Our experiments explore FunBO's ability to generate novel and efficient AFs across a wide variety of settings. In particular, we demonstrate its potential to generate AFs that generalize well to the optimization of functions both in distribution (ID, i.e. within function classes) and out of distribution (OOD, i.e. across function classes) by running three different types of experiments:

1. OOD-Bench tests generalization across function classes by running FunBO with $\mathcal{G}$ containing different standard global optimization benchmarks and testing on a set $\mathcal{F}$ that similarly comprises diverse functions in terms of smoothness, input ranges and dimensionality and output magnitudes. We do not scale the output values nor normalise the input domains to facilitate learning, but rather use the objective functions as available in standard BO packages out-of-the-box. In this case $\mathcal{G}$ and $\mathcal{F}$ do not share any particular structure, thus the generated AFs are closer to general-purpose AFs.
2. ID-Bench, HPO-ID and GPs-ID test FunBO-generated AFs within function classes for standard global optimization benchmarks, HPO tasks, and general function classes, respectively. As this setting is closer to the one considered by meta-learning approaches introduced in Section 2, we compare FunBO against MetaBO (Volpp et al., 2020),[8] the state-of-the-art transfer AF.

[8]We used the author-provided implementation at `https://github.com/boschresearch/MetaBO`.

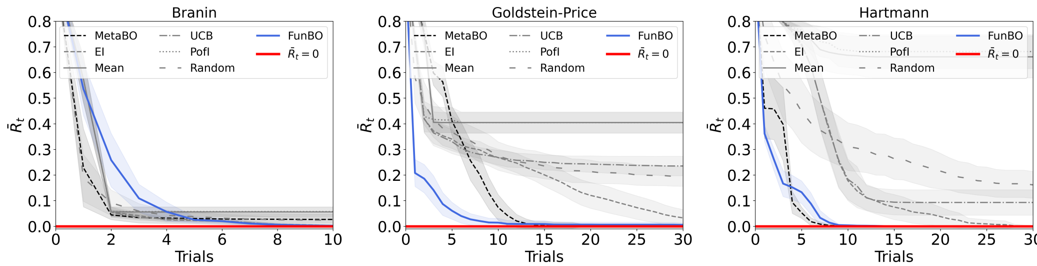

Figure 5: ID-Bench. Average BO performance when using known general purpose AFs (gray lines), the AF learned by MetaBO (black dashed line) and α_{FunBO} (blue line) on 100 function instances. Shaded area gives ± standard deviations/2. The red line represents $\bar{R}_t = 0$, i.e. zero average regret.

3. FEW-SHOT demonstrates how FunBO can be used in the context of few-shot fast adaptation of an AF. In this case, the AF is learnt using a general function class as $\mathcal{G}$ and is then tuned, using a very small (5) number of examples, to optimize a specific synthetic function. We compare our approach to Hsieh et al. (2021),[9] the most relevant few-shot learning method.

We report all results in terms of normalized average simple regret on a test set, $\bar{R}_t$, as a function of the trial t. For an objective function f, this is defined as $R_t = f(\boldsymbol{x}_t^*) - y^*$ where y^* is the true optimum and $\boldsymbol{x}_t^*$ is the best selected point within the data collected up to t. As $\mathcal{F}$ might include functions with different scales, we normalize the regret values to be in $[0, 1]$ before averaging them. To isolate the effects of different acquisition functions, we employ the same setting across all methods in terms of (i) number of trials T, (ii) hyperparameters of the GP surrogate models (tuned offline), (iii) evaluation grid for the AF, which is set to be a Sobol grid (Sobol', 1967) on the input space, and (iv) initial design, which includes the input point giving the maximum function value on the grid. Note that here we use a GP model with zero mean function and RBF kernel across experiments. Therefore, the discovered AFs are conditioned on this choice of surrogate model. All experiments are conducted using FunSearch with default hyperparameters in Romera-Paredes et al. (2023)[10] unless otherwise stated. We employ Codey, an LLM fine-tuned on a large code corpus and based on the PaLM model family (Google-PaLM-2-Team, 2023), to generate AFs.[11]

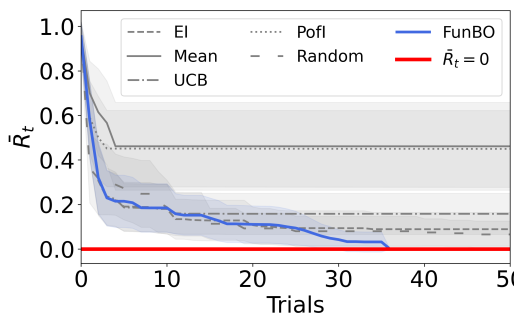

Figure 4: OOD-Bench. Average BO performance when using known general purpose AFs and α_{FunBO}. Shaded area gives ± standard deviations/2. The red line gives $\bar{R}_t = 0$, i.e. zero average regret.

OOD-Bench. We test the capabilities of FunBO to generate an AF that performs well *across* function classes by including the one-dimensional functions Ackley, Levy, and Schwefel in $\mathcal{G}_{\text{Tr}}$ and using the one-dimensional Rosenbrock function for $\mathcal{G}_{\text{V}}$. We test the resulting α_{FunBO} on nine very different objective functions: Sphere ($d = 1$), Styblinski-Tang ($d = 1$), Weierstrass ($d = 1$), Beale ($d = 2$), Branin ($d = 2$), Michalewicz ($d = 2$), Goldstein-Price ($d = 2$) and Hartmann with both $d = 3$ and $d = 6$. We do not compare against MetaBO as (i) it was developed for settings in which the functions in $\mathcal{G}$ and $\mathcal{F}$ belong to the same class and, (ii) the neural AF is trained with evaluation points of a given dimension, thus it cannot be deployed for the optimization of functions across different d. For completeness, we report a comparison with a dimensionality-agnostic version of MetaBO in Appendix C.1 (Fig. 11) together with all experimental details, e.g., input ranges and hyperparameter settings.

[9] We used the author-provided implementation at `https://github.com/pinghsieh/FSAF`.

[10] See code at `https://github.com/google-deepmind/funsearch`.

[11] Codey is publicly accessible via its API (Vertex AI, 2023). For AF sampling, we used 5 Codey instances running on tensor processing units on a computing cluster. For scoring, we used 100 CPUs evaluators per LLM instance.

AF *interpretation:* In this experiment, α_{FunBO} (Fig. 3, left) represents a combination of EI and UCB which, due to the `beta*predictive_std` term, is more exploratory than EI but, considering the incumbent value, still factors in the expected magnitude of the improvement and reduces to EI when `beta=0`. This determines the way α_{FunBO} trades-off exploration and exploitation which can be visualized by looking at the "exploration path", i.e., the sequence of $\boldsymbol{x}$ values selected over t, as shown in the right plots of Fig. 3 (t measured on the secondary y-axis). For objective functions that are smooth, for example Styblinski-Tang (top plot), the exploration path of α_{FunBO} matches those of EI and UCB. In this scenario, all AFs exhibit similar behavior, converging to $\boldsymbol{x}^*$ (red vertical line) with less than 25 trials. When instead the objective function has a lot of local optima (bottom plot) as in Weierstrass, both EI and UCB get stuck after a few trials while FunBO keeps on exploring the search space eventually converging to $\boldsymbol{x}^*$. Notice how in this plot the convergence paths of all AFs differ and only the blue line aligns with the red line, i.e., converges to $\boldsymbol{x}^*$, after a few trials.

Using α_{FunBO} to optimize the nine functions in $\mathcal{F}$ leads to a fast and accurate convergence to the global optima (Fig. 4). The same is confirmed when extending the test set to include 50 scaled and translated instances of the functions in $\mathcal{F}$ (Fig. 11, right). Interestingly, the input spaces considered in this experiment vary significantly. This seems to suggest that scale does not affect the discovery of new AFs as long as the possible scale variability is accounted for in the training set. Further investigation is needed to assess FunBO robustness to more extreme scale differences, such as those often encountered in robot simulations or high-dimensional parameter spaces. Finally, Fig. 4 shows a surprisingly good performance of random search. This is due to the fact that random search performs competitively on functions with numerous local optima, which are generally harder to optimize. Aggregating performance across all functions in $\mathcal{F}$ highlights that *no single known general-purpose* AF *consistently outperforms the others*. This aligns with the well-established understanding that the effectiveness of AF can vary significantly across different types of black-box functions and is consistent with findings reported in the literature (Perrone et al., 2019; Li et al., 2018).

ID-Bench. Next we evaluate FunBO capabilities to generate AFs that perform well *within* function classes using Branin, Goldstein-Price and Hartmann ($d = 3$). For each of these three functions, we train both FunBO and MetaBO with $|\mathcal{G}| = 25$ instances of the original function obtained by scaling and translating it with values in $[0.9, 1.1]$ and $[-0.1, 0.1]^d$ respectively.[12] For FunBO we randomly assign 5 functions in $\mathcal{G}$ to $\mathcal{G}_{\text{V}}$ and keep the rest in $\mathcal{G}_{\text{Tr}}$. We test the performance of the learned AFs on another 100 instances of the same function, with randomly sampled values of scale and translation from the same ranges. We additionally compare against a BO algorithm that uses EI, UCB, PofI, MEAN or a random selection of points. All hyper-parameter settings for this experiment are provided in Appendix C.2. Across all objective functions, α_{FunBO} leads to a convergence performance that outperform general purpose AFs (Fig. 5). More importantly, despite using the same inputs of EI or UCB, FunBO is able reach performances that are comparable or superior to those of AFs that are parametrized by neural networks and use additional inputs (Fig. 5). In terms of interpretability, notice how the AF for Goldstein-Price (Fig. 14) can be written as $\sigma^2(\mathbf{x}|\mathcal{D}_t)\Phi(\frac{y^*-\mu(\mathbf{x}|\mathcal{D}_t)}{\sigma(\mathbf{x}|\mathcal{D}_t)})$ thus giving a modified PofI where the probability of observing an improvement over the incumbent is multiplied by the predictive variance.

The AFs found in this experiment (code in Figs. 13-15) are "customized" to a given function class thus being closer, in spirit, to the transfer AF. However, in order to further validate the generalizability of α_{FunBO} found in OOD-Bench, we tested such AF across instances of Branin, Goldstein-Price and Hartmann (Fig. 12, green line). We found it to perform well against general purpose AFs thus confirming the strong results observed in OOD-Bench while being, as expected, slower than AFs customized to a specific objective.

HPO-ID. We test FunBO on two HPO tasks where the goal is to minimize the loss ($d = 2$) of an RBF-based SVM and an AdaBoost algorithm.[13] As in ID-Bench, we test the ability to generate AFs that generalize well within function classes. Therefore, we train FunBO and MetaBO with losses computed on a random selection of 35 of the 50 available datasets and test on losses computed on the remaining 15 datasets. For FunBO we randomly assign 5 dataset to $\mathcal{G}_{\text{V}}$ and keep the rest in $\mathcal{G}_{\text{Tr}}$.

[12] Throughout the paper we adopt MetaBO's translation and scaling ranges.

[13] We use precomputed loss values across 50 datasets given as part of the HyLAP project (`http://www.hylap.org/`). For SVM, the two hyperparameters are the RBF kernel parameter and the penalty parameter while for AdaBoost they correspond to the number of product terms and the number of iterations.

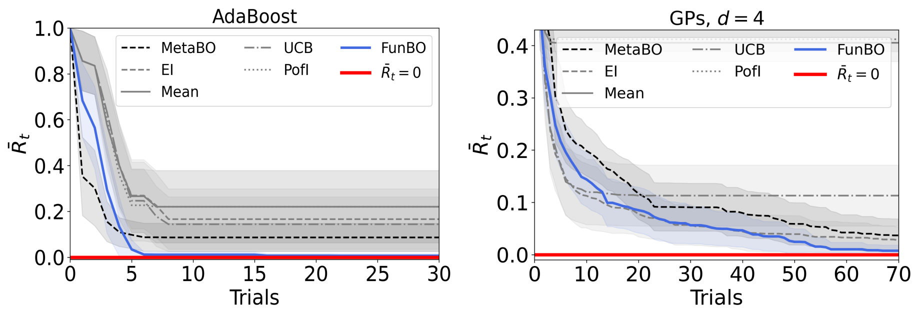

Figure 6: Average BO performance when using known general purpose AFs (gray lines), the AF learned by MetaBO (black dashed line) and α_{FunBO} (blue line). Shaded area gives $\pm$ standard deviations/2. The red line represents $\bar{R}_t = 0$, i.e. zero average regret. *Left*: HPO-ID. *Right*: GPs-ID with $d = 4$.

FunBO identifies AFs (code in Fig. 17-18) that outperform all other AFs in AdaBoost (Fig. 6, left) while performing similarly to general purpose or meta-learned AFs for SVM (Fig. 16). Across the two tasks, α_{FunBO} found in OOD-Bench still outperforms general-purpose AFs while yielding slightly worse performance compared to MetaBO and FunBO customized AFs (Fig. 16, green lines).

GPs-ID. Similar results are obtained for general function classes whose members do not exhibit any particular shared structure. We let $\mathcal{G}_{\text{Tr}}$ include 25 functions sampled from a GP prior with $d = 3$, RBF kernel and length-scale drawn uniformly from $[0.05, 0.5]$. We test the found AF on 100 other GP samples defined both for $d = 3$ and $d = 4$ and length-scale values sampled similarly. As done by Volpp et al. (2020), we consider a dimensionality-agnostic version of MetaBO that allows deploying the function learned from $d = 3$ functions on $d = 4$ objectives. We found α_{FunBO} to outperform all other AFs (code in Fig. 20) in $d = 4$ (Fig. 6, right) while matching EI and outperforming MetaBO in $d = 3$ (Fig. 19, left).

FEW-SHOT. We conclude our experimental analysis by demonstrating how FunBO can be used in the context of few-shot adaptation. In this setting, we aim at learning an AF customized to a specific function class by "adapting" an initial AF with a small number of instances from the target class. We consider Ackley ($d = 2$) as the objective function and compare against FSAF (Hsieh et al., 2021), which is the closest few-shot adaptation method for BO. FSAF trains the initial AF with a set of GPs, adapts it using 5 instances of scaled and translated Ackley functions, then tests the adapted AF on 100 additional Ackley instances, generated in the same manner. Note that FSAF uses a large variety of GP functions with different kernels and various hyperparameters for training the initial AF. On the contrary, FunBO few-shot adaptation is performed by setting the initial h function to the one found in GPs-ID (Fig. 7, green line) using 25 GPs with RBF kernel, and including the 5 instances of Ackley used by FSAF in $\mathcal{G}_{\text{Tr}}$. Despite the limited training set, FunBO adapts very quickly to the new function instances, identifying an AF (code in Fig. 21) that outperforms both general purpose AFs and FSAF (Fig. 7, blue line).

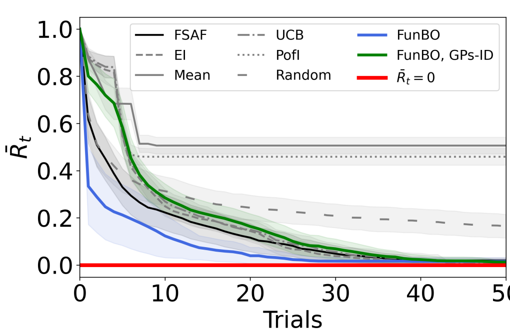

Figure 7: FEW-SHOT.

5 Related work

LLMs as mutation operators. FunBO expands FunSearch (Romera-Paredes et al., 2023), an evolutionary algorithm pairing an LLM with an evaluator to solve open problems in mathematics and algorithm design. Prior to FunSearch, the idea of using LLMs as mutation operators paired with a scoring mechanism had been explored to a create a self-improvement loop (Lehman et al., 2023), to optimize code for robotic simulations, or to evolve stable diffusion images with simple genetic algorithms (Meyerson et al., 2023). Other works explore the use of LLMs to search over neural network architectures described with Python code (Nasir et al., 2023; Zheng et al., 2023; Chen et al., 2024), find formal proofs for automatic theorem proving (Polu & Sutskever, 2020; Jiang et al., 2022) or automatically design heuristics (Liu et al., 2024a).

Meta-learning for BO. Our work is also related to the literature on meta-learning for BO. In this realm, several studies have focused on meta-learning an accurate surrogate model for the objective function exploiting observations from related functions, for instance by using standard multi-task GPs (Swersky et al., 2013; Yogatama & Mann, 2014) or ensembles of GP models (Feurer et al., 2018; Wistuba et al., 2018; Wistuba & Grabocka, 2021). Others have focused on meta-learning general purpose optimizers by using recurrent neural networks with access to gradient information (Chen et al., 2017) or transformers (Chen et al., 2022). Note that, while meta-learned surrogate models *explicitly* learn structure from past functions observing data-points for each of them, methods that meta-learn AFs via $\mathcal{G}$ *implicitly* learn similarities between these objectives by observing the optimization pattern that each previously sampled AF obtained for each objective function in $\mathcal{G}$. Interestingly, the most significant performance gains observed for the approach proposed by Chen et al. (2022) stem from using a standard AF (EI) on top of the transformer architecture for output predictions. This confirms the continued importance of AFs as crucial components in BO, even when combined with transformer-based approaches, and highlights the importance of a method such as FunBO that can be seamlessly integrated with these newer architectures, potentially leading to further improvements in performance. More relevant to our work are studies focusing on transferring information from related tasks by learning novel AFs that more efficiently solve the classic exploration-exploitation trade-off in BO algorithms (Volpp et al., 2020; Hsieh et al., 2021; Maraval et al., 2024). In contrast to prior works in this literature, FunBO produces AFs that are more interpretable, simpler and cheaper to deploy than neural network-based AFs and generalize not only within specific function classes but also across different classes.

LLMs and black-box optimization. Several works investigated the use of LLMs to solve black-box optimization problems. For instance, both Liu et al. (2024b) and Yang et al. (2024) framed optimization problems in natural language and asked LLMs to iteratively propose promising solutions and/or evaluate them. Similarly, Ramos et al. (2023) replaced surrogate modeling with LLMs within a BO algorithm targeted at catalyst or molecule optimization. Other works have focused on exploiting black-box methods for prompt optimization (Sun et al., 2022; Chen et al., 2023; Cheng et al., 2023; Fernando et al., 2023), solving HPO tasks with LLMs (Zhang et al., 2023) or identifying optimal LLM hyperparameter settings via black-box optimization approaches (Wang et al., 2023; Tribes et al., 2024). Concurrent to our work, Yao et al. (2024) propose to use an LLM coupled with an evolutionary procedure to find cost-aware AFs. Several works investigated the use of LLMs to solve black-box optimization problems. For instance, both Liu et al. (2024b) and Yang et al. (2024) framed optimization problems in natural language and asked LLMs to iteratively propose promising solutions and/or evaluate them. Similarly, Ramos et al. (2023) replaced surrogate modeling with LLMs within a BO algorithm targeted at catalyst or molecule optimization. Other works have focused on exploiting black-box methods for prompt optimization (Sun et al., 2022; Chen et al., 2023; Cheng et al., 2023; Fernando et al., 2023), solving HPO tasks with LLMs (Zhang et al., 2023) or identifying optimal LLM hyperparameter settings via black-box optimization approaches (Wang et al., 2023; Tribes et al., 2024). Concurrent to our work, Yao et al. (2024) propose to use an LLM coupled with an evolutionary procedure to find cost-aware AFs.

AFs representations Works proposing new meta-learned or general purpose AFs can also be classified based on the representation used for the AF. Differently from general-purpose AFs, for which an analytical representation is available, recent works have explored representing AFs via neural networks or code. Among the works using neural networks, Volpp et al. (2020) proposed a *neural* AF that is a MLP with relu-activations while Chen et al. (2022) and Maraval et al. (2024) jointly trained surrogate models and AFs via transformers or neural processes. Instead, the recent work by Yao et al. (2024) represents AFs for setting with limited experimentation budgets in code.

6 Conclusions and Discussion

We tackled the problem of discovering novel, well performing AFs for BO through FunBO, a FunSearch-based algorithm which explores the space of AFs by letting an LLM iteratively modify the AF expression in native computer code to improve the efficiency of the corresponding BO algorithm. We have shown across a variety of settings that FunBO learns AFs that generalize well within and across function classes while being easily adaptable to specific objective functions of interest with only a few training examples.

Limitations. FunBO inherits the strengths of FunSearch along with some of its inherent constraints. While FunSearch allows finding programs that are concise and interpretable, it works best for programs that can be quickly evaluated and for which the score provides an accurate quantification of the improvement achieved. Therefore, a potential limitation of FunBO is the computational overhead associated with running a full BO loop for each function in $\mathcal{G}$, which significantly increases the evaluation time of every sampled AF (especially when T is high). This limits the scalability of FunBO for larger sets $\mathcal{G}$ and hinders its application to more complex optimization problems, such as those with multiple objectives. In addition, the simple metric considered in this work in Eq. (1), only captures the distance from the true optimum and the number of trials needed to identify it. More research needs to be done to understand if a metric that better characterizes the convergence path for a given AF can improve FunBO performance. Furthermore, each FunBO experiment shown in this work required obtaining a large number of LLM samples. This means that the overall cost of experiments, which depends on the LLM used as well as the algorithm's implementation (e.g. single threaded or distributed, as originally proposed by FunSearch), can be high. Finally, as reported by Romera-Paredes et al. (2023), the variance in the quality of the AF found by FunBO is high. This is due to the randomness in both the LLM sampling and the evolutionary procedure. While we were able to reproduce the results shown for ID-Bench, HPO-ID and GPs-ID with different FunBO experiments, finding AFs that perform well across function classes required multiple FunBO runs.

Future work. This work opens up several promising avenues for future research. While our focus here was on the simplest single-output BO algorithm with a GP surrogate model, FunBO can be extended to learn new AFs for various adaptations of this problem, such as constrained optimization, noisy evaluations, or alternative surrogate. For instance, in order to deal with cases where the objective to be optimized requires very expensive/time-consuming evaluations, one could explore learning an AF by using a set $\mathcal{G}$ that includes cheaper and lower-fidelity evaluations of the objective. By accounting for the difference between low-fidelity and high-fidelity evaluations in the surrogate models, one can investigate whether FunBO can learn AFs that transfer to more expensive-to-evaluate objectives. We speculate that in these settings, a key challenge is to find a small but representative set of low-fidelity simulators that can be used to drive the LLM exploration by providing a meaningful signal for the optimisation process while keeping the cost limited. In addition, FunBO can be used to search in the space of functions with different inputs thus potentially discovering e.g. non myopic AFs. Our method is inherently flexible and can accommodate such extensions which we view as natural follow-up work. Additionally, FunBO demonstrates the potential to harness the power of LLMs while maintaining the interpretability of AFs expressed in code. This opens an exciting avenue for exploring how and what assumptions can be encoded within AFs, based on the desired program characteristics and prior knowledge about the objective function. Finally, the discovered AFs might have intrinsic value, independently on how they were discovered. Future work could focus on more extensively test their properties and identify those that can be added to the standard suite of AFs available in BO packages.

REFERENCES

Carolin Benjamins, Elena Raponi, Anja Jankovic, Carola Doerr, and Marius Lindauer. Self-adjusting weighted expected improvement for bayesian optimization. *arXiv preprint arXiv:2306.04262*, 2023.

James Bergstra, Rémi Bardenet, Yoshua Bengio, and Balázs Kégl. Algorithms for hyper-parameter optimization. In *Advances in Neural Information Processing Systems*, volume 24, 2011.

Roberto Calandra, André Seyfarth, Jan Peters, and Marc Peter Deisenroth. Bayesian optimization for learning gaits under uncertainty: An experimental comparison on a dynamic bipedal walker. *Annals of Mathematics and Artificial Intelligence*, 76:5–23, 2016.

Angelica Chen, David Dohan, and David So. Evoprompting: Language models for code-level neural architecture search. In *Advances in Neural Information Processing Systems*, volume 36, 2024.

Lichang Chen, Jiuhai Chen, Tom Goldstein, Heng Huang, and Tianyi Zhou. Instructzero: Efficient instruction optimization for black-box large language models. *arXiv preprint arXiv:2306.03082*, 2023.

Yutian Chen, Matthew W Hoffman, Sergio Gómez Colmenarejo, Misha Denil, Timothy P Lillicrap, Matt Botvinick, and Nando Freitas. Learning to learn without gradient descent by gradient descent. In *International Conference on Machine Learning*, pp. 748–756, 2017.

Yutian Chen, Xingyou Song, Chansoo Lee, Zi Wang, Richard Zhang, David Dohan, Kazuya Kawakami, Greg Kochanski, Arnaud Doucet, Marc' Aurelio Ranzato, Sagi Perel, and Nando de Freitas. Towards learning universal hyperparameter optimizers with transformers. In *Advances in Neural Information Processing Systems*, volume 35, pp. 32053–32068, 2022.

Jiale Cheng, Xiao Liu, Kehan Zheng, Pei Ke, Hongning Wang, Yuxiao Dong, Jie Tang, and Minlie Huang. Black-box prompt optimization: Aligning large language models without model training. *arXiv preprint arXiv:2311.04155*, 2023.

Hyunghun Cho, Yongjin Kim, Eunjung Lee, Daeyoung Choi, Yongjae Lee, and Wonjong Rhee. Basic enhancement strategies when using Bayesian optimization for hyperparameter tuning of deep neural networks. *IEEE Access*, 8:52588–52608, 2020.

E G Coffman, M R Garey, and D S Johnson. Approximation algorithms for bin-packing — an updated survey. In G Ausiello, M Lucertini, and P Serafini (eds.), *Algorithm Design for Computer System Design*, pp. 49–106. Springer Vienna, 1984.

Qin Ding, Yue Kang, Yi-Wei Liu, Thomas Chun Man Lee, Cho-Jui Hsieh, and James Sharpnack. Syndicated bandits: A framework for auto tuning hyper-parameters in contextual bandit algorithms. *Advances in Neural Information Processing Systems*, 35:1170–1181, 2022.

C. Fernando, D. Banarse, H. Michalewski, S. Osindero, and T. Rocktäschel. Promptbreeder: Self-referential self-improvement via prompt evolution. *arXiv preprint arXiv:2309.16797*, 2023.

Matthias Feurer, Benjamin Letham, and Eytan Bakshy. Scalable meta-learning for Bayesian optimization using ranking-weighted Gaussian process ensembles. In *AutoML Workshop at ICML*, volume 7, 2018.

Peter I Frazier, Warren B Powell, and Savas Dayanik. A knowledge-gradient policy for sequential information collection. *SIAM Journal on Control and Optimization*, 47(5):2410–2439, 2008.

Roman Garnett. *Bayesian Optimization*. Cambridge University Press, 2023.

Google-PaLM-2-Team. PaLM 2 Technical Report. *arXiv preprint arXiv:2305.10403*, 2023.

Matthew Hoffman, Eric Brochu, Nando De Freitas, et al. Portfolio allocation for bayesian optimization. In *UAI*, pp. 327–336, 2011.

Bing-Jing Hsieh, Ping-Chun Hsieh, and Xi Liu. Reinforced few-shot acquisition function learning for Bayesian optimization. In *Advances in Neural Information Processing Systems*, volume 34, pp. 7718–7731, 2021.

Albert Qiaochu Jiang, Wenda Li, Szymon Tworkowski, Konrad Czechowski, Tomasz Odrzygóźdź, Piotr Miłoś, Yuhuai Wu, and Mateja Jamnik. Thor: Wielding hammers to integrate language models and automated theorem provers. In *Advances in Neural Information Processing Systems*, volume 35, pp. 8360–8373, 2022.

Donald R Jones, Matthias Schonlau, and William J Welch. Efficient global optimization of expensive black-box functions. *Journal of Global Optimization*, 13:455–492, 1998.

Ksenia Korovina, Sailun Xu, Kirthevasan Kandasamy, Willie Neiswanger, Barnabas Poczos, Jeff Schneider, and Eric Xing. Chembo: Bayesian optimization of small organic molecules with synthesizable recommendations. In *International Conference on Artificial Intelligence and Statistics*, pp. 3393–3403, 2020.

Harold J Kushner. A new method of locating the maximum point of an arbitrary multipeak curve in the presence of noise. *Journal Basic Engineering*, 86(1):97–106, 1964.

Tze Leung Lai and Herbert Robbins. Asymptotically efficient adaptive allocation rules. *Advances in Applied Mathematics*, 6(1):4–22, 1985.

Joel Lehman, Jonathan Gordon, Shawn Jain, Kamal Ndousse, Cathy Yeh, and Kenneth O Stanley. Evolution through large models. In *Handbook of Evolutionary Machine Learning*, pp. 331–366. Springer, 2023.

Lisha Li, Kevin Jamieson, Giulia DeSalvo, Afshin Rostamizadeh, and Ameet Talwalkar. Hyperband: A novel bandit-based approach to hyperparameter optimization. *Journal of Machine Learning Research*, 18(185):1–52, 2018.

Fei Liu, Xialiang Tong, Mingxuan Yuan, Xi Lin, Fu Luo, Zhenkun Wang, Zhichao Lu, and Qingfu Zhang. Evolution of heuristics: Towards efficient automatic algorithm design using large language mode. *arXiv preprint arXiv:2401.02051*, 2024a.

Tennison Liu, Nicolás Astorga, Nabeel Seedat, and Mihaela van der Schaar. Large language models to enhance Bayesian optimization. In *International Conference on Learning Representations*, 2024b.

Alexandre Maraval, Matthieu Zimmer, Antoine Grosnit, and Haitham Bou Ammar. End-to-end meta-Bayesian optimisation with transformer neural processes. In *Advances in Neural Information Processing Systems*, volume 36, 2024.

Elliot Meyerson, Mark J Nelson, Herbie Bradley, Adam Gaier, Arash Moradi, Amy K Hoover, and Joel Lehman. Language model crossover: Variation through few-shot prompting. *arXiv preprint arXiv:2302.12170*, 2023.

Jonas Mockus. On Bayesian methods for seeking the extremum. *Proceedings of the IFIP Technical Conference*, pp. 400–404, 1974.

Muhammad U Nasir, Sam Earle, Julian Togelius, Steven James, and Christopher Cleghorn. LLMatic: Neural architecture search via large language models and quality diversity optimization. *arXiv preprint arXiv:2306.01102*, 2023.

Valerio Perrone, Huibin Shen, Matthias W Seeger, Cedric Archambeau, and Rodolphe Jenatton. Learning search spaces for bayesian optimization: Another view of hyperparameter transfer learning. *Advances in neural information processing systems*, 32, 2019.

Stanislas Polu and Ilya Sutskever. Generative language modeling for automated theorem proving. *arXiv preprint arXiv:2009.03393*, 2020.

Mayk Caldas Ramos, Shane S Michtavy, Marc D Porosoff, and Andrew D White. Bayesian optimization of catalysts with in-context learning. *arXiv preprint arXiv:2304.05341*, 2023.

Carl Edward Rasmussen and Christopher KI Williams. *Gaussian Processes for Machine Learning*. MIT Press Cambridge, MA, 2006.

Bernardino Romera-Paredes, Mohammadamin Barekatain, Alexander Novikov, Matej Balog, M Pawan Kumar, Emilien Dupont, Francisco JR Ruiz, Jordan S Ellenberg, Pengming Wang, Omar Fawzi, et al. Mathematical discoveries from program search with large language models. *Nature*, pp. 1–3, 2023.

John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *CoRR*, 2017.

Jasper Snoek, Hugo Larochelle, and Ryan P Adams. Practical Bayesian optimization of machine learning algorithms. In *Advances in Neural Information Processing Systems*, volume 25, 2012.

I. M. Sobol'. On the distribution of points in a cube and the approximate evaluation of integrals. *Zhurnal Vychislitel'noi Matematiki i Matematicheskoi Fiziki*, 7, 1967.

Tianxiang Sun, Yunfan Shao, Hong Qian, Xuanjing Huang, and Xipeng Qiu. Black-box tuning for language-model-as-a-service. In *International Conference on Machine Learning*, pp. 20841–20855, 2022.

Kevin Swersky, Jasper Snoek, and Ryan P Adams. Multi-task Bayesian optimization. In *Advances in Neural Information Processing Systems*, volume 26, 2013.

Reiko Tanese. *Distributed Genetic Algorithms for Function Optimization*. PhD thesis, University of Michigan, 1989.

Christophe Tribes, Sacha Benarroch-Lelong, Peng Lu, and Ivan Kobyzev. Hyperparameter optimization for large language model instruction-tuning. In *AAAI Conference on Artificial Intelligence*, 2024.

Google Cloud Vertex AI. Code models overview. 2023. URL https://cloud.google.com/vertex-ai/docs/generative-ai/code/code-models-overview.

Michael Volpp, Lukas P Fröhlich, Kirsten Fischer, Andreas Doerr, Stefan Falkner, Frank Hutter, and Christian Daniel. Meta-learning acquisition functions for transfer learning in Bayesian optimization. In *International Conference on Learning Representations*, 2020.

Chi Wang, Xueqing Liu, and Ahmed Hassan Awadallah. Cost-effective hyperparameter optimization for large language model generation inference. In *International Conference on Automated Machine Learning*, pp. 21–1, 2023.

Zi Wang and Stefanie Jegelka. Max-value entropy search for efficient Bayesian optimization. In *International Conference on Machine Learning*, pp. 3627–3635, 2017.

Martin Wistuba and Josif Grabocka. Few-shot Bayesian optimization with deep kernel surrogates. In *International Conference on Learning Representations*, 2021.

Martin Wistuba, Nicolas Schilling, and Lars Schmidt-Thieme. Scalable Gaussian process-based transfer surrogates for hyperparameter optimization. *Machine Learning*, 107(1):43–78, 2018.

Chengrun Yang, Xuezhi Wang, Yifeng Lu, Hanxiao Liu, Quoc V Le, Denny Zhou, and Xinyun Chen. Large language models as optimizers. In *International Conference on Learning Representations*, 2024.

Yiming Yao, Fei Liu, Ji Cheng, and Qingfu Zhang. Evolve cost-aware acquisition functions using large language models. *arXiv preprint arXiv:2404.16906*, 2024.

Dani Yogatama and Gideon Mann. Efficient transfer learning method for automatic hyperparameter tuning. In *International Conference on Artificial Intelligence and Statistics*, pp. 1077–1085, 2014.

Michael R Zhang, Nishkrit Desai, Juhan Bae, Jonathan Lorraine, and Jimmy Ba. Using large language models for hyperparameter optimization. In *NeurIPS 2023 Foundation Models for Decision Making Workshop*, 2023.

Mingkai Zheng, Xiu Su, Shan You, Fei Wang, Chen Qian, Chang Xu, and Samuel Albanie. Can GPT-4 perform neural architecture search? *arXiv preprint arXiv:2304.10970*, 2023.

```python
import numpy as np
from scipy import stats

def acquisition_function(predictive_mean, predictive_var, incumbent, beta=1.0):
  """Returns the index of the point to collect in a vector of eval points.

  Given the posterior mean and posterior variance of a GP model for the objective function,
  this function computes an heuristic and find its optimum. The next function evaluation
  will be placed at the point corresponding to the selected index in a vector of
  possible eval points.

  Args:
    predictive_mean: an array of shape [num_points, dim] containing the predicted mean
        values for the GP model on the objective function for ‘num_points‘ points of
        dimensionality ‘dim‘.
    predictive_var: an array of shape [num_points, dim] containing the predicted variance
        values for the GP model on the objective function for ‘num_points‘ points
        of dimensionality ‘dim‘.
    incumbent: current optimum value of objective function observed.
    beta: a possible hyperparameter to construct the heuristic.

  Returns:
    An integer representing the index of the point in the array of shape [num_points, dim]
    that needs to  be selected for function evaluation.
  """
  z = (incumbent - predictive_mean) / np.sqrt(predictive_var)
  predictive_std = np.sqrt(predictive_var)
  vals = (incumbent - predictive_mean) * stats.norm.cdf(z) + predictive_std * stats.norm.pdf(z)
  return np.argmax(vals)
```

Figure 8: Python code for FunBO initial h function with full docstring.

A Code for FunBO Components

Fig. 8 gives the Python code for the initial acquisition function used by FunBO, including the full docstring. The docstring describes the inputs of the function and the way in which the function itself is used within the evaluate function e. Evaluation of the functions generated by FunBO is done by first running a full BO loop (see Fig. 9 for Python code) and then, based on its output (the initial optimal input value, the true optimum, the found optimum and the percentage of steps taken before finding the latter), computing the score as in the Python code of Fig. 10. Note how the latter captures how accurately and quickly a BO algorithm using the proposed AF finds the true optimum.

B Programs Database

The DB structure matches the one proposed by FunSearch (Romera-Paredes et al., 2023). We discuss it here for completeness. A multiple-deme model (Tanese, 1989) is employed to preserve and encourage diversity in the generated programs. Specifically, the program population in DB is divided into N_{DB} islands, each initialized with the given initial h and evolved independently. Within each island, programs are clustered based on their scores on the functions in $\mathcal{G}_{\text{Tr}}$, with AFs having the same scores grouped together. Sampling from DB involves first uniformly selecting an island and then sampling two AFs from it. Within the chosen island, a cluster is sampled, favoring those with higher score values, followed by sampling a program within that cluster, favoring shorter ones. The newly generated AF is added to the same island associated with the instances in the prompt, but to a cluster reflecting its scores on $\mathcal{G}_{\text{Tr}}$. Every 4 hours, all programs from the $N_{\text{DB}}/2$ islands with the lowest-scoring best AF are discarded. These islands are then reseeded with a single program from

```python
"""Evaluate an AF with a full BO loop for the objective f."""

import GPy
import numpy as np
import utils

def run_bo(
    f,    # objective function to minimize
    acquisition_function, # h given by LLM
    num_eval_points = 1000,
    num_trials = 30):
  """Run a BO loop and return the minimum objective functions found and the percentage of
  trials required to reach it."""

  # Get evaluation points for AF. get_eval_points() returns a given number of points on a
  # Sobol grid on the f's input space
  eval_points = utils.get_eval_points(f, num_eval_points)

  # Get the initial point with get_initial_design(). This is set to be the point giving the
  # maximum (worst) function evaluation among eval_points
  initial_x, initial_y = utils.get_initial_design(f)

  # Initialize GP hyper-parameters. We pre-compute the RBF kernel hyper-parameters
  # for each given f. These are returned by get_hyperparameters().
  hp = utils.get_hyperparameters(f)

  # Initialize kernel and model.
  kernel = GPy.kern.RBF(input_dim=input_dim, variance=hp['variance'],
  lengthscale=hp['lengthscale'], ARD=hp['ard'])
  model = GPy.models.GPRegression(initial_x, initial_y, kernel)

  # Get initial predictive mean and var.
  predictive_mean, predictive_var = model.predict(eval_points)

  # Get initial optimum value.
  found_min = initial_min_y = float(np.min(model.Y))

  # Get true optimum value.
  true_min = np.min(f(eval_points))

  # Optimization loop.
  for _ in range(num_trials):
    new_input = acquisition_function(eval_points,  # Get new point using AF.
        predictive_mean, predictive_var, found_min)
    new_output = f(new_input)  # Evaluate new point.
    model.set_XY(np.concatenate((model.X, new_input), axis=0), # Append to dataset.
                 np.concatenate((model.Y, new_output), axis=0))
    # Get updated mean and var
    predictive_mean, predictive_var = model.predict(eval_points)
    found_min = float(np.min(model.Y))  # Get current optimum value.

  # Get percentage of trials (note that we remove the number of given points in the
  initial design) needed to identify the optimum.
  percentage_steps_before_converging = (np.argmin(model.Y) - len(
    initial_design_inputs)) / (num_trials) if found_min == true_min else 1.0
  return (found_min, true_min, initial_min_y, percentage_steps_before_converging)
```

Figure 9: Python code for the first part of e used in FunBO. This function runs a full BO loop with a given number of trials and points on a Sobol grid to assess how efficiently a given AF allows optimizing f.

```python
"""Score an AF given the output of run_bo()."""

import numpy as np

def score(found_min, true_min, initial_min_y, percentage_steps_before_converging):
  """Compute a score based on the output of run_bo()."""

    # Get score based on how close the found optimum is to the true one (first term
    # in Eq. (1)).
    score_min_reached = 1.0 - np.abs(found_min - true_min) / (initial_min_y - true_min)

    # Get score based on how the percentage of trials needed to identify the true
    # optimum (second term in Eq. (1)).
    score_steps_needed = 1.0 - percentage_steps_needed

  return score_min_reached + score_steps_needed
```

Figure 10: Python code for the second part of e used in FunBO. Based on the output of run_bo(), this function computes a score capturing how accurately and quickly an AF allows identifying the true optimum.

the surviving islands. This procedure eliminates under-performing AFs, creating space for more promising programs. See the Methods section in Romera-Paredes et al. (2023) for further details.

C EXPERIMENTAL DETAILS

In this section, we provide the experimental details for all our experiments. We run FunBO with $\mathcal{T} = 48$hrs, $B = 12$ and $N_{\text{DB}} = 10$. To isolate the effect of using different AFs and eliminate confounding factors related to AF maximization or surrogate models, we maximized all AFs on a fixed Sobol grid (of size N_{SG}) over each function's input space. We also ensure the same initial design across all methods (including the point with the highest/worst function value on the Sobol grid) and used consistent GP hyperparameters which are tuned offline and fixed. In particular, we use a GP model with zero mean function and RBF kernel defined as $K_\theta(\boldsymbol{X}, \boldsymbol{X}') = \sigma_f^2 \text{exp}(-||\boldsymbol{X} - \boldsymbol{X}'||^2/2\ell^2)$ with $\theta = (\ell, \sigma_f^2)$ where ℓ and σ_f^2 are the length-scale and kernel variance respectively. The Gaussian likelihood noise σ^2 is set to $1\text{e}-5$ unless otherwise stated. We set $T = 30$ for all experiments apart for HPO-ID and GPs-ID for which we use $T = 20$ to ensure faster evaluations of generated AFs. We used the MetaBO implementation provided by the authors at `https://github.com/boschresearch/MetaBO`, retaining default parameters except for removing the local maximization of AFs and ensuring consistency in the initial design. We followed the same procedure for FSAF, using code available at `https://github.com/pinghsieh/FSAF`. We ran UCB with $\beta = 1$. Experiment-specific settings are detailed below.

C.1 OOD-BENCH

The parameter configurations adopted for each objective function used in this experiment, either in $\mathcal{G}$ or in $\mathcal{F}$, are given in Table 1. Notice that for Hartmann with $d = 3$ we use an ARD kernel. Scaled and translated functions are obtained with translations sampled uniformly in $[-0.1, 0.1]^d$ and scalings sampled uniformly in $[0.9, 1.1]$. Fig. 11 gives the results achieved by α_{FunBO} (blue line) and a dimensionality agnostic version of MetaBO that does not take the possible evaluation points as input of the neural AF. This allows the neural AF to be trained on one-dimensional functions and be used to optimize functions across input dimensions.

Table 1: Parameters used for OOD-Bench.

	d	$\mathcal{X}$	N_{SG}	ℓ	σ_f^2	σ_f^2
Ackley	1	$[-4, 4]$	1000	0.21	28.19	$1e-5$
Levy	1	$[-10, 10]$	1000	1.05	83.32	$1e-5$
Schwefel	1	$[-500, 500]$	1000	18.46	76868.65	$1e-5$
Rosenbrock	1	$[-5, 10]$	1000	1.20	87328.20	$1e-5$
Sphere	1	$[-5, 5]$	1000	18.46	924202.43	$1e-5$
Styblinski-Tang	1	$[-5, 5]$	1000	7.34	119522207.86	$1e-5$
Weierstrass	1	$[-0.5, 0.5]$	1000	0.01	0.39	$1e-5$
Beale	2	$[-4, 5]^2$	10000	0.46	546837.32	$1e-5$
Branin	2	$[-5, 10] \times [0, 15]$	10000	4.65	155233.52	$1e-5$
Michalewicz	2	$[0, \pi]^2$	10000	0.22	0.10	$1e-5$
Goldstein-Price	2	$[-2, 2]^2$	10000	0.27	117903.96	$1e-5$
Hartmann-3	3	$[0, 1]^3$	1728	$[0.716, 0.298, 0.186]$	0.83	$1.688e-11$
Hartmann-6	6	$[0, 1]^6$	729	1.0	1.0	$1e-5$

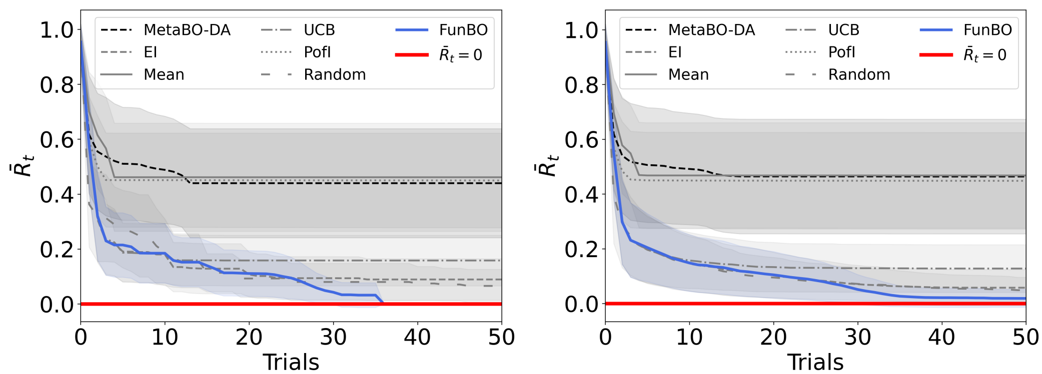

Figure 11: OOD-Bench. Average BO performance when using known general purpose AFs (gray lines with different patterns), the AF learned by a dimensionality agnostic version of MetaBO (MetaBO-DA, black dashed line) and α_{FunBO} (blue line). Shaded area gives $\pm$ standard deviations$/2$. The red line represents $\bar{R}_t = 0$, i.e., zero average regret. *Left*: $\mathcal{F}$ includes nine different synthetic functions. *Right*: Extended test set including, for each function in $\mathcal{F}$, 50 randomly scaled and translated instances.

C.2 ID-BENCH

The parameter configurations for Branin, Goldstein-Price and Hartmann are given in Table 2. For this experiment, we adopt the parameters used by Volpp et al. (2020) thus optimize the functions in the unit-hypercube and use ARD RBF kernels. Fig. 12 gives the results achieved by α_{FunBO} (blue line) and the AF found by FunBO for OOD-Bench (green). The Python code for the found AFs is given in Figs. 13-15.

Table 2: Parameters used for ID-Bench.

	d	$\mathcal{X}$	N_{SG}	ℓ	σ_f^2	σ_f^2
Branin	2	$[0, 1]^2$	961	$[0.235, 0.578]$	2.0	$8.9e-16$
Goldstein-Price	2	$[0, 1]^2$	961	$[0.130, 0.07]$	0.616	$1e-6$
Hartmann-3	3	$[0, 1]^3$	1728	$[0.716, 0.298, 0.186]$	0.83	$1.688e-11$

C.3 HPO-ID

For this experiment, we adopt the GP hyperparameters used by Volpp et al. (2020). From the training datasets used in MetaBO, we assign "bands", "wine", "coil2000", "winequality-red" and "titanic" for

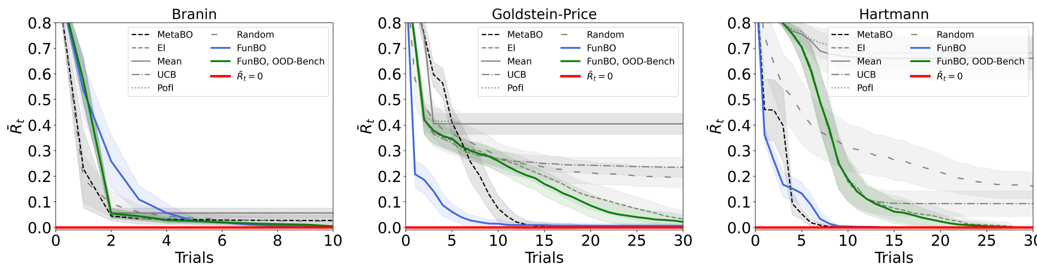

Figure 12: ID-Bench. Average BO performance when using known general purpose AFs (gray lines with different patterns), α_{FunBO} found in OOD-Bench (green line), the AF learned by MetaBO (black dashed line) and α_{FunBO} (blue line) on 100 instances of Branin, Goldstein-Price and Hartmann. Shaded area gives $\pm$ standard deviations$/2$. The red line represents $\bar{R}_t = 0$, i.e., zero average regret.

```python
import numpy as np
from scipy import stats

def acquisition_function(predictive_mean, predictive_var, incumbent, beta=1.0):
  """Returns the index of the point to collect ..."""
  y_pred = predictive_mean + 2 * predictive_var
  diff_current_best_y_pred = incumbent - y_pred
  bound_standard_deviation = np.maximum(np.sqrt(predictive_var), 1e-15)
  z = diff_current_best_y_pred / bound_standard_deviation
  vals = (diff_current_best_y_pred * stats.norm.cdf(z)
          + np.sqrt(predictive_var) * stats.norm.cdf(z + 0.5)
          + (stats.norm.cdf(z) - stats.norm.cdf(z + 0.5)) * predictive_var / 2)
  a = np.maximum(diff_current_best_y_pred, incumbent)
  alpha = diff_current_best_y_pred if incumbent > 0.0 else -np.inf
  alpha = np.maximum(alpha, 0.) * (-alpha + 0.5 * a) - y_pred
  y_vals = np.absolute(alpha + a + np.abs(y_pred)) * (a >= 0.)
  for y_val in y_vals:
    idx = np.argmax(vals - (y_val - y_pred) / bound_standard_deviation)
    vals[idx] = 0
  return np.argmax(vals)
```

Figure 13: ID-Bench. Python code for α_{FunBO} for Branin. The BO performance corresponding to this AF is given in Fig. 5 (left).

```python
import numpy as np
from scipy import stats

def acquisition_function(predictive_mean, predictive_var, incumbent, beta=1.0):
  """Returns the index of the point to collect ..."""
  shape, dim = predictive_mean.shape
  best_score = 0.0
  g_i = 0.0

  predictive_var[(shape-10)//2] *= dim
  predictive_var[~ np.isfinite(predictive_var)] = 1.0

  for i in range(predictive_mean.shape[0]):
    curr_z = (incumbent - predictive_mean[i]) / np.sqrt(predictive_var[i])
    new_score = predictive_var[i] *  stats.norm.cdf(curr_z, 0.5)
    if new_score > best_score:
      best_score = new_score
      g_i = i
  return g_i
```

Figure 14: ID-Bench. Python code for α_{FunBO} for Goldstein-Price. The BO performance corresponding to this AF is given in Fig. 5 (middle).

```python
import numpy as np
from scipy import stats

def acquisition_function(predictive_mean, predictive_var, incumbent, beta=1.0):
  """Returns the index of the point to collect ..."""
  diff_current_best_mean = incumbent - predictive_mean
  standard_deviation = np.sqrt(predictive_var)
  z = diff_current_best_mean / standard_deviation
  vals = diff_current_best_mean * stats.norm.cdf(z)**3 + (
      stats.norm.cdf(z)**2 + stats.norm.cdf(z) + 1) * stats.norm.pdf(z)
  index = np.argmax(stats.truncnorm.cdf(vals, a=-0.1, b=0.1))
  return index
```

Figure 15: ID-Bench. Python code for α_{FunBO} for Hartmann. The BO performance corresponding to this AF is given in Fig. 5 (right).

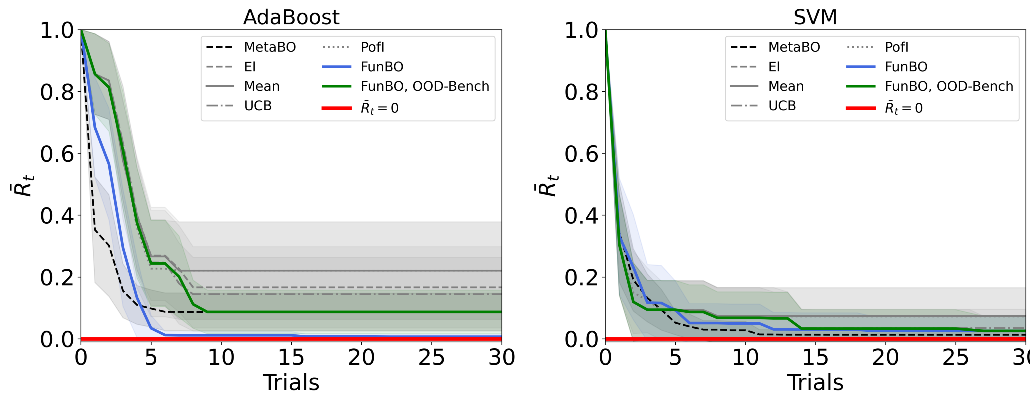

Figure 16: HPO-ID. Average BO performance when using known general purpose AFs (gray lines with different patterns), a meta-learned AF by MetaBO (black dashed line), α_{FunBO} found in OOD-Bench (green lines) and α_{FunBO} (blue lines). Shaded area gives $\pm$ standard deviations/2. The red line represents $\bar{R}_t = 0$, i.e., zero average regret.

```
import numpy as np
from scipy import stats

def acquisition_function(predictive_mean, predictive_var, incumbent, beta=1.0):
  """Returns the index of the point to collect ..."""
  c1 = np.exp(-beta)
  c2 = 2.0 * beta * np.exp(-beta)
  alpha = np.sqrt(2.0) * beta * np.sqrt(predictive_var)
  z = (incumbent - predictive_mean) / alpha
  vals = -abs(c1 * np.exp( - np.power(z, 2)) - 1.0 + c1 + incumbent
    ) + 2.0 * beta * np.power(z+c2, 2)
  vals -= np.log(np.power(alpha, 2))
  vals[np.argmin(vals)] = 1.0
  return np.argmin(vals)
```

Figure 17: HPO-ID. Python code for α_{FunBO} for AdaBoost. The BO performance corresponding to this AF is given in Fig. 6 (left).

Adaboost, and “bands”, “breast-cancer”, “banana”, “yeast” and “vehicle’ for SVM to $\mathcal{G}_{\text{V}}$. We keep the rest in $\mathcal{G}_{\text{Tr}}$. Fig. 16 gives the results achieved by α_{FunBO} (blue lines) and the AF found by FunBO for OOD-Bench (green lines). The Python code for the found AFs is given in Figs. 17-18.

C.4 GPS-ID

The functions included in both $\mathcal{G}$ and $\mathcal{F}$ are sampled from a GP prior with RBF kernel and length-scale values drawn uniformly from $[0.05, 0.5]$. The functions are optimized in the input space $[0, 1]^3$ with $N_{\text{SG}} = 1728$ points. In terms of GP hyperparameters, we set $\sigma_f^2 = 1.0$, $\sigma^2 = 1\text{e}-20$ and use the length-scale value used to sample each function as ℓ. Fig. 19 gives the results achieved by α_{FunBO} and the AF found by FunBO for OOD-Bench. The Python code for α_{FunBO} is given in Fig. 20.

C.5 FEW-SHOT

For this experiment, the 5 Ackley functions used to “adapt” the initial AF are obtained by scaling and translating the output and inputs values with translations and scalings uniformly sampled in $[-0.1, 0.1]^d$ and $[0.9, 1.1]$ respectively. The test set includes 100 instances of Ackley similarly obtained with scale and translations values in $[0.7, 1.3]$ and $[-0.3, 0.3]^d$ respectively. Furthermore, we consider $[0, 1]^2$ as input space and use $N_{\text{SB}} = 1000$. The GP hyperparameters are set to $\ell =$

```python
import numpy as np
from scipy import stats

def acquisition_function(predictive_mean, predictive_var, incumbent, beta=1.0):
  """Returns the index of the point to collect ..."""
  z = (incumbent - predictive_mean) / np.sqrt(predictive_var)
  vals = (incumbent - predictive_mean) * stats.norm.cdf(z
    ) + np.sqrt(predictive_var) * stats.norm.pdf(z)
  t0_val = stats.norm(loc=incumbent, scale=np.sqrt(predictive_var)).pdf(incumbent)
  t1_val = z * stats.norm.pdf(z)
  vals = ((vals * t1_val - t0_val) / (1 - 2 * t1_val)
          + t1_val*(vals/(1-2*t1_val))
          - vals/(1 - 2*t1_val)**2 + t1_val*(t1_val - z)/beta)
  return np.argmax(vals)
```

Figure 18: HPO-ID. Python code for α_{FunBO} for SVM. The BO performance corresponding to this AF is given in Fig. 16 (right).

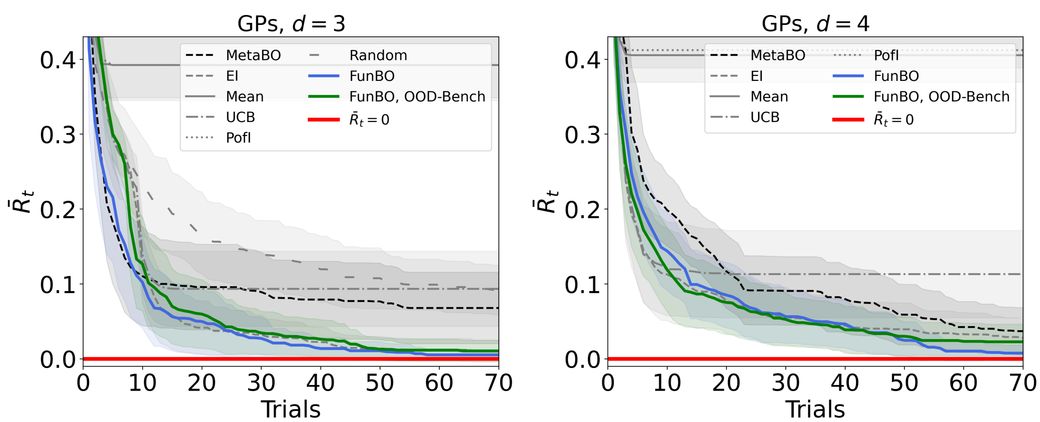

Figure 19: Average BO performance when using known general purpose AFs (gray lines with different patterns), the AF learned by MetaBO (black dashed line), α_{FunBO} found in OOD-Bench (green lines) and α_{FunBO} (blue lines). Shaded area gives $\pm$ standard deviations/2. The red line represents $\bar{R}_t = 0$, i.e. zero average regret. *Left*: GPs-ID. $\mathcal{F}$ includes functions with $d = 3$. *Right*: $\mathcal{F}$ includes functions with $d = 4$.

```python
import numpy as np
from scipy import stats

def acquisition_function(predictive_mean, predictive_var, incumbent, beta = 1.0):
  """Returns the index of the point to collect ..."""
  z = (incumbent - predictive_mean) / np.sqrt(predictive_var)
  vals = ((incumbent - predictive_mean) * stats.norm.cdf(z
    ) + np.sqrt(predictive_var) * stats.norm.pdf(z))**2
  vals = vals / (1 + (z / beta)**2 * np.sqrt(predictive_var))**2
  return np.argmax(vals)
```

Figure 20: GPs-ID. Python code for α_{FunBO}. The BO performance corresponding to this AF is given in Fig. 6 (right).

```python
import numpy as np
from scipy import stats

def acquisition_function(predictive_mean, predictive_var, incumbent, beta=1.0):
  """Returns the index of the point to collect ..."""
    num_points, _ = predictive_mean.shape
  a = 10
  z = (predictive_mean + 0.000001 - incumbent) / np.sqrt(predictive_var)
  vals = 1 / ((1 + (z / beta)**2 * np.sqrt(a * predictive_var + 0.00001)) **2)
  beta_sqrt_p_z = np.sqrt(beta) * z
  vals *= (1 + (z / beta)**2)*predictive_var/(
      (1+ (beta_sqrt_p_z / np.sqrt(predictive_var))**2 * predictive_var) * (
          1+(beta_sqrt_p_z / np.sqrt(predictive_var))**2))
  vals += (1 - beta_sqrt_p_z / np.sqrt(predictive_var))**2 * predictive_var/ (
      1 + (beta_sqrt_p_z / np.sqrt(predictive_var))**2 * predictive_var)**2
  vals = (1 + (z / beta)**2) * vals- (1 - (z / beta)**2) * np.exp(- 1) ** 2
  vals = np.sqrt(a * predictive_var) * vals / np.sqrt(
      a * predictive_var + 0.00001)
  vals *= np.sqrt(np.sqrt(a * predictive_var) * predictive_var)
  vals *= predictive_var**2
  vals[:num_points // 2] = 0
  return np.argmax(vals)
```

Figure 21: FEW-SHOT. Python code for α_{FunBO}. The BO performance corresponding to this AF is given in Fig. 7.

$[0.07, 0.018]$ (ARD kernel), $\sigma_f^2 = 1.0$ and $\sigma^2 = 8.9\text{e} - 16$. Python code for α_{FunBO} is given in Fig. 21.

C.6 ICLR REBUTTAL

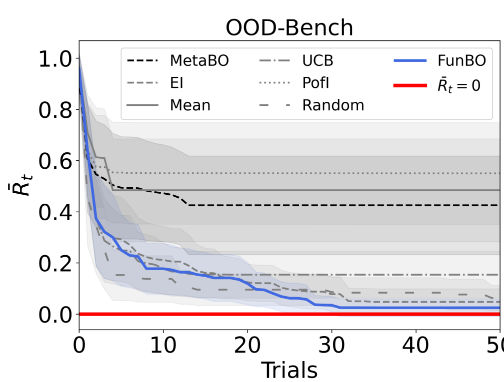

Figure 22: OOD-Bench. Average performance with increased Sobol grid resolution.

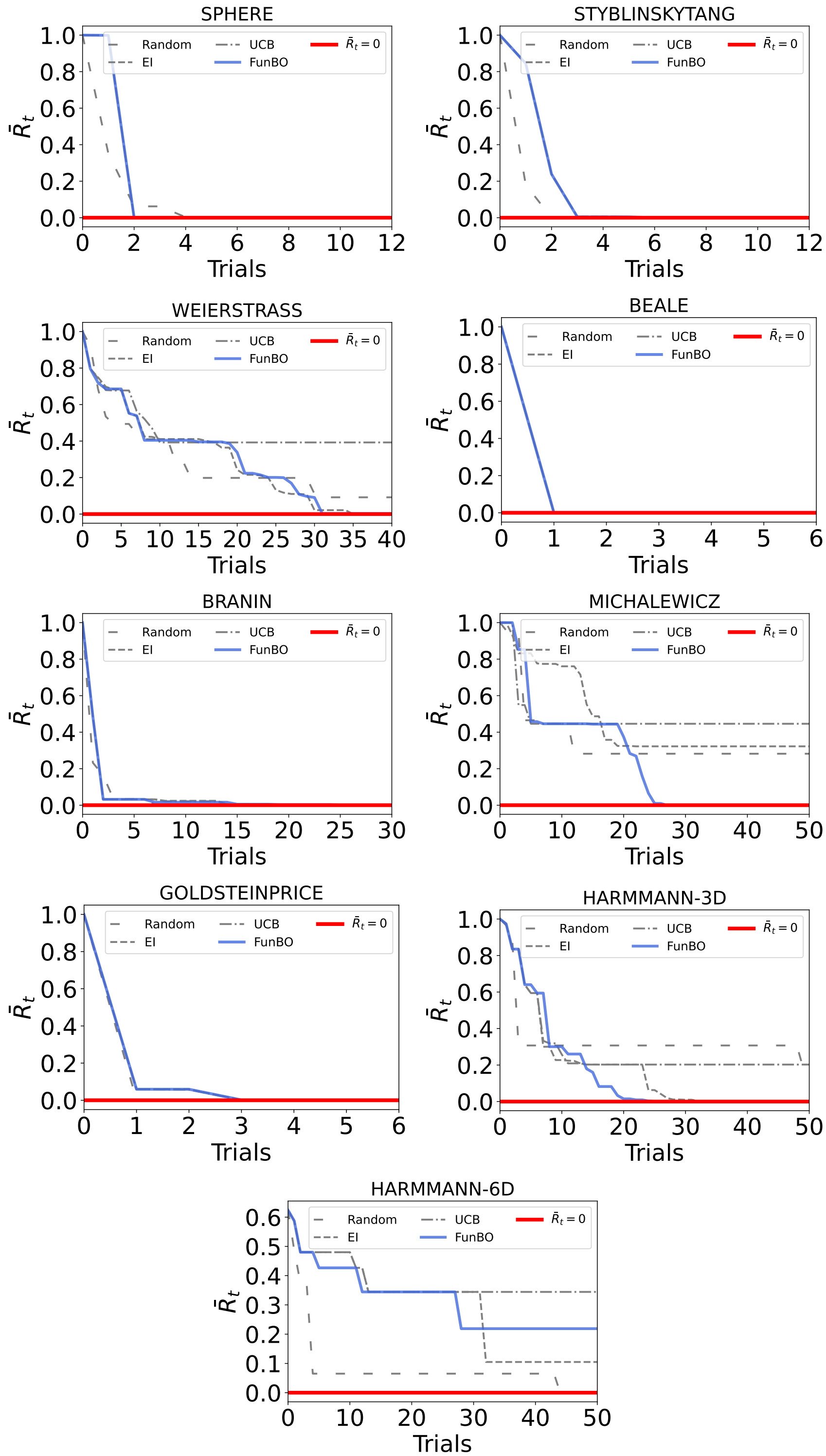

Figure 23: OOD-Bench. Performance on the single functions included in the test set using an increased Sobol grid resolution.

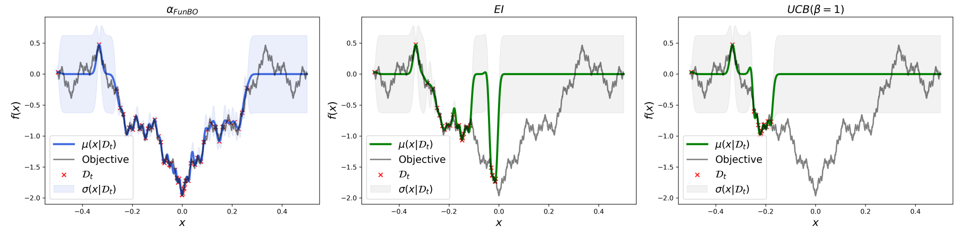

Figure 24: OOD-Bench. GP surrogate models for the Weierstrass function.

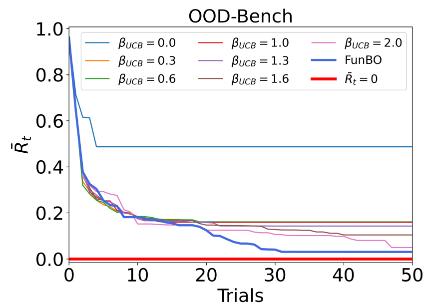

Figure 25: OOD-Bench. Performance of α_{FunBO} and UCB with different β values.