Automated Benchmark-Driven Design and Explanation of Hyperparameter Optimizers

Julia Moosbauer*, Martin Binder*,
Lennart Schneider, Florian Pfisterer, Marc Becker, Michel Lang, Lars Kotthoff, Bernd Bischl

***Abstract*—Automated hyperparameter optimization (HPO) has gained great popularity and is an important component of most automated machine learning frameworks.**

However, the process of designing HPO algorithms is still an unsystematic and manual process: New algorithms are often built on top of prior work, where limitations are identified and improvements are proposed. Even though this approach is guided by expert knowledge, it is still somewhat arbitrary. The process rarely allows for gaining a holistic understanding of which algorithmic components drive performance and carries the risk of overlooking good algorithmic design choices.

We present a principled approach to automated benchmark-driven algorithm design applied to multi-fidelity HPO (MF-HPO). First, we formalize a rich space of MF-HPO candidates that includes, but is not limited to, common existing HPO algorithms and then present a configurable framework covering this space. To find the best candidate automatically and systematically, we follow a programming-by-optimization approach and search over the space of algorithm candidates via Bayesian optimization. We challenge whether the found design choices are necessary or could be replaced by more naive and simpler ones by performing an ablation analysis. We observe that using a relatively simple configuration (in some ways, simpler than established methods) performs very well as long as some critical configuration parameters are set to the right value.

***Index Terms*—Algorithm design, algorithm analysis, hyperparameter optimization, multifidelity, automated machine learning**

I. Introduction

Machine learning (ML) is, in many regards, an optimization problem, and many ML methods can be expressed as algorithms that perform loss minimization with respect to a given objective function. The higher-level task of selecting the ML method and its configuration is often framed as an optimization problem as well, sometimes referred to as a *hyperparameter optimization* (HPO) [1] or *combined algorithm selection and hyperparameter optimization* (CASH) problem [2]. Successfully addressing this problem can lead to large performance gains compared to simply using defaults, and in the context of automated machine learning (AutoML), the use of HPO can make ML more accessible to non-experts. Because of their potential benefits to ML performance and usability, it is of particular interest to design optimization algorithms that perform particularly well on the HPO problem.

Optimization problems arise in many fields of science and engineering, but as the no-free-lunch theorem states, there is no one optimization algorithm that solves all problems equally well [3]. To design suitable optimizers, it is therefore important to understand the characteristics of HPO:

- **Black-box**: The objective usually provides no analytical information [4] – such as a gradient. Thus, the application of many traditional optimization methods – such as BFGS – is rendered inappropriate or at least questionable.
- **Complex search space**: The search space of the optimization problem is often high-dimensional and may contain continuous, integer-valued and categorical dimensions. Often, there are dependencies between dimensions or even specific hyperparameter values [5].
- **Expensive**: A single evaluation of the objective function may take hours or days. Thus, the total number of possible function evaluations is often severely limited [4].
- **Low-fidelity approximations possible**: An approximation of the true objective value at lower expense can often be obtained, for example, through a partial evaluation [6].
- **Low effective dimensionality**: The landscape of the objective function can usually be approximated well by a function of a small subset of all dimensions [7].

Recent HPO and AutoML research has focused on finding and improving optimization algorithms that work particularly well under these conditions. A common approach is to tackle HPO by estimating a local or global structure of the objective landscape by some form of predictive model. This introduces additional overhead and complexity with the aim of reducing the overall number of expensive objective evaluations necessary to find an approximate optimum. Typical representatives of this approach are Bayesian optimization (BO) [8] algorithms and frameworks based on BO, which are global optimization schemes based on a non-linear regression model, e.g., a Gaussian process or random forest. They have shown significant improvements in performance compared to other methods [9] but carry a significant overhead. Furthermore, BO is somewhat difficult to parallelize due to its sequential nature, although many variants exist (e.g. [10]–[13]).

Multi-fidelity HPO (MF-HPO) algorithms aim to accelerate the optimization process by exploiting cheaper proxy functions of the objective function itself (e.g., by training ML models on a smaller subsample of the available training data, or by running fewer training iterations). Bandit-based algorithms like Hyperband (HB) [14] have become particularly popular because of their good trade-off between optimization performance and simplicity.

Progress in the field of HPO often consists of iterative improvements of established algorithms. Considerable work exists, for example, to improve the limitations of HB: Asynchronous successive halving (ASHA) [15] proposes a sophisticated way to make efficient use of parallel resources, BO Hyperband (BOHB) [16] improves performance during later parts of a run by incorporating surrogate assistance into HB, and asynchronous BOHB (A-BOHB) [17] unites a bandit-based optimization scheme using model-based guidance with asynchronous parallelization.

*Equal Contribution

While these conceptual extensions of HPO all have their respective merit, it is often somewhat overlooked that the simplicity of an optimization algorithm (i.e., how difficult modifications and extensions are, and on how many dependencies a system relies [18]) heavily influences its adoption in practice. Random search (RS), for example, still enjoys great popularity, as it is extremely simple to implement and parallelize, has almost no overhead, and is able to take advantage of the aforementioned low effective dimensionality [7]. Furthermore, algorithmic developments identify and address limitations of prior research, but rarely question core algorithmic choices that have been made in the original implementation. Many multi-fidelity algorithms, for example, are extensions and further developments of HB that take the fixed successive halving schedule [19] for granted. The process of designing a good MF-HPO optimizer in practice – and many other algorithmic solutions in science in general – can therefore often feel somewhat like a "manual stochastic local search on the meta level". The drawback of this manual procedure is that the design space of all HPO algorithms is not systematically searched, and parts of the design space are excluded by prior algorithmic decisions. If "established" algorithms are not challenged, there is a risk that algorithms that work well will be overlooked., and it is often hard to identify what algorithmic components make a difference. In particular, it is possible that overly complicated algorithms are developed by extending "established" designs, only some of which contribute meaningfully to performance gains. Sometimes certain technical components of an algorithm, which are neither exposed nor discussed in detail, may also influence performance significantly.

A. Contributions

We make a principled demonstration of how HPO algorithm design can be performed systematically and automatically with a benchmark-driven approach following the programming-by-optimization paradigm [20]. In particular, the contributions of this work are:

- **Formalization**: We formalize the design space of MF-HPO algorithms and demonstrate that established MF-HPO algorithms represent instances within this space.
- **Framework**: Based on this formalization, we present a rich, configurable framework for MF-HPO algorithms, whose software implementation we call SMASHY (Surrogate Model Assisted HYperband).
- **Configuration**: Based on the formalization and framework, we follow an empirical approach to design an MF-HPO algorithm by optimization, given a large benchmark suite. This configuration procedure does not only consider performance, but also, e.g., the simplicity of the design.
- **Benchmark**: As in general any HPO algorithm will be applied in a diverse set of application scenarios, we evaluate the performance of our newly designed algorithm on a representative set of problems that were not previously used for its configuration (i.e., a clean test-set approach on the meta-level) and compare them with established implementations of HPO methods.
- **Explanation**: For the resulting MF-HPO system, we systematically assess and explain the effect of different design choices on overall algorithmic performance. Furthermore, we investigate the behavior of algorithmic design components in the context of specific problem scenarios; i.e., we investigate which algorithmic components lead to performance improvements for simple HPO with numeric hyperparameters, AutoML pipeline configuration, and neural architecture search.

II. Related Work

HPO is one of the most essential components of current AutoML methods [1], and MF-HPO has recently become more prominent, given that cheap, low-fidelity evaluations have proven useful to speed up optimization, especially for expensive HPO of complex ML algorithms on larger data sets [14]. While AutoML tools have historically relied on a limited set of HPO methods, we argue that the optimal HPO method depends on problem characteristics, and therefore a systematic development of HPO methods under consideration of problem characteristics is required. Approaches towards such systematic development have often relied on a *high-level* language or template that allows expressing solution algorithms for a given problem class, e.g. to solve constraint satisfaction problems [21]–[23], satisfiability problems [24], scheduling problems [25], or general multi-objective combinatorial problems [26] [27].

Even if a high-level language is available, manual configuration of such frameworks is laborious and requires expert knowledge. This motivates the design philosophy of "Programming by Optimization" [20] (PBO), which advocates for allowing algorithmic choices in a software system (instead of fixing them at the time of implementation) and automatic configuration by optimization for a given problem context.

As one approach to automatic and efficient algorithm configuration, racing-based strategies have been used to design optimization algorithms. For example, iterated F-RACE [28] has been used for the automatic design of multi-objective ant colony optimization algorithms [26]. Similarly, IRACE [29] has been used for the automatic design multi-objective evolutionary algorithms [27] or to meta-configure the parameters IRACE itself [30]. Another commonly used framework is SMAC [5], which extends the sequential model-based optimization paradigm (SMBO, see also Section IV-A2) to an algorithm configuration setting. This is achieved through the use of an intensification procedure that governs across how many problem instances each configuration is evaluated, trading off computational cost against confidence regarding the superiority of a given configuration. While such intensification mechanisms have been used in other work before [31] [32], SMAC also uses instance features describing properties of a problem instance are used to train the empirical performance model predicting the performance of a configuration on a new problem instance. Besides racing and sequential model-based approaches, genetic algorithms have also been used to evolve optimal solvers [33].

We argue that the design of HPO algorithms can be seen as an instance of PBO. However, while there are many approaches that focus on individual algorithmic choices (e.g., the choice of a surrogate model for BO [34]), we are not aware of many cases where PBO is applied to designing HPO systems themselves. One exception is [35], who use SMACv3 [36] to automatically configure Bayesian optimization (BO) for HPO from a flexible search space of components. We take a similar approach here in that the algorithmic choices are exposed as hyperparameters that can be tuned. However, unlike [35], we do not configure an established HPO method (such as BO) with a predefined structure and associated control

parameters (e.g., varying the surrogate model of BO). Instead, we introduce a *new* configurable algorithmic framework, which covers many different MF-HPO structures, including well-established principles for multi-fidelity handling (e.g., Successive Halving) as well as new approaches (e.g., equal batch size in all proposals).

In addition to designing well-performing algorithms, it is equally important to facilitate an understanding of the effects of all considered design choices. The field of *sensitivity analysis* (SA) comprises a multitude of methods to assess the importance of input factors on the output of a mathematical model [37]. Functional ANOVA (fANOVA) methods, which decompose the response of a (mathematical) model or function into lower-order components, are a widely studied method in the field of SA, dating back to [38]. This class of methods has also become popular in the field of ML to analyze the importance of hyperparameters [39].

Popular ways of analyzing effects of algorithmic effects in ML and algorithm configuration are *ablation studies* [40]. This involves measuring the performance when removing one or more of algorithmic subcomponents to understand the relative contribution of the ablated components to overall performance. There are different ways of performing an ablation analysis; probably the most common approach is *leave-one-component-out* (LOCO) ablation [41]. In the context of algorithm configuration, [40] proposes an ablation approach that links a source configuration (e.g., the default) to a target (e.g., the optimized configuration) through an ablation path.

Nevertheless, many existing works that propose or improve HPO or algorithm configuration systems do not analyze the algorithmic choices of an optimized system, and the ones that do perform relatively straightforward analyses. For example, [21] compare the designs their approach finds automatically to the designs expert humans generated. [42] perform ANOVA and non-parametric Friedman tests to investigate in detail the effects that algorithmic choices, found through automatic configuration [26], have on the performance of multi-objective ant colony optimization algorithms. [43] present a modular framework for CMA-ES variants on which they perform optimization; in particular, they investigate how the optimized configuration changes when the search space is enlarged by introducing new components.

III. Methodology

A. Supervised Machine Learning

Supervised ML typically deals with a dataset (which is, mathematically speaking, a tuple) $\mathcal{D} = \left((\mathbf{x}^{(i)}, y^{(i)})\right) \in (\mathcal{X} \times \mathcal{Y})^n$ of n observations, assumed to be drawn i.i.d. from a data-generating distribution $\mathbb{P}_{xy}$. An ML model is a function $\hat{f} : \mathcal{X} \to \mathbb{R}^g$ that assigns a prediction to a feature vector from $\mathcal{X}$.[1] $\hat{f}$ is itself constructed by an *inducer* function $\mathcal{I}$, i.e., the model-fitting algorithm. The inducer $\mathcal{I} : (\mathcal{D}, \boldsymbol{\lambda}) \mapsto \hat{f}$ uses training data $\mathcal{D}$ and a vector of *hyperparameters* $\boldsymbol{\lambda} \in \Lambda$ that govern its behavior. The overall goal of supervised ML is to derive a model $\hat{f}$ from a data set $\mathcal{D}$ so that $\hat{f}$ predicts data sampled from $\mathbb{P}_{xy}$ best. The quality of a prediction is measured as the discrepancy between predictions and ground truth. This is operationalized by the loss function $L : \mathcal{Y} \times \mathbb{R}^g \to \mathbb{R}_0^+$, which is to be minimized during model fitting. In contrast to the optimisation problems that we will define in Sections III-B and III-C, we term this the "first level" optimisation problem.

The expectation of the loss value of predictions made for data samples drawn from $\mathbb{P}_{xy}$ is the *generalization error*

$$GE := \mathbb{E}_{(\mathbf{x},y)\sim\mathbb{P}_{xy}}\left[L(y, \hat{f}(x))\right] \tag{1}$$

which cannot be computed directly if $\mathbb{P}_{xy}$ is not known beyond the available data $\mathcal{D}$. Therefore, one often uses so-called *resampling* techniques that fit models on N_{iter} subsamples $\mathcal{D}[J_j]$ and evaluate them on complements $\mathcal{D}[-J_j]$ of these subsets to obtain an estimate of the generalization error

$$\widehat{GE}(\mathcal{I}, \boldsymbol{\lambda}, \mathbf{J}) = \frac{1}{N_{\text{iter}}} \sum_{j=1}^{N_{\text{iter}}} L\big(y[-J_j], \mathcal{I}\left(\mathcal{D}[J_j], \boldsymbol{\lambda}\right)\left(x[-J_j]\right)\big). \tag{2}$$

Depending on the resampling method, the inducer $\mathcal{I}$, and the quantity of data in $\mathcal{D}$, estimating the generalization error $\widehat{GE}(\mathcal{I}, \boldsymbol{\lambda}, \mathbf{J})$ can require large amounts of computational resources.

B. Hyperparameter Optimization

The goal of HPO is to identify a hyperparameter configuration that performs well in terms of the estimated generalization error in Equation (2). Often, optimization only concerns a subspace of available hyperparameters because some hyperparameters might be set based on prior knowledge or due to other constraints. One would therefore split up the space of hyperparameters Λ into a subspace of hyperparameters Λ_S over which optimization takes place, and the remaining hyperparameters $\Lambda_C = \Lambda / \Lambda_S$ for which values $\boldsymbol{\lambda}_C$ are given exogenously. We define the HPO problem as:

$$\boldsymbol{\lambda}_S^* \in \underset{\boldsymbol{\lambda}_S \in \Lambda_S}{\operatorname{argmin}}\ c(\boldsymbol{\lambda}_S) = \underset{\boldsymbol{\lambda}_S \in \Lambda_S}{\operatorname{argmin}}\ \widehat{GE}(\mathcal{I}, (\boldsymbol{\lambda}_S, \boldsymbol{\lambda}_C), \mathbf{J}). \tag{3}$$

Here, $\boldsymbol{\lambda}_S^*$ denotes a theoretical optimum, and $c(\boldsymbol{\lambda}_S)$ is a shorthand for the estimated generalization error in Equation (2). We refer to Problem 3 as the "second level" optimisation problem.

Hyperparameters can be either continuous, discrete, or categorical, and search spaces are often a mix of the different types. The search space may be hierarchical, i.e., some subordinate hyperparameters can only be set in a meaningful way if another parent hyperparameter takes a certain value. In particular, many AutoML frameworks perform optimization over a hierarchical hyperparameter space that represents the components of a complex ML pipeline [1].

Many HPO algorithms can be characterized by how they handle two different trade-offs: (a) The exploration vs. exploitation trade-off refers to how much budget an optimizer spends on either trying to directly exploit the currently available knowledge base by evaluating very close to the currently best candidates (e.g., local search) or whether it explores the search space to gather new knowledge (e.g., random search). (b) The inference vs. search trade-off refers to how much time and overhead is spent to induce a model from the currently available archive data in order to exploit past evaluations as much as possible. Other relevant aspects that HPO algorithms differ in are: *Parallelizability*, i.e., how many configurations a tuner can (reasonably) propose at the same time; *global vs. local* behavior of the optimizer, i.e., if updates are always quite close to already evaluated configurations; *noise handling*, i.e., if the optimizer takes into account that the estimated generalization

[1] where g allows handling of multi-output regression, as well as multiclass classification with g classes by returning decision scores.

error is noisy; *search space complexity*, i.e., if and how hierarchical search spaces can be handled; *multi-fidelity*, i.e., if the optimizer uses cheaper evaluations to infer performance on the full data.

Multi-fidelity methods make use of the fact that the resampling procedure in Equation (2) can be modified in multiple ways to make evaluation cheaper: one can (i) reduce the training sizes $|J_j|$ via subsampling, as model evaluation complexity is often at least linear in training set size, or (ii) change some components in $\boldsymbol{\lambda}$ in a way that makes model fits cheaper. Examples of (ii) are reducing the overall number of training cycles performed by a neural network fitting process, or reducing the number of base learner fits in a bagging or boosting method. These modifications can both increase the variance of $\widehat{GE}$ and introduce an (often pessimistic) bias, as models trained on smaller datasets or with values of $\boldsymbol{\lambda}$ that make fitting cheaper often have worse generalization errors.

We introduce a *fidelity* parameter $r \in (0, 1]$ that influences the resource requirements of the evaluation of $\widehat{GE}$ and define

$$c(\boldsymbol{\lambda}_S; r) := \widehat{GE}\left(\mathcal{I}, \left(\boldsymbol{\lambda}_S, \boldsymbol{\lambda}_C(r)\right), \mathbf{J}(r)\right). \tag{4}$$

With this definition we make the choice that r should influence the evaluation cost of $\widehat{GE}$ only by modifying the resampling, $J(r)$ or by modifying a hyperparameter $\boldsymbol{\lambda}_C(r)$. Typically, r only affects one of these aspects at a time, and if it affects $\boldsymbol{\lambda}_C$, it only affects a single hyperparameter dimension.

Note that we normally assume that a higher fidelity r returns a better model in terms of the estimate of the generalization error, and the best estimate is returned for $r = 1$. Therefore, r enters the expression in a way where it can influence performance, but is not searched over. We define $c(\boldsymbol{\lambda}_S) := c(\boldsymbol{\lambda}_S; 1)$ as in [44], and the optimization problem remains as in Equation (3).

This assumption may be violated in some scenarios, and model performance could worsen for a higher value of r (e.g., a neural network, which may overfit on a small dataset if trained for too many epochs). In this case, we define the optimization problem as $(\boldsymbol{\lambda}_S^*, r^*) \in \operatorname{argmin}_{\lambda_S \in \Lambda_S, r \in (0,1]} c(\lambda_S; r)$.

The resource requirements of evaluating $c(\boldsymbol{\lambda}; r)$ can have a complicated relationship with $\boldsymbol{\lambda}$ and r; in practice, r is chosen in such a way that it has an overwhelming and linear influence on resource demand. The overall cost of optimization up to a given point in the optimization process is therefore assumed to be the cumulative sum of the values of r of all evaluations of $c(\boldsymbol{\lambda}; r)$ up to that point. We can also interpret r as the fraction of the budget of a single full fidelity model evaluation that must be spent for evaluating $c(\boldsymbol{\lambda}; r)$.

Given the definition of the HPO problem, we present an (MF-)HPO algorithm for a single, synchronous worker in its most generic form in Algorithm 1. Until a pre-determined budget is exhausted, such an algorithm decides in every iteration (a) which configuration(s) $\boldsymbol{\lambda}_S$ to evaluate next and (b) which fidelity r to use for evaluation; non-multi-fidelity algorithms set this to $r = 1$ as default. The algorithm makes use of an *archive* $\mathcal{A}$, a database recording previously proposed hyperparameter configurations and, if available, their evaluation results. This database can be shared among multiple worker processes that optimize concurrently.

Algorithm 1 A generic HPO algorithm

1: **while** budget is not exhausted **do**
2: Propose $\left(\boldsymbol{\lambda}_S^{(i)}, r^{(i)}\right), i = 1, ..., k$, based on archive $\mathcal{A}$
3: Write proposals into a shared archive $\mathcal{A}$
4: Estimate generalization error(s) $c\left(\boldsymbol{\lambda}_S^{(i)}; r^{(i)}\right)$
5: Write results into shared archive $\mathcal{A}$
6: **end while**
7: Wait for workers to synchronize
8: Return best configuration in archive $\mathcal{A}$

The optimization process can be accelerated by making efficient use of parallel resources. We distinguish between *synchronous* and *asynchronous* scheduling. The former starts multiple evaluations synchronously at the same time and waits until all of these have finished. To be more precise, a number of $k > 1$ configurations is proposed in line 2 and evaluated in parallel in line 4, all within the inner loop of Algorithm 1. Given K available parallel resources, it should be ensured that the number k of configurations scheduled in parallel is not significantly smaller than K and that the evaluation runtimes amongst these k configurations do not differ significantly in order to avoid unnecessarily idling single parallel resources. In contrast, for asynchronous scheduling, Algorithm 1 is run individually in K separate worker processes. Given a shared archive that is synchronized between the workers, every worker can independently schedule new configurations to evaluate.

C. Algorithm Design and Configuration

Our goal will be to design and configure a new HPO algorithm based on a superset of design choices included in previously published HPO methods. We are interested in finding a configuration (or making design choices) based on a set of training instances that works across a broad set of future problem instances. This problem is called *algorithm configuration* [5], [45]. It is quite similar to HPO; a major difference is that algorithm configuration optimizes the configuration of an arbitrary algorithm over a diverse set of often heterogeneous instances for optimal average performance, while HPO performs a per-instance configuration of an ML inducer for a single data set. We introduce the following notation for consistency with the relevant literature: $\boldsymbol{\gamma}$ denotes configuration parameters controlling our optimizer A, while $\boldsymbol{\lambda}$ denotes hyperparameters optimized by our optimizer, controlling our inducer $\mathcal{I}$. The algorithm configuration problem can be formally stated as follows: Given an algorithm $A : \Omega \times \Gamma \to \Lambda$ parametrized by $\boldsymbol{\gamma} \in \Gamma$ and a distribution $\mathbb{P}_\Omega$ over problem instances Ω together with a cost metric ζ, we must find a parameter setting $\boldsymbol{\gamma}^*$ that minimizes the expected $\zeta(A)$ over $\mathbb{P}_\Omega$:

$$\boldsymbol{\gamma}^* \in \operatorname*{argmin}_{\boldsymbol{\gamma} \in \Gamma} \mathbb{E}_{\omega \sim \mathbb{P}_\Omega} \left[\zeta(A(\omega, \boldsymbol{\gamma}))\right]. \tag{5}$$

In our example, Γ corresponds to the space of possible components of our HPO method and Ω to a class of HPO problems (i.e., ML methods and datasets on which they are evaluated) for which their configuration should be optimal. Based on a training set of representative instances $\{\omega_i\}$ drawn from $\mathbb{P}_\Omega$, a configuration $\boldsymbol{\gamma}^*$ that minimizes c across these instances should be chosen through optimization. When necessary, we refer to this process as the "third level" optimization problem to distinguish it from the optimization performed by the HPO algorithm A, i.e., the second level optimization.

IV. Formalizing a broad class of MF-HPO Algorithms

We aim to find an HPO algorithm that performs particularly well in the multi-fidelity setting. To design an algorithm by optimization, we propose a framework and search space of HPO algorithm candidates that covers a large class of possible algorithms and focus on a subclass of algorithms similar to Hyperband because of their favorable properties. This subclass focuses on multi-fidelity algorithms that use a pre-defined schedule of geometrically increasing fidelity evaluations containing algorithms like Hyperband [14] and BOHB [16].

The basis of this framework is presented in Algorithm 2, which can be configured by combining algorithmic building blocks in novel ways. The main difference to Algorithm 1 is that the *Propose* part is specified more explicitly. At its core, Algorithm 2 consists of two parts: (i) sampling new configurations at low fidelities (lines 2–7) and (ii) increasing the fidelity for existing configurations (lines 8–14). In contrast to Algorithm 1, Algorithm 2 makes use of state variables t, b, and r to account for optimization progress. However, these variables are only shown in Algorithm 2 for clarity and can, in principle, be inferred from the archive $\mathcal{A}$. As argued in Section III, every single worker instance of Algorithm 1 can, in principle, be scheduled asynchronously, but we do not consider this in this work.

In its first iteration, Algorithm 2 uses a SAMPLE-subroutine to initialize the initial batch C of μ solution candidates. The fidelity of the evaluation of the proposed configurations is refined iteratively; when all configurations in the batch have been evaluated with given fidelity r, the top $1/\eta_{\text{surv}}$ fraction of configurations is evaluated with a fidelity that is increased by a factor of η_{fid}. When the fidelity cannot be further increased for a batch because all of its configurations were evaluated at full fidelity $r = 1$, they are set aside, and a new batch of configurations is sampled.

The SAMPLE subroutine creates new configurations to be evaluated, possibly using information from the archive to propose points that are likely to perform well. We allow that any inducer $\mathcal{I}_{f_{\text{sur}}}$ that produces a surrogate model f_{sur} can be used for model-assisted sampling. The subroutine works by at first sampling a number of points from a given generating distribution $\mathbb{P}_{\boldsymbol{\lambda}}(\mathcal{A})$. The performance of these points is then predicted using the surrogate model, and points with unfavorable predictions are discarded in a process we refer to as *filtering*. This process is repeated until the requested number μ of non-discarded points is obtained. N_{s} and ρ have the same function as in [16] (see Section IV-A5), with the filter factor N_{s} controlling the number of sampled points needed for each of the μ points returned, and ρ controlling the fraction of points that are not filtered. Thus, the configuration space of sampling methods also includes purely random sampling, as in Hyperband, by setting $\rho = 1$. The influence of the surrogate model on sampled candidates is larger when (i) the number of sampled configurations N_{s} is large, or (ii) the fraction ρ of candidates sampled at random is small. We present two slightly different SAMPLE algorithms: SAMPLETOURNAMENT (Algorithm 3) and SAMPLEPROGRESSIVE (Algorithm 4) based on this principle (see Appendix A). Both allow to use different N_{s} values for different points they sample, parameterized by N_{s}^0 and N_{s}^1.

While hyperparameters $\boldsymbol{\lambda}_S$ are proposed by one of the two SAMPLE methods, the fidelity hyperparameter r follows a fixed schedule similar to Successive Halving [19] and Hyperband [14], with a few extensions. For one, the survivor factor η_{surv} can be a different value from the fidelity scaling factor η_{fid}. Furthermore, the algorithm allows three scheduling modes, controlled by *batch_method*: `SH` does Successive Halving. The `HB` mode evaluates brackets, as performed by Hyperband. While $\mu(b)$ is, in principle, a free configuration parameter for every value of b, we choose to set $\mu(b)$ so that total budget expenditure is approximately equal between all brackets. This follows the principle used in Hyperband, but the dependency on η_{surv} and η_{fid} is more complex and determined dynamically. Finally, `equal` *batch_method* uses equal batch sizes for every evaluation. Individuals that perform badly at low fidelity are removed, as in SH, but new individuals are sampled to fill up batches to the original size. Because new individuals are added to the batches at all fidelity steps, it is not necessary to use brackets with different initial fidelities, and therefore, only a single repeating bracket $b = 1$ is used. The `equal` method is an original contribution of this work and was designed to be similar to `HB` while using parallel resources more efficiently; the two batch scheduling methods are illustrated in Figure 1.

If the exploration-exploitation tradeoff is not balanced properly, the optimization progress can either stagnate or function evaluations are wasted due to too much exploration of uninteresting regions of the search space. However, the relative importance of exploration and exploitation can change throughout the course of optimization, where exploration performed later during the optimization is not as useful as during the beginning. The given configuration space makes it possible to make the exploration-exploitation tradeoff dependent on optimization progress by providing the option to make $\rho(t)$ and $\left(N_{\text{s}}^0(t), N_{\text{s}}^1(t)\right)$ dependent on the proportion of exhausted total budget at every configuration proposal step. It is likely that large values of $\rho(t)$ / small values of $N_{\text{s}}^{\cdot}(t)$ perform better when t is small. Conversely, it is likely that small $\rho(t)$ / large $N_{\text{s}}^{\cdot}(t)$ work well for large t.

A. Common MF-HPO Algorithms Covered by Algorithm 2

The following describes a few common HPO algorithms that can be instantiated within this framework; see Table I for specific configuration parameter settings within Algorithm 2 that correspond to these algorithms.

1) Random search (RS): Configurations $\boldsymbol{\lambda}_S$ are drawn (uniformly) at random, and every configuration is evaluated with full fidelity $r = 1$. Parallelization is straightforward, as configurations are drawn independently.

2) Bayesian Optimization (BO) [8]: The configuration that maximizes an acquisition function $a(\boldsymbol{\lambda})$ (e.g., expected improvement, EI [4]) is proposed and evaluated with the full fidelity $r = 1$. $a(\boldsymbol{\lambda})$ is based on a surrogate model trained on the archive $\mathcal{A}$. BO can be parallelized by either using methods that can propose multiple points at the same time using a single surrogate model or, alternatively, by fitting a surrogate model on the anticipated outcome of configurations that were proposed but not yet evaluated [11]. BO can be represented in Algorithm 2 by using an inducer $\mathcal{I}_{f_{\text{surr}}}$ that produces a function f_{surr} equal to the composition of model prediction and acquisition function. In its basic form, BO is not an MF algorithm and therefore always sets $r = 1$.

3) Successive halving (SH) [19]: Successive halving, also called Sequential Halving [46], is a simple multi-fidelity optimization algorithm that combines random sampling of configurations with a fixed schedule for r. At the beginning, a batch of μ configurations is sampled randomly and evaluated with an initial fidelity $r_{\text{min}} < 1$. This is

(a)

	$b=1$		$b=2$		$b=3$		$b=4$	
i	$\|C\|$	r	$\|C\|$	r	$\|C\|$	r	$\|C\|$	r
1	8	1/8	6	1/4	4	1/2	4	1
2	4	1/4	3	1/2	2	1		
3	2	1/2	1	1				
4	1	1						

(b)

	$b=1$	
i	$\|C\|$	r
1	8	1/8
2	8	1/4
3	8	1/2
4	8	1

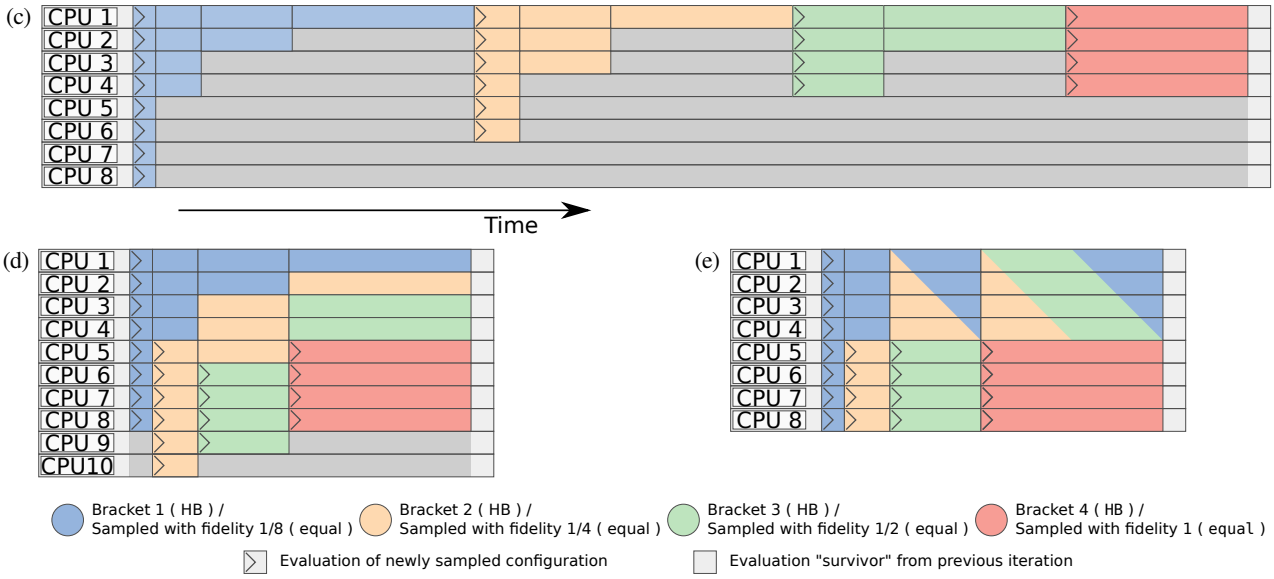


Fig. 1: Illustration of the different *batch_method*s used, corresponding to values of $\eta_{fid} = \eta_{surv} = 2$, $s = 4$, $\mu = 8$.
The tables show (a) the `HB` method and (b) the `equal` method. Shown are the number $|C|$ and fidelity value r of configurations being evaluated in the iterations i of the various brackets counted by b. Except for i, the variables are the same as in Algorithm 2.
Subfigures (c) - (e) illustrate resource utilization by the batch methods, given availability of parallel resources. (c): Naively scheduling the configuration evaluations one batch after another can make use of available parallel resources, but leaves many of them idle. (d): Hypothetical way of scheduling configuration evaluations of different brackets at the same time so that all configurations with the same r-value are scheduled together utilizes resources more efficiently, but the number of evaluations in each batch still varies. (e): The simpler `equal` batch scheduling method always evaluates the same number of configurations within each batch and, therefore, makes optimal use of available parallel resources.

TABLE I: RS, BO, SH, HB, BOHB as instances of Algorithm 2. η, ρ, N_s are configuration parameters of the respective algorithms. "—" denotes that the value has no influence on the algorithm in this configuration.
*: BO and BOHB use inducers that produce non-standard model functions, which do not aim to predict the actual performance of configurations, and instead calculate the value of an acquisition function such as EI [4] (for BO) or the ratio of two kernel density estimator (KDE) models (for BOHB).
†: In a small departure from BOHB, Algorithm 2 uses the KDE estimate of good points for all sampled points, even when randomly interleaved. BOHB randomly interleaves from a uniform distribution.

Algorithm	$\mu(b)$	s	η_{surv}	η_{budget}	$\mathcal{I}_{f_{\text{sur}}}$	ρ	N_s	*batch_mode*	$\mathbb{P}_{\boldsymbol{\lambda}}(\mathcal{A})$
RS	—	1	—	—	—	1	—	—	uniform
BO	1	1	—	—	e.g. GP+EI*	ρ	N_s	—	uniform
SH	μ	$\lfloor -\log_\eta(r_{\min}) \rfloor + 1$	η	η	—	1	—	SH	uniform
HB	$\lceil s \cdot \frac{\eta^{s-b}}{s-b+1} \rceil$	$\lfloor -\log_\eta(r_{\min}) \rfloor + 1$	η	η	—	1	—	HB	uniform
BOHB	$\lceil s \cdot \frac{\eta^{s-b}}{s-b+1} \rceil$	$\lfloor -\log_\eta(r_{\min}) \rfloor + 1$	η	η	TPE*	ρ	N_s	HB	KDE†

Algorithm 2 SMASHY algorithm

Configuration Parameters: batch size schedule $\mu(b)$, number of fidelity stages s, survival rate η_{surv}, fidelity rate η_{fid}, SAMPLE method (either SAMPLETOURNAMENT or SAMPLEPROGRESSIVE), *batch_method* (one of `equal`, `SH`, or `HB`), total budget B; further configuration parameters of SAMPLE: $\mathcal{I}_{f_{\text{sur}}}, \mathbb{P}_{\boldsymbol{\lambda}}(\mathcal{A}), \rho(t), \left(N_{\text{s}}^0(t), N_{\text{s}}^1(t)\right), n_{\text{trn}}$.

State Variables: Expended budget fraction $t \leftarrow 0$, bracket counter $b \leftarrow 1$ (remains 1 for *batch_method* $\in$ {`equal`, `SH`}), current fidelity $r \leftarrow 1$, batch of proposed configurations $C \leftarrow \emptyset$

while $t < 1$ **do**
  if $r = 1$ **then** ▷ Generate new batch of configurations
    $r \leftarrow (\eta_{\text{fid}})^{b-s}$
    $C \leftarrow \text{SAMPLE}\big(\mathcal{A}, \mu(b), r; \mathcal{I}_{f_{\text{sur}}}, \mathbb{P}_{\boldsymbol{\lambda}}(\mathcal{A}), \rho(t), \left(N_{\text{s}}^0(t), N_{\text{s}}^1(t)\right), n_{\text{trn}}\big)$
    if *batch_method* = `HB` **then**
      $b \leftarrow (b \bmod s) + 1$
    end if
  else ▷ Progress fidelity
    $r \leftarrow r \cdot \eta_{\text{fid}}$
    $C \leftarrow \text{SELECT\_TOP}\,(C, |C|/\eta_{\text{surv}})$
    if *batch_method* = `equal` **then**
      $\tilde{\mu} \leftarrow \mu(b) - |C|$
      $C \leftarrow C \cup \text{SAMPLE}\big(\mathcal{A}, \tilde{\mu}, r; \mathcal{I}_{f_{\text{sur}}}, \mathbb{P}_{\boldsymbol{\lambda}}(\mathcal{A}), \rho(t), \left(N_{\text{s}}^0(t), N_{\text{s}}^1(t)\right), n_{\text{trn}}\big)$
    end if
  end if
  Evaluate configuration(s) $c\,(\boldsymbol{\lambda}_S; r)$ for all $\boldsymbol{\lambda}_S \in C$
  Write results into shared archive $\mathcal{A}$
  $t \leftarrow t + r \cdot |C|/B$ ▷ Update budget spent
end while

followed by repeated "halving" steps, where the top fraction η^{-1} of configurations is kept and evaluated after r is increased by a factor of η, until the maximum fidelity value is reached. The schedule is chosen to keep the total sum of all evaluated r constant in each batch. Both η_{surv} and η_{fid} in Algorithm 2 correspond to SH's η-parameter.

4) Hyperband (HB) [14]: Similar to SH, Hyperband uses a fixed schedule for the fidelity parameter r, but it augments SH by using multiple *brackets* b of SH runs starting at different $r_{\text{min}}(b)$ and with different $\mu(b)$. The number of brackets is set to

$$s = \lfloor \log_\eta(1/r_{\text{min}}) \rfloor + 1, \tag{6}$$

which coincides with the number of fidelity steps that can be performed on a geometric scale on the interval $[r_{\text{min}}, 1]$. In bracket $b \in \{1, 2, \ldots, s\}$, a number of $\mu(b)$ samples are initially sampled and evaluated with initial fidelity $r = \eta^{s-b}$. $\mu(b)$ is chosen such that each bracket needs an approximately similar amount of budget: $\mu(b) = \lceil s \cdot \frac{\eta^{s-b}}{s-b+1} \rceil$.

5) Bayesian Optimization Hyperband (BOHB) [16]: Model-based methods outperform Hyperband when a relatively large amount of budget is available and many objective function evaluations can be performed. BOHB was created to overcome this drawback. This method iterates through successive halving brackets like Hyperband, but, instead of sampling new configurations randomly, it uses information from the archive to propose points that are likely to perform well. A total number of N_s configurations are proposed for evaluation; ρ are sampled at random, and the rest are chosen based on a surrogate model induced on the evaluated configurations in $\mathcal{A}$. The models used by BOHB are a pair of kernel density estimators of the top and bottom configurations in $\mathcal{A}$, similar to the process in [47]. To implement BOHB in Algorithm 2, one therefore needs to use an inducer $\mathcal{I}_{f_{\text{sur}}}$ that produces a function that calculates the ratio of kernel densities, an unusual kind of regression model.

B. Limitations and Further MF-HPO Algorithms

The following lists notable HPO algorithms not currently covered by the optimization space of Algorithm 1. They were excluded because they differ in too substantial ways from the other algorithms considered here.

1) FABOLAS [48]: Fabolas is a continuous multi-fidelity BO method, where the conditional validation error is modelled as a Gaussian process using a complex kernel-capturing covariance with the training set fraction $r \in (0, 1]$ to allow for adaptive evaluation at different resource levels.

2) Asynchronous successive halving (ASHA) [15] and asynchronous Hyperband: Hyperband, as well as SH, have the drawback that batch sizes decrease throughout the stages of an SH run, preventing efficient utilization of parallel resources. ASHA is an effective method to parallelize SH by an asynchronous parallelization scheme. A shared archive across a number of different workers is maintained. Instead of waiting until all n configurations of a batch have been evaluated for fidelity r, every free worker queries the shared archive $\mathcal{A}$ for "promotable" configurations (i.e., configurations that belong to the fraction of top η^{-1} configurations evaluated with the same fidelity). Asynchronous Hyperband works similarly.

3) Asynchronous BOHB (A-BOHB) [17]: A-BOHB, an asynchronous extension of BOHB where configurations are sampled from a joint Gaussian Process, explicitly capturing correlations across fidelities. In contrast to ASHA and asynchronous versions of BOHB in the original BOHB publication [16], A-BOHB does not perform synchronization after each stage but instead uses a stopping rule [49] to asynchronously determine whether a configuration should continue to run or be terminated.

V. EXPERIMENTAL ANALYSIS

Given the formalization of the framework in Section IV, our goal is to find the best representative (out of this class of algorithms) by solving the third-level optimization problem in Equation (5), and explain the role of specific algorithmic components in a benchmark-driven approach. We aim to answer the following research questions:

RQ1: How does the optimal configuration of our MF-HPO framework differ between problem scenarios, i.e., do different problem scenarios benefit from different HPO algorithms?

RQ2: How does our optimized MF-HPO algorithm compare to other established HPO implementations?

RQ3: Does the successive-halving fidelity schedule have an advantage over the simpler equal-batch-size schedule?

RQ4: What is the effect of using multi-fidelity methods in general?

RQ5a: Does changing SAMPLE configuration parameters throughout the optimization process offer an advantage?

TABLE II: Three benchmark collections of YAHPO Gym used in our benchmark.

			Hyperparameter Types					
Scenario	Target Metric	d	Cont.	Integer	Categ.	Hierarchical	# Instances	# Training Set
lcbench: HPO of a neural network	cross entropy loss	7	6	1	0	✗	35	8
rbv2_super: AutoML pipeline configuration	log loss	38	20	11	7	✓	89	30
nb301: Neural architecture search	validation accuracy	34	0	0	34	✓	1	—

RQ5b: Does (more complicated) surrogate-assisted sampling in SAMPLE provide an advantage over using simple random sampling with surrogate filtering?

RQ6: What effect do different surrogate models (or using no model at all) have on performance?

RQ7: Does the equal-batch-size schedule give an advantage over established methods when parallel resources are available?

We rely on benchmark scenarios of the YAHPO Gym benchmark suite [50], each of which provides a number of related *instances* of optimization problems. The benchmark scenarios we have chosen cover three important application areas of AutoML: Hyperparameter optimization of a neural network (*lcbench*), AutoML pipeline configuration (*rbv2_super*), and neural architecture search (*nb301*). These classes of problems do not only represent common and relevant tasks for researchers and practitioners in the field; as presented in Table II, they are also quite different with regards to: (1) the dimensionality of the search space, (2) hyperparameter types (categorical, integer, continuous), and (3) whether there are hierarchical dependencies between hyperparameters. More details on the characteristics of the problem classes are given in Appendix B. To avoid an optimistic bias in the analysis caused by over-adaption to the random peculiarities of the particular instances used during configuration, we are using meta-holdout splits on the level of HPO problem instances (see Appendix IV). This means that for analysing the performance of a configured candidate of Algorithm 2, we are evaluating this candidate by running it on instances that were not seen during configuration. Algorithm 2 is always run with a budget limit corresponding to $30 \cdot d$ full fidelity evaluations (where d is the dimension of the problem instance).

A. Algorithm Design via Configuration

First, we describe the experiments we conducted to configure Algorithm 2 via optimization.

We follow the PBO principle, and configure Algorithm 2 by optimizing separately for different HPO scenarios, namely for *lcbench* and *rbv2_super*, resulting in two optimized configurations $\boldsymbol{\gamma}^{*\textit{lcbench}}$ and $\boldsymbol{\gamma}^{*\textit{rbv2\_super}}$, respectively. The *nb301* scenario is not used for configuration, but exclusively for subsequent analysis.

For the algorithm configuration of our framework (third level), the performance objective $\mathbb{E}_{\omega\sim\mathbb{P}_\Omega}\left[\zeta(A(\omega,\boldsymbol{\gamma}))\right]$ for a configuration γ in Equation (5) is estimated by running Algorithm 2 (i.e., second level optimization) configured by $\boldsymbol{\gamma}$ on a set of problem instances and taking the average of observed performances. For this, all problem instances included in the respective benchmark scenario that have not been held out for subsequent analysis are used. As configurator for our framework we use BO with the lower confidence bound acquisition function [51] with interleaved random configurations every three evaluations[2]. Configuration is repeated three times for each scenario, each running for 60 hours, with different random seeds. To get the overall best configuration, the set of all evaluated configurations $\boldsymbol{\gamma}$ (i.e., the third level optimization archive) is combined into a single data set for each scenario. To estimate the actual best configuration, a common *identification criterion* [52] is used: a surrogate model is fitted on the combined datasets and the optimum among the in-sample predictions of this model is used ($\boldsymbol{\gamma}^{*\textit{lcbench}}$ and $\boldsymbol{\gamma}^{*\textit{rbv2\_super}}$, respectively). We also store the (surrogate-smoothed) optima of all three individual optimization runs and record the range of configuration parameter values to obtain an estimate of the uncertainty of the overall optimal configurations.

The search space used for the optimization of Algorithm 2 is shown in Appendix C, Table V. While the batch size μ is constant in the `equal` *batch_method*, it changes for every bracket when *batch_method* is `HB`. The batch sizes $\mu(2), \mu(3), \ldots$ are constructed from $\mu(1)$ dynamically as described in Section IV. The search space contains several surrogate learners: Random forests [53] (`RF`), K-nearest-neighbors with k set to 1 (`KNN1`), kernelized K-nearest-neighbors with "optimal" weighting [54] (`KKNN7`), and the ratio of density predictions of good and bad points, similar to tree parzen estimators [47] without a hierarchical structure as in BOHB [16] (`TPE`). For the pre-filtering sample distribution $\mathbb{P}_{\boldsymbol{\lambda}}(\mathcal{A})$, we evaluate both uniform sampling (`uniform`), and sampling from the estimated density of good points as done in BOHB [16] (`KDE`). *filter_mb* determines whether the surrogate model makes predictions assuming the highest fidelity value r observed (`TRUE`), as opposed to assuming the fidelity of the points being sampled; in the framework of the SAMPLE Algorithms 3 and 4 in Appendix A, this influences the behavior of $\mathcal{I}_{f_{\text{surr}}}$. Note that the maximum number of fidelity steps per batch s is not part of the search space and instead inferred automatically from η_{fid} and the lower bound for r that is given as part of the optimization problem instance. As in Hyperband, it is set to the largest number of stages that is possible given η_{fid} and the lower bound on r according to Equation (6).

B. Algorithm Analysis

Our goal in this work is not only to determine configurations of Algorithm 2 that perform well on the respective benchmarking scenarios, but also to determine what effect individual components have on performance. However, performing a complete sensitivity analysis would be prohibitively computationally expensive, as it would require evaluation of the objective (i.e., running Algorithm 2) in an experimental design of different configurations. Instead, we evaluate the performance of the candidate configurations found in Section V-A and alternative configurations – which are chosen in a way to allow for answering our research questions – on the benchmark

[2]Note that this optimizer used for third-level optimization is not an instance of Algorithm 2

test instances which were held out during configuration. A simple method to answer many of these questions is to take the optimized configuration of Algorithm 2 and swap components of it for simpler components (or removing them completely), thereby performing a one-factor-at-a-time analysis or an ablation study. However, the optimal values of some components may interact strongly with other components. We therefore auto-configure the framework several times under certain constraints dictated by our particular research question at hand. For example, to investigate the effect of varying n_{trn} and N_{s} over t, we run the optimization of Algorithm 2 with the constraint $n(0)$ to be equal to $n(1)$ and compare the resulting configuration to the overall optimum $\boldsymbol{\gamma}$. Table III lists the different values of $\boldsymbol{\gamma}$ we generate under different constraints. For each value of $\boldsymbol{\gamma}$, we run the respectively configured HPO algorithm on both the *lcbench* and the *rbv2_super* scenario, and (unless stated otherwise) once each for *batch_method* set to `equal` and `HB`. We refer to an optimized configuration that was obtained on the *lcbench* scenario with *batch_method* set to `equal` as $\boldsymbol{\gamma}^{*lcbench}[\texttt{equal}]$, and to the overall optimum (i.e., the better of $\boldsymbol{\gamma}^{*lcbench}[\texttt{equal}]$ and $\boldsymbol{\gamma}^{*lcbench}[\texttt{HB}]$) as $\boldsymbol{\gamma}^{*lcbench}$; similar for *rbv2_super*.

Every evaluation of a framework configuration, i.e., a complete HPO run on a problem instance, is repeated 30 times (with different random seeds) to allow for statistical analysis.

The analysis of our research questions is based on the following tables and visualizations. Table VI in Appendix D shows the configuration parameters that were selected for each benchmark scenario with various search space restrictions. We perform all optimization runs constrained to the fidelity scheduling `equal` and `HB`, respectively, and denote the resulting optimal configurations $\gamma^*[\texttt{equal}]$ and $\gamma^*[\texttt{HB}]$. Figure 2 shows the configuration values of the top 80 evaluated points according to their surrogate-predicted performance. The ranges covered by the bee swarms are again an indicator of approximate ranges of configuration values that can be expected to work well. Figure 5 shows the final performance at $30 \cdot d$ full-budget evaluations for all optimization runs that were performed. The standard error shown is the estimated standard deviation of the mean of benchmark-instance-wise performance, representing uncertainty about the "true" performance mean if an infinite number of benchmark instances of the given class of problems were available.

We now describe in more detail how we operationalize each of the research question RQ1-RQ7 and report results.

RQ1: How does the optimal configuration differ between problem scenarios, i.e., do different problem scenarios benefit from different HPO algorithms?

Setup: We investigate the difference in the values that $\boldsymbol{\gamma}^{*lcbench}$ and $\boldsymbol{\gamma}^{*rbv2\_super}$ take, and put this difference in perspective by comparing it to the uncertainty of these values. To evaluate how well $\boldsymbol{\gamma}^{*lcbench}$ and $\boldsymbol{\gamma}^{*rbv2\_super}$ generalize to other problem scenarios, we evaluate them on the respective instances of scenarios that they were not configured on.

Results: As can be seen in Table VI and in Figure 2, many of the selected components of the $\boldsymbol{\gamma}^*$ are relatively close to each other across the two scenarios on which they were optimized, relative to their uncertainty ranges. $\mathcal{I}_{f_{\text{surr}}}$ is chosen as `KNN1` on *rbv2_super*, but can also use `KKNN7` on *lcbench*, which in fact seems to be slightly preferred. This is interesting as KNN-based models are rarely considered in surrogate-based HPO; the typically preferred random forest model was not selected. $\mathbb{P}_{\boldsymbol{\lambda}}(\mathcal{A})$ takes any of the two values for *rbv2_super*, but is chosen to be `KDE` in *lcbench*. Finally, $\rho(0)$ is close to 1 in the beginning on *rbv2_super*, and closer to 0 (although still greater than $\rho(1)$) for *lcbench*.

The degree to which the differences in $\boldsymbol{\gamma}^*$ influence the outcome can be observed in Figure 5. The optimized results generalize well to test instances from the same scenario as they were configured on. Figure 3 shows the optimization progress (on unseen test instances) of configurations if configured on the same scenario vs. configurations that were configured on a different scenario. We see, for example, a clear advantage of the configurations that we obtained by optimizing directly on *lcbench* when we evaluate them on their respective held out test instances. We suspect that this difference in performance is mainly due to the different choices of surrogate model classes $\mathcal{I}_{f_{\text{surr}}}$ as well as the random interleave fraction ρ (cf. Figure 2), and that specific settings for these two algorithmic components are needed for *lcbench* to reach optimal performance.

This is not the case for the *rbv2_super* scenario, where none of the different algorithms seem to clearly exploit the problem structure of *rbv2_super* better than others.

RQ2: How does the optimized algorithm compare to other established HPO implementations?

Setup: We evaluate several well-known HPO algorithms in their default configuration on the same benchmark instances: for BOHB [16], we use the implementation found in `HpBandSter`[3] (version 0.7.4); for HB [14], we use `mlr3hyperband`[4] (version 0.1.2); for SMAC [5], we use the SMACv3 package[5] (version 1.0.1). We also construct a traditional Gaussian process-based BO (GPBO) [4] with mlrMBO[6] (version 1.1.5). As GPBO works best with numerical search spaces, we only evaluate it on *lcbench*. Note that GPBO, SMAC, and RS are not multi-fidelity algorithms and therefore always evaluate points with maximum fidelity 1.

Results: The performance curves for the mean normalized regret are shown in Figure 3, and the final performance values at $30 \cdot d$ full fidelity evaluations are shown in Figure 5. A critical difference plot and test can be seen in Figure 4b. The behavior of RS, HB, BOHB, and SMAC is not surprising; initially, RS and SMAC perform the same, as SMAC evaluates an initial random design. After this, the performance of SMAC improves quickly. HB and BOHB initially both perform better than RS or SMAC because of their multi-fidelity evaluations, but there is little difference between them. After a while, BOHB starts to outperform HB because of its surrogate-based sampling, which aligns with the observations in [16]. Therefore, BOHB performs well for most budgets, often being the best optimizer for a budget of one as well as for 100 full fidelity evaluations. Given its multi-fidelity characteristics, HB is a good choice for low budgets, while SMAC is well suited for larger optimization budgets. Our framework is very competitive on both *lcbench* and *rbv2_super*, but is outperformed by SMAC on *nb301*. We assume that this is because Algorithm 2 was not explicitly optimized for the *nb301* scenario.

Although our framework was only optimized for performance at $30 \cdot d$ evaluations, it is also competitive with BOHB after fewer evaluations, as seen in Figure 4b.

[3]https://github.com/automl/HpBandSter
[4]https://cran.r-project.org/package=mlr3hyperband
[5]https://github.com/automl/SMAC3
[6]https://cran.r-project.org/package=mlrMBO

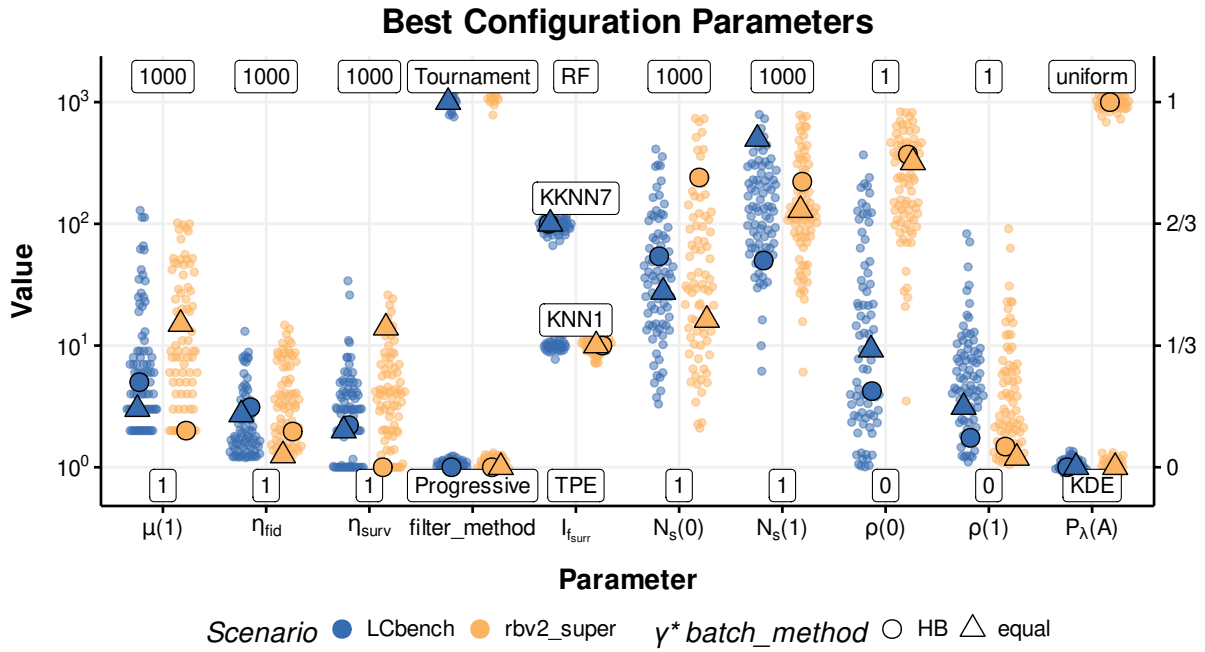


Fig. 2: Beeswarm plot of the best configurations according to the surrogate model over the meta-optimization archive of γ^*. Shown are the top 80 configuration points (according to the surrogate-model-predicted performance) that were evaluated during optimization. Levels of discrete parameters are shown. Most numeric parameters are on a log-scale (left axis), except for $\rho(0)$, $\rho(1)$, which are on a linear scale (right axis). Instead of showing both $N_\text{s}^0(t)$ and $N_\text{s}^1(t)$, their geometric mean $N_\text{s}(t)$ is shown. The highlighted large points are $\gamma^*[\texttt{HB}]$ and $\gamma^*[\texttt{EQUAL}]$, which were found on both benchmark scenarios.

RQ3: Does the successive-halving fidelity schedule have an advantage over the (simpler) equal-batch-size schedule?

Setup: It is likely that the type of fidelity scheduling used interacts with other configuration parameters. Therefore, we investigate the difference of resulting optimal configurations $\gamma^*[\texttt{equal}]$ and $\gamma^*[\texttt{HB}]$.

Results: In both scenarios, the batch method `HB` is ultimately selected for the optimum γ^*, although Figures 4a and 4b show that the difference to batch size `equal` is not statistically significant at $\alpha = 1\%$. We observe that the `equal` fidelity scheduling mode has several advantages: it is much simpler than `HB` as it does not need to keep track of SH brackets, and does not need to adapt $\mu(b)$ to make the expended budget at each bracket approximately equal. As another benefit, it allows for easy parallel scheduling of evaluations (see also Figure 1). This is because it always schedules the same number of function evaluations at a time, which can therefore be run synchronously.

RQ4: What is the effect of using multi-fidelity methods in general?

Setup: We evaluate the performance of a modified γ^* where the number of fidelity stages s is set to to 1, thus ensuring that configurations are only evaluated with maximum fidelity 1.[7]

Results: Our results show the superiority of MF-HPO methods compared to HPO methods that do not make use of lower-fidelity approximations. Figure 4a suggests that multi-fidelity methods are significantly better than their non-multi-fidelity counterparts if optimization is stopped at an intermediate overall budget corresponding to 100 full fidelity evaluations. To be more precise, we see that BOHB as well as both optimized variants $\boldsymbol{\gamma}^*[\texttt{equal}]$ and $\boldsymbol{\gamma}^*[\texttt{HB}]$ (optimized for the respective scenario, respectively) significantly outperform SMAC under this strict budget constraint. In line with [14], HB significantly outperforms RS for this budget. On the other hand, Figure 4b provides evidence that multi-fidelity methods can achieve performance on the same level as state-of-the-art methods that do not make use of low fidelity approximations (e.g., SMAC) for larger budgets. We conclude that a properly designed multi-fidelity mechanism provides substantial improvements of anytime performance without affecting performance for larger budgets negatively. In our opinion, the gain in anytime performance justifies the additional algorithmic complexity that is introduced by multi-fidelity methods.

RQ5a and RQ5b: Does changing SAMPLE configuration parameters throughout the optimization process offer an advantage? Does (more complicated) surrogate-assisted sampling in SAMPLE provide an advantage over using simple random sampling with surrogate filtering?

Setup: To investigate RQ5a (i.e., the effect of the dependence of ρ, n_trn and the N_s configuration parameters on t), we performed an optimization where this t-dependence was removed. As these parameters are interpolated between the values at $t = 0$ and $t = 1$, this corresponds to restricting the search space to where these values are equal, as shown for γ_2 in Table III. In addition to this, we ran another optimization where we further restricted N_s^0 and N_s^1 to be equal,

[7]Because s is not part of the search space Γ and is instead given by Equation 6, this is achieved by setting η_fid to ∞.

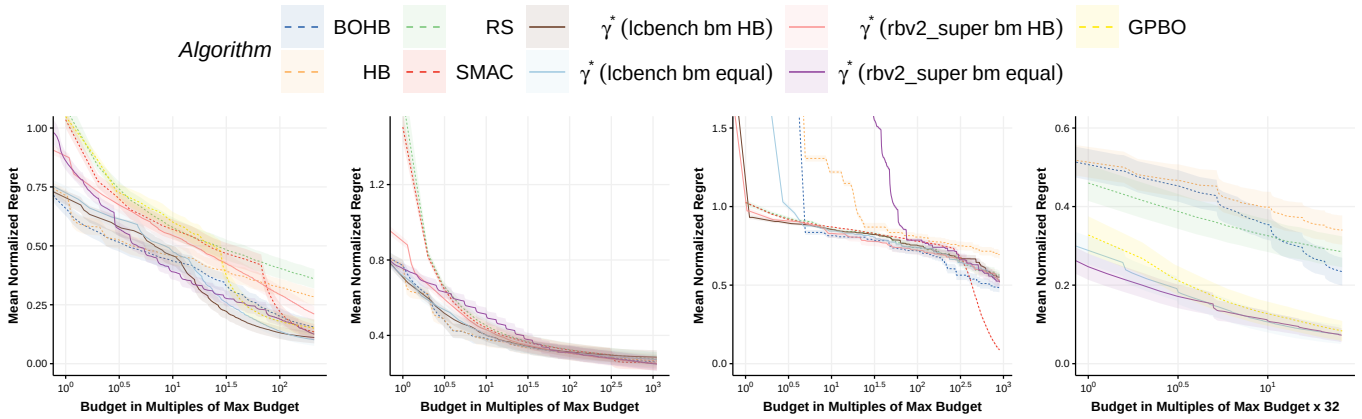
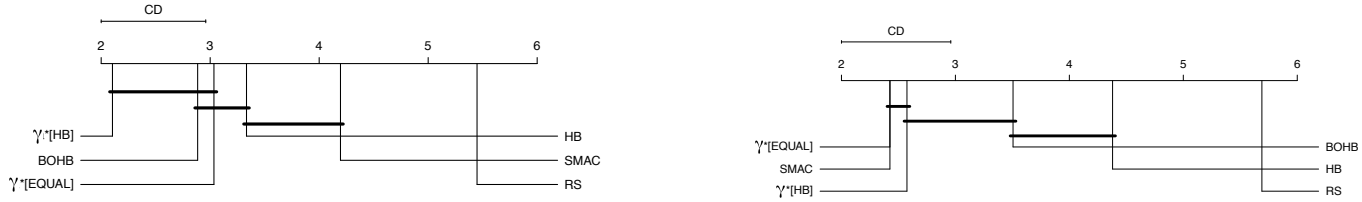


Fig. 3: Optimization progress (mean normalized regret) of serial evaluation on each benchmark scenario as well as 32x parallel evaluation on *lcbench*. Different configurations of Algorithm 2 are executed on benchmark functions that have not been used for the meta-optimization itself, and the progress of these algorithm runs is shown. "γ^*(lcbench bm equal)" is the configuration obtained from optimizing on *lcbench* with *batch_method* `equal`, other labels are contructed similarly. Shown is the mean over 30 evaluations, averaged over all available test benchmark instances for each of the three scenarios. The uncertainty bands show the standard error over the test instances. Note the log-scale on the x-axis. Regret is calculated as the difference between the best evaluation performance so far and the overall best value found on each benchmark instance over all experiments; normalized such that 1 corresponds to the median of the performance of all randomly sampled full fidelity evaluations. We plot performance values observed by the HPO algorithm which depend on evaluation fidelity. This is the reason for the initially "slow" convergence of algorithms that make their first full-fidelity evaluation late. Note that μ of $\gamma^*[\texttt{equal}]$ was set to 32 for the parallel evaluations, and HB and BOHB were only naïvely parallelized to simulate a synchronous "single optimizer, multiple workers" environment. See Figure 6 in Appendix E for a larger version.



(a) Intermediate optimization budget of 100 full evaluations

(b) Full evaluation budget (final performance)

Fig. 4: Critical difference plot [55] comparing the performance of different algorithms across all instances and scenarios. For each of the three scenarios, the mean performance (across replications) for each of the six algorithms is computed ($\boldsymbol{\gamma}^*[\texttt{HB}]$ is equal to $\boldsymbol{\gamma}^{*lcbench}[\texttt{HB}]$ for instances of the lcbench scenario, and to $\boldsymbol{\gamma}^{*rbv}[\texttt{HB}]$ for the *rbv2_super* scenario; same for $\boldsymbol{\gamma}^*[\texttt{EQUAL}]$). The critical difference test is based on the ranks of the algorithms computed per scenario and instance. Lower ranks are better. Horizontal bold bars indicate that there is no significant difference between algorithms ($\alpha = 1\%$). GPBO, which was not evaluated on all scenarios, is not included.

n_{trn} to be 1, and only the `tournament` *filter_method* be used for RQ5b. The performance of the resulting configurations gives an indication of the performance that is lost for the gain in simplicity.

Results: The observations made for γ_2 (forbidding change over time) and γ_3 (forbidding change over time and within each batch) are slightly contradictory. In particular, the *nb301* performance of $\gamma_2^{lcbench}[\texttt{HB}]$ is a visible outlier with regards to optimization performance. There is no obvious explanation from inspecting the configuration parameters of $\gamma_2^{lcbench}[\texttt{HB}]$, but it is possible that it is an accidental "good fit" of configuration parameters to the specific landscape of *nb301*.

On *lcbench* and *rbv2_super*, the impact of restricting the search space is smaller and within the uncertainty of the performance of a single configuration. However, we note that both changing configuration parameters over time and within each batch sample introduces significant complexity to the algorithm; thus we prefer the restricted optimization results over γ^*.

RQ6: What effect do different surrogate models (or using no model at all) have on performance?

Setup: We evaluate the overall result $\boldsymbol{\gamma}^*[\texttt{equal}]$ with $\mathcal{I}_{f_{\text{sur}}}$ set to each of the inducers in the original search space (see Table V). Furthermore, $\boldsymbol{\gamma}^*[\texttt{equal}]$ is evaluated with ρ set to 1 (i.e., all points are sampled randomly from a distribution that may be non-uniform), and finally, with $\rho = 1$ and $\mathbb{P}_{\boldsymbol{\lambda}}(\mathcal{A}) = \texttt{uniform}$ (i.e., all points are sampled completely uniformly at random).

Results: Surprisingly, the simple k-nearest-neighbors algorithm seems to be chosen consistently by the algorithm configuration for both *lcbench* and *rbv2_super* (see Figure 2), either with a value of $k = 1$ or $k = 7$. This result is in line with what we already speculated for RQ1. Our ablation experiments suggest that the performance of the optimizer is on average best when using this surrogate learner, even though the differences do not seem

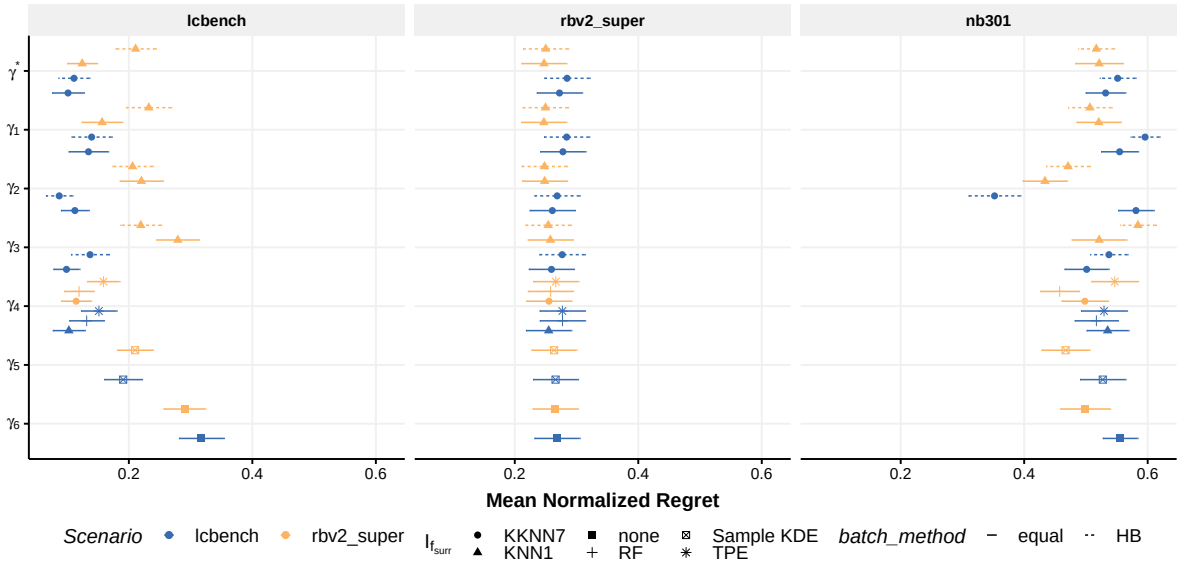


Fig. 5: Mean normalized regret of final performance on "test" benchmark instances for the configuration, shown in Table III. Shown is the mean over 30 evaluations, averaged over all available test benchmark instances for each of the three scenarios. The uncertainty bands show standard error over instance means. Regret is calculated as the difference between the best evaluation performance so far and the overall best value on each benchmark instance over all experiments; normalized such that 1 corresponds to the median of the performance of all randomly sampled full fidelity evaluations.

to be significant. `KNN1` is therefore a reasonable, and simpler, alternative to more complex surrogate learners like the `TPE`-based method proposed for the original BOHB algorithm.

RQ7: Does the equal-batch-size schedule give an advantage over established methods when parallel resources are available?

Setup: Optimization of ML methods that are expensive to evaluate is often done in parallel; we evaluate the performance of our method and other methods in a (simulated) parallel setting. We evaluate γ^*[`equal`] with μ set to 32 and with an optimization budget of $30 \cdot 4 \cdot d$, where d is the dimensionality of the optimization problem. We compare it to GPBO with qLCB [10] for 32 parallel evaluations, and simulate parallel execution of RS by running $30 \cdot 4 \cdot d$ random evaluations. Both BOHB and SMAC offer parallelized versions, but the YAHPO Gym benchmark package does not yet provide support for asynchronous parallel evaluations [50]. However, since HB and BOHB propose evaluations in batches, we compared HB and BOHB by accounting for submitted batches in increments of 32, essentially simulating a single HB/BOHB optimizer sending evaluations to 32 parallel workers and waiting for their completion synchronously.

Results: Figure 3 shows that our algorithm is competitive with GPBO – a state-of-the-art synchronously parallel optimization algorithm – when evaluated with 32 parallel resources. This result also shows the main advantage that the `equal` fidelity schedule has over scheduling like HB, as synchronously parallelizing HB or BOHB puts them at a great disadvantage over even RS. For HB and BOHB, it is necessary to use asynchronously parallelized methods [15], [17] or use an archive shared between multiple workers [16] to obtain competitive results. However, synchronous objective evaluations are much easier to implement in many environments than asynchronous communication between workers, making the advantage of the simplicity of the `equal` schedule even more pronounced.

C. Reproducibility and Open Science

The implementation of the framework in Algorithm 2 and reproducible scripts for the algorithm configuration and analysis are available in public repositories.[8] All data that were generated by our analyses are available as well.

VI. Conclusion

We presented a principled approach and framework to benchmark-driven algorithm design and applied it to generic multi-fidelity HPO. We formalized the search space of multi-fidelity hyperparameter optimizers and created a rich and configurable optimization framework. Given the search space, we used BO for meta-optimization of our framework on two different problem scenarios within the field of AutoML, and evaluated the result on held out test problems and an entirely held out test scenario. We evaluated the configured optimizers and compared to BOHB, HB, SMAC, and a simple RS as reference. We performed an extensive analysis of the effect of different algorithmic components on performance, while also considering the additional algorithmic complexity they introduce. Our configured framework showed equal and in some cases superior performance to widely-used HPO algorithms.

The additional algorithmic complexity introduced by multi-fidelity evaluations provides substantial benefits. However, based on our experiments, we argue that design choices made by established multi-fidelity optimizers like BOHB can be replaced by simpler

[8] https://github.com/mlr-org/smashy,
https://github.com/compstat-lmu/paper_2021_benchmarking_special_issue

TABLE III: Summary of Experiment. Shown are the various optimizer configurations $\boldsymbol{\gamma}$ that were obtained from optimizations with different constraints. "Name": The name by which we refer to the configuration in the text. "RQ": The research question that mainly relates to the configuration. "Optimize": Whether the given configuration was obtained by conducting a (possibly constrained) optimization (✓), or by substituting values into the global optimum $\boldsymbol{\gamma}^*$.

Name	RQ	Optimize	Design Modification
$\boldsymbol{\gamma}^*$	1, 2, 3	✓	none (global optimization)
$\boldsymbol{\gamma}_1$	4	✗	$\eta_{\text{fid}} \to \infty$
$\boldsymbol{\gamma}_2$	5a	✓	$n_{\text{trn}}(0) = n_{\text{trn}}(1),\ N_{\text{s}}^0(0) = N_{\text{s}}^0(1),\ N_{\text{s}}^1(0) = N_{\text{s}}^1(1),\ \rho(0) = \rho(1)$
$\boldsymbol{\gamma}_3$	5b	✓	*filter_method*$\to$ `tournament`, $n_{\text{trn}} \to 1,\ N_{\text{s}}^0(0) = N_{\text{s}}^0(1) = N_{\text{s}}^1(0) = N_{\text{s}}^1(1),\ \rho(0) = \rho(1)$
$\boldsymbol{\gamma}_4$	6	✗	*batch_method*$\to$ `equal`, $\mathcal{I}_{f_{\text{sur}}} \to *$
$\boldsymbol{\gamma}_5$	6	✗	*batch_method*$\to$ `equal`, $\rho \to 0$
$\boldsymbol{\gamma}_6$	6	✗	*batch_method*$\to$ `equal`, $\rho \to 0,\ \mathbb{P}_{\boldsymbol{\lambda}}(\mathcal{A}) \to$ `uniform`
$\boldsymbol{\gamma}_7$	7	✗	*batch_method*$\to$ `equal`, $\mu \to 32$, quadruple budget

choices: For example, the (more complex) SH schedule is not significantly better than a schedule using equal batch sizes, which allows for more efficient parallelization.

KDE-based sampling of points to propose, whether filtered by a surrogate model or not, was consistently chosen by our framework. This detail, which is not usually presented as the main feature of BOHB, seems to have an unexpectedly large impact. On the other hand, our optimization results suggest that a surprisingly simple surrogate learner (`knn`, $k = 1$) can perform even better.

Some components of our search space with large algorithmic complexity have not shown much benefit. Optimization on *rbv2_super* did choose time-varying random interleaving, and overall, more aggressive filtering late during an optimization run ($N_{\text{s}}(1) > N_{\text{s}}(0)$) was slightly favored, but the results did not consistently outperform a configuration obtained from a restricted optimization that excluded time-varying configuration parameters.

Our analysis of the set of best observed performances during optimization indicates that there is a large agreement between benchmark scenarios about what the optimal γ^* configuration should be, with parameters that control (model-based) sampling and the surrogate model being the notable exception. This suggests that there may be a set of configuration parameters that are either generally good for many ML problems, or have little impact on performance and can therefore be set to the simplest value. However, some configuration parameters should be adapted to the properties of the particular given optimization problem. The meta-optimization framework presented in this work can be used in future work to investigate the relationship between features of optimization problems and related optimal configurations.

Other fruitful directions for future work include the more in-depth evaluation of asynchronous evaluations; asynchronous methods are important nowadays where parallel resources are plentiful, but current widely-used surrogate-based benchmarks do not allow for easy asynchronous evaluations. Suggested methods – such as waiting with a sleep-timer for an appropriate amount [16] – are impractical for meta-optimization.

ACKNOWLEDGMENTS

The authors of this work take full responsibilities for its content. Lennart Schneider is supported by the Bavarian Ministry of Economic Affairs, Regional Development and Energy through the Center for Analytics – Data – Applications (ADACenter) within the framework of BAYERN DIGITAL II (20-3410-2-9-8). Lars Kotthoff is supported by NSF grant 1813537. Michel Lang is partly supported by the Research Center Trustworthy Data Science and Security (https://rc-trust.ai).

REFERENCES

[1] B. Bischl, M. Binder, M. Lang, T. Pielok, J. Richter, S. Coors, J. Thomas, T. Ullmann, M. Becker, A. Boulesteix, D. Deng, and M. Lindauer, "Hyperparameter optimization: Foundations, algorithms, best practices and open challenges," *CoRR*, vol. abs/2107.05847, 2021.

[2] L. Kotthoff, C. Thornton, H. H. Hoos, F. Hutter, and K. Leyton-Brown, "Auto-weka: Automatic model selection and hyperparameter optimization in weka," in *Automated Machine Learning: Methods, Systems, Challenges*, F. Hutter, L. Kotthoff, and J. Vanschoren, Eds. Cham: Springer International Publishing, 2019, pp. 81–95.

[3] D. H. Wolpert and W. G. Macready, "No free lunch theorems for optimization," *IEEE Transactions on Evolutionary Computation*, vol. 1, no. 1, pp. 67–82, 1997.

[4] D. R. Jones, M. Schonlau, and W. J. Welch, "Efficient global optimization of expensive black-box functions," *J. Glob. Optim.*, vol. 13, no. 4, pp. 455–492, 1998.

[5] F. Hutter, H. H. Hoos, and K. Leyton-Brown, "Sequential model-based optimization for general algorithm configuration," in *Learning and Intelligent Optimization*, C. A. C. Coello, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 507–523.

[6] K. Swersky, J. Snoek, and R. P. Adams, "Freeze-thaw bayesian optimization," *CoRR*, vol. abs/1406.3896, 2014.

[7] J. Bergstra and Y. Bengio, "Random search for hyper-parameter optimization," *J. Mach. Learn. Res.*, vol. 13, pp. 281–305, 2012.

[8] J. Snoek, H. Larochelle, and R. P. Adams, "Practical bayesian optimization of machine learning algorithms," in *Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 2*, ser. NIPS'12. Red Hook, NY, USA: Curran Associates Inc., 2012, p. 2951–2959.

[9] 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 *NeurIPS 2020 Competition and Demonstration Track, 6-12 December 2020, Virtual Event / Vancouver, BC, Canada*, ser. Proceedings of Machine Learning Research, H. J. Escalante and K. Hofmann, Eds., vol. 133. PMLR, 2020, pp. 3–26.

[10] F. Hutter, H. H. Hoos, and K. Leyton-Brown, "Parallel algorithm configuration," in *Learning and Intelligent Optimization*, ser. Lecture Notes in Computer Science, Y. Hamadi and M. Schoenauer, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, no. 7219, pp. 55–70.

[11] B. Bischl, S. Wessing, N. Bauer, K. Friedrichs, and C. Weihs, "MOI-MBO: multiobjective infill for parallel model-based optimization," in *Learning and Intelligent Optimization - 8th International Conference, Lion 8, Gainesville, FL, USA, February 16-21, 2014. Revised Selected Papers*, ser. Lecture Notes in Computer Science, P. M. Pardalos, M. G. C. Resende, C. Vogiatzis, and J. L. Walteros, Eds., vol. 8426. Springer, 2014, pp. 173–186.

[12] J. González, Z. Dai, P. Hennig, and N. D. Lawrence, "Batch bayesian optimization via local penalization," in *Proceedings of the 19th International Conference on Artificial Intelligence and Statistics, AISTATS 2016, Cadiz, Spain, May 9-11, 2016*, ser. JMLR Workshop and Conference Proceedings, A. Gretton and C. C. Robert, Eds., vol. 51. JMLR.org, 2016, pp. 648–657.

[13] C. Chevalier and D. Ginsbourger, "Fast computation of the multi-points expected improvement with applications in batch selection," in *Learning and Intelligent Optimization*. Springer Berlin Heidelberg, 2013, pp. 59–69.

[14] L. Li, K. G. Jamieson, G. DeSalvo, A. Rostamizadeh, and A. Talwalkar, "Hyperband: A novel bandit-based approach to hyperparameter optimization," *J. Mach. Learn. Res.*, vol. 18, pp. 185:1–185:52, 2017.

[15] L. Li, K. G. Jamieson, A. Rostamizadeh, E. Gonina, J. Ben-tzur, M. Hardt, B. Recht, and A. Talwalkar, "A system for massively parallel hyperparameter tuning," in *Proceedings of Machine Learning and Systems 2020, MLSys 2020, Austin, TX, USA, March 2-4, 2020*, I. S. Dhillon, D. S. Papailiopoulos, and V. Sze, Eds. mlsys.org, 2020.

[16] S. Falkner, A. Klein, and F. Hutter, "BOHB: robust and efficient hyperparameter optimization at scale," in *Proceedings of the 35th International Conference on Machine Learning, ICML 2018, Stockholmsmässan, Stockholm, Sweden, July 10-15, 2018*, ser. Proceedings of Machine Learning Research, J. G. Dy and A. Krause, Eds., vol. 80. PMLR, 2018, pp. 1436–1445.

[17] L. C. Tiao, A. Klein, C. Archambeau, and M. W. Seeger, "Model-based asynchronous hyperparameter optimization," *CoRR*, vol. abs/2003.10865, 2020.

[18] D. Sculley, G. Holt, D. Golovin, E. Davydov, T. Phillips, D. Ebner, V. Chaudhary, M. Young, J. Crespo, and D. Dennison, "Hidden technical debt in machine learning systems," in *Advances in Neural Information Processing Systems 28: Annual Conference on Neural Information Processing Systems 2015, December 7-12, 2015, Montreal, Quebec, Canada*, C. Cortes, N. D. Lawrence, D. D. Lee, M. Sugiyama, and R. Garnett, Eds., 2015, pp. 2503–2511.

[19] K. G. Jamieson and A. Talwalkar, "Non-stochastic best arm identification and hyperparameter optimization," in *Proceedings of the 19th International Conference on Artificial Intelligence and Statistics, AISTATS 2016, Cadiz, Spain, May 9-11, 2016*, ser. JMLR Workshop and Conference Proceedings, A. Gretton and C. C. Robert, Eds., vol. 51. JMLR.org, 2016, pp. 240–248.

[20] H. H. Hoos, "Programming by Optimization," *Communications of the Association for Computing Machinery (CACM)*, vol. 55, no. 2, pp. 70–80, Feb. 2012.

[21] S. Minton, "Automatically Configuring Constraint Satisfaction Programs: A Case Study," *Constraints*, vol. 1, pp. 7–43, 1996.

[22] S. J. Westfold and D. R. Smith, "Synthesis of Efficient Constraint Satisfaction Programs," *Knowl. Eng. Rev.*, vol. 16, no. 1, pp. 69–84, 2001.

[23] D. Balasubramaniam, L. de Silva, C. A. Jefferson, L. Kotthoff, I. Miguel, and P. Nightingale, "Dominion: An Architecture-driven Approach to Generating Efficient Constraint Solvers," in *9th Working IEEE/IFIP Conference on Software Architecture*, Jun. 2011, pp. 228–231.

[24] A. R. KhudaBukhsh, L. Xu, H. H. Hoos, and K. Leyton-Brown, "SATenstein: automatically building local search SAT solvers from components," in *Proceedings of the 21st International Joint Conference on Artifical Intelligence*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2009, pp. 517–524.

[25] J.-N. Monette, Y. Deville, and P. van Hentenryck, "Aeon: Synthesizing Scheduling Algorithms from High-Level Models," in *Operations Research and Cyber-Infrastructure*, 2009, pp. 43–59.

[26] M. López-Ibáñez and T. Stützle, "The automatic design of multi-objective ant colony optimization algorithms," *IEEE Transactions on Evolutionary Computation*, vol. 16, no. 6, pp. 861–875, 2012.

[27] L. C. T. Bezerra, M. López-Ibáñez, and T. Stützle, "Automatic Component-Wise Design of Multiobjective Evolutionary Algorithms," *IEEE Transactions on Evolutionary Computation*, vol. 20, no. 3, pp. 403–417, 2016.

[28] M. Birattari, Z. Yuan, P. Balaprakash, and T. Stützle, "F-race and iterated f-race: An overview," *Experimental methods for the analysis of optimization algorithms*, pp. 311–336, 2010.

[29] M. López-Ibáñez, J. Dubois-Lacoste, L. P. Cáceres, M. Birattari, and T. Stützle, "The irace package: Iterated racing for automatic algorithm configuration," *Operations Research Perspectives*, vol. 3, pp. 43–58, 2016.

[30] N. Dang, L. P. Cáceres, P. D. Causmaecker, and T. Stützle, "Configuring irace using surrogate configuration benchmarks," in *Proceedings of the Genetic and Evolutionary Computation Conference, GECCO 2017, Berlin, Germany, July 15-19, 2017*, P. A. N. Bosman, Ed. ACM, 2017, pp. 243–250.

[31] F. Hutter, H. H. Hoos, K. Leyton-Brown, and T. Stützle, "Paramils: an automatic algorithm configuration framework," *Journal of Artificial Intelligence Research*, vol. 36, pp. 267–306, 2009.

[32] O. Maron and A. W. Moore, "The racing algorithm: Model selection for lazy learners," *Artif. Intell. Rev.*, vol. 11, no. 1-5, pp. 193–225, 1997. [Online]. Available: https://doi.org/10.1023/A:1006556606079

[33] S. van Rijn, H. Wang, M. van Leeuwen, and T. Bäck, "Evolving the structure of Evolution Strategies," in *IEEE Symposium Series on Computational Intelligence (SSCI)*, 2016, pp. 1–8.

[34] G. Malkomes and R. Garnett, "Automating bayesian optimization with bayesian optimization," *Advances in Neural Information Processing Systems*, vol. 31, pp. 5984–5994, 2018.

[35] M. Lindauer, M. Feurer, K. Eggensperger, A. Biedenkapp, and F. Hutter, "Towards assessing the impact of Bayesian optimization's own hyperparameters," 2019.

[36] 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," 2021.

[37] A. Saltelli, "Sensitivity analysis for importance assessment," *Risk Analysis*, vol. 22, no. 3, pp. 579–590, 2002.

[38] W. Hoeffding, "A Class of Statistics with Asymptotically Normal Distribution," *The Annals of Mathematical Statistics*, vol. 19, no. 3, pp. 293 – 325, 1948.

[39] F. Hutter, H. Hoos, and K. Leyton-Brown, "An efficient approach for assessing hyperparameter importance," in *Proceedings of the 31st International Conference on Machine Learning*, ser. Proceedings of Machine Learning Research, E. P. Xing and T. Jebara, Eds., vol. 32, no. 1. Bejing, China: PMLR, 22–24 Jun 2014, pp. 754–762.

[40] C. Fawcett and H. H. Hoos, "Analysing differences between algorithm configurations through ablation," *J. Heuristics*, vol. 22, no. 4, pp. 431–458, 2016.

[41] S. Sheikholeslami, M. Meister, T. Wang, A. H. Payberah, V. Vlassov, and J. Dowling, "Autoablation: Automated parallel ablation studies for deep learning," in *EuroMLSys@EuroSys 2021, Proceedings of the 1st Workshop on Machine Learning and Systemsg Virtual Event, Edinburgh, Scotland, UK, 26 April, 2021*, E. Yoneki and P. Patras, Eds. ACM, 2021, pp. 55–61.

[42] M. López-Ibáñez and T. Stützle, "An experimental analysis of design choices of multi-objective ant colony optimization algorithms," *Swarm Intelligence*, vol. 6, pp. 207–232, 2012.

[43] J. de Nobel, D. Vermetten, H. Wang, C. Doerr, and T. Bäck, "Tuning as a Means of Assessing the Benefits of New Ideas in Interplay with Existing Algorithmic Modules," in *Genetic and Evolutionary Computation Conference Companion*. Association for Computing Machinery, 2021, pp. 1375–1384.

[44] A. Klein, L. C. Tiao, T. Lienart, C. Archambeau, and M. Seeger, "Model-based asynchronous hyperparameter and neural architecture search," *arXiv preprint arXiv:2003.10865*, 2020.

[45] M. Birattari, *Tuning Metaheuristics - A Machine Learning Perspective*, ser. Studies in Computational Intelligence. Springer, 2009, vol. 197.
[46] Z. Karnin, T. Koren, and O. Somekh, "Almost optimal exploration in multi-armed bandits," in *International Conference on Machine Learning*. PMLR, 2013, pp. 1238–1246.
[47] J. Bergstra, R. Bardenet, Y. Bengio, and B. Kégl, "Algorithms for hyper-parameter optimization," *Advances in neural information processing systems*, vol. 24, 2011.
[48] A. Klein, S. Falkner, S. Bartels, P. Hennig, and F. Hutter, "Fast bayesian optimization of machine learning hyperparameters on large datasets," in *Artificial Intelligence and Statistics*. PMLR, 2017, pp. 528–536.
[49] D. Golovin, B. Solnik, S. Moitra, G. Kochanski, J. Karro, and D. Sculley, "Google vizier: A service for black-box optimization," in *Proceedings of the 23rd ACM SIGKDD international conference on knowledge discovery and data mining*, 2017, pp. 1487–1495.
[50] F. Pfisterer, L. Schneider, J. Moosbauer, M. Binder, and B. Bischl, "YAHPO Gym – Design Criteria and a new Multifidelity Benchmark for Hyperparameter Optimization," *arXiv:2109.03670 [cs, stat]*, 2021, arXiv: 2109.03670.
[51] D. R. Jones, "A taxonomy of global optimization methods based on response surfaces," *Journal of Global Optimization*, vol. 21, no. 4, pp. 345–383, 2001.
[52] H. Jalali, I. Van Nieuwenhuyse, and V. Picheny, "Comparison of kriging-based algorithms for simulation optimization with heterogeneous noise," *European Journal of Operational Research*, vol. 261, no. 1, pp. 279–301, 2017.
[53] L. Breiman, "Random forests," *Mach. Learn.*, vol. 45, no. 1, pp. 5–32, 2001.
[54] R. J. Samworth, "Optimal weighted nearest neighbour classifiers," *The Annals of Statistics*, vol. 40, no. 5, Oct. 2012.
[55] J. Demsar, "Statistical comparisons of classifiers over multiple data sets," *J. Mach. Learn. Res.*, vol. 7, pp. 1–30, 2006.

APPENDIX A
SAMPLE ALGORITHMS

The pseudocode of both SAMPLE algorithms is presented here for clarity.

Algorithm Algorithm 3, SAMPLETOURNAMENT, diversifies the set of points proposed through an extension that draws different points $\{\boldsymbol{\lambda}_S\}$ in n *tournaments* at each invocation of SAMPLE. Each tournament yields the top n_{trn} points out of $n_{\text{trn}} \cdot N_{\text{s}}^{(i)}$ samples according to the surrogate model, where $i \in \{1, \ldots, n\}$. We parameterize the number of points sampled for each tournament using the configuration parameters N_{s}^0 and N_{s}^1; the effective value of N_{s} for each point is interpolated geometrically[9]: $N_{\text{s}}^{(i)} = \left\lfloor (N_{\text{s}}^0)^{(n-i)/(n-1)} \cdot (N_{\text{s}}^1)^{(i-1)/(n-1)} \right\rceil$. The special case of $n_{\text{trn}} = 1$ corresponds to a basic SAMPLE subroutine where points $\boldsymbol{\lambda}_S^{(i)}$ are each independently filtered with different effective filter factors $N_{\text{s}}^{(i)}$.

Besides the sampling method described above, we propose an alternative method, Algorithm 4, which we name SAMPLEPROGRESSIVE: instead of sampling $N_{\text{s}}^{(i)}$ points independently for each configuration $\boldsymbol{\lambda}_S^{(i)}$ with $i \in \{1, \ldots, \mu\}$, we sample a single ordered pool $\mathcal{P}$ of $\mu \cdot \max(N_{\text{s}}^0, N_{\text{s}}^1)$ random points once at the beginning of SAMPLE. Each $\boldsymbol{\lambda}_S^{(i)}$ is then selected as the point with the best surrogate-predicted performance from the first $\mu \cdot N_{\text{s}}^{(i)}$ points in $\mathcal{P}$ that was not already selected before.

Algorithm 3 SAMPLETOURNAMENT algorithm

Input: Archive $\mathcal{A}$, number of points to generate μ, current fidelity r

Configuration Parameters: Surrogate learner $\mathcal{I}_{f_{\text{sur}}}$, generating distribution $\mathbb{P}_{\boldsymbol{\lambda}}(\mathcal{A})$, random interleave fraction ρ, sample filtering rates $(N_{\text{s}}^0, N_{\text{s}}^1)$, points to sample per tournament round n_{trn}.

State Variables: Batch of proposed configurations $C \leftarrow \emptyset$

Use ρ to decide how many points $n_{\text{random\_interleave}}$ to sample without filter
$C \leftarrow$ Sample $n_{\text{random\_interleave}}$ points from $\mathbb{P}_{\boldsymbol{\lambda}}(\mathcal{A})$
$n \leftarrow \lceil (\mu - n_{\text{random\_interleave}})/n_{\text{trn}} \rceil$ ▷ Numter of tournament rounds
$f_{\text{sur}} \leftarrow \mathcal{I}_{f_{\text{sur}}}(\mathcal{A})$ ▷ Surrogate model
for $i \leftarrow 1$ **to** n **do**
    $n_{\text{sample}} \leftarrow \left\lfloor (N_{\text{s}}^0)^{\frac{n-i}{n-1}} \cdot (N_{\text{s}}^1)^{\frac{i-1}{n-1}} \right\rceil$
    $C_0 \leftarrow$ Sample n_{sample} configurations from $\mathbb{P}_{\boldsymbol{\lambda}}(\mathcal{A})$
    Predict performances of points in C_0 using f_{sur}
    $C \leftarrow C \cup \text{SELECT\_TOP}\,(C_0, \min(n_{\text{trn}}, \mu - |C|))$
end for
return C

[9]Here, $\lfloor \cdot \rceil$ is the operation that rounds to the nearest integer

Algorithm 4 SAMPLEPROGRESSIVE algorithm

Input: Surrogate learner $\mathcal{I}_{f_{\text{sur}}}$, Archive $\mathcal{A}$, number of points to generate μ, current fidelity r, random interleave fraction ρ, sample filtering rates $(N_{\text{s}}^0, N_{\text{s}}^1)$, generating distribution $\mathbb{P}_{\boldsymbol{\lambda}}(\mathcal{A})$

State Variables: Batch of proposed configurations $C \leftarrow \emptyset$, (ordered) pool of sampled points to select from $\mathcal{P}$

Use ρ to decide how many points $n_{\text{random\_interleave}}$ to sample without filter
$C \leftarrow$ Sample $n_{\text{random\_interleave}}$ configurations from $\mathbb{P}_{\boldsymbol{\lambda}}(\mathcal{A})$
$\mu \leftarrow \mu - n_{\text{random\_interleave}}$
$n_{\text{pool}} \leftarrow \mu \cdot \max(N_{\text{s}}^0, N_{\text{s}}^1)$
$\mathcal{P} \leftarrow$ Sample n_{pool} configurations from $\mathbb{P}_{\boldsymbol{\lambda}}(\mathcal{A})$
$f_{\text{sur}} \leftarrow \mathcal{I}_{f_{\text{sur}}}(\mathcal{A})$ ▷ Surrogate model
Predict performances of points in $\mathcal{P}$ using f_{sur}
for $i \leftarrow 1$ **to** μ **do**
    $n_{\text{options}} \leftarrow \left\lfloor (N_{\text{s}}^0)^{\frac{\mu-i}{\mu-1}} \cdot (N_{\text{s}}^1)^{\frac{\mu-1}{\mu-1}} \right\rceil$
    $\mathcal{P}_{\text{options}} \leftarrow$ first n_{options} elements of $\mathcal{P}$
    $S \leftarrow \text{SELECT\_TOP}\,(\mathcal{P}_{\text{options}}, 1)$
    $C \leftarrow C \cup S$
    $\mathcal{P} \leftarrow \mathcal{P} - S$
end for
return C

APPENDIX B
BENCHMARK COLLECTIONS

While the underlying data for lcbench and nb301 have been previously used in publications ([1], [2]), rbv2_super is a novel task that has not been investigated previously in literature.

Benchmarks in the YAHPO Gym are implemented as surrogate model benchmarks, where a Wide & Deep [3] neural network was fitted to a set of pre-evaluated performance values of hyperparameter configurations.

HPO on a neural network (*lcbench* [1]): The first set of problems covers HPO on a relatively small and numeric search space. The neural network (more precisely, a funnel-shaped multilayer perceptron) that is tuned has a total of 7 numerical hyperparameters. The fidelity of an evaluation can be controlled by setting the number of epochs over which the neural network is trained. The instances belonging to this scenario represent HPO performed on 35 different classification tasks taken from OpenML [4]. As a target metric, we choose the cross entropy loss on the validation set.

AutoML pipeline configuration (*rbv2_super* [5]): Second, we investigate the problem of configuring an AutoML pipeline. Here, a learning algorithm must be selected first from the following candidates: approximate k nearest neighbors [6], elastic net linear models [7], random forests [8], decision trees [9], support vector machines [10], and gradient boosting [11]. The hyperparameters of each learner are chosen conditioned on this learner being active, i.e., there are hierarchical hyperparameter dependencies. The fidelity of a single evaluation can be controlled by choosing the size of the training data set that is used to train the respective learner. The automated optimization of the pipeline is performed for 89 different classification tasks [5], again taken from OpenML. As a target metric, we opt for the log loss.

Neural architecture search (*nb301* [2]): As third problem scenario, we consider neural architecture search. The search space of architectures is given by the darts search space [12], and architectures were trained and evaluated on CIFAR-10 [13]. A convolutional neural network is constructed by stacking so-called normal and reduction cells that each can be represented as a directed acyclic graph consisting of an ordered sequence of vertices (nodes) resembling feature maps, with each directed edge associated with an operation that transforms the input node. A tabular representation can be derived using 34 categorical parameters with 24 dependencies. Each architecture can be trained for 1 to 98 epochs, allowing again for lower fidelity evaluations. The target metric is validation accuracy.

APPENDIX C
META-OPTIMIZATION SEARCH SPACE

The full optimization space used for optimization of γ^* is presented here in Table V. Other γ results have the restrictions applied to them, as shown in Table III in Section V.

TABLE IV: Instances within the benchmarking scenarios *lcbench*, *rbv2_super*, and *nb301* within the YAHPO Gym test suite that have been used for the experimental analysis (Section V). We show the instances that have been used for optimization only (Section V-A), and the instances that have been held out from optimization and exclusively used for analysis (Section V-B).

Scenario	Instances used for configuration	Instances held out for analysis
lcbench	3945, 7593, 126026, 167201, 168329, 168868, 168908, 189354	34539 126025, 126029, 146212, 167083, 167104, 167149, 167152, 167161, 167168, 167181, 167184, 167185, 167190, 167200, 168330, 168331, 168335, 168910, 189862, 189865, 189866, 189873, 189905, 189906, 189908, 189909
rbv2_super	1050, 1053, 1056, 1068, 12, 1461, 1464, 1489, 1510, 1515, 188, 3, 307, 32, 37, 375, 38, 40496, 40498, 40701, 40978 40979, 40983, 41142, 41146, 41156, 41157, 458, 46, 6332,	42, 44, 4534, 4538, 469, 470, 50, 54, 60, 1040, 1049, 1063, 1067, 11, 1111, 14, 1462, 1468, 1475, 1476, 1478, 1479, 1480, 1485, 1486, 1487, 1494, 1497, 15, 1501, 16, 18, 181, 182, 22, 23, 23381, 24, 28, 29, 31, 312, 334, 377, 40499, 40536, 40670, 40900, 40966, 40975, 40981, 40982, 40984, 40994, 41138, 41143, 41212, 4134, 4154
nb301	–	1

TABLE V: Meta-optimization search space used to configure Algorithm 2. Some configuration parameters are optimized on a non-linear *scale*, meaning e.g. the optimizer optimizes a value of $\log \mu(1)$ ranging from $\log 2$ to $\log 200$.

Parameter	Meaning	Range	Scale
$\mu(1)$	(first bracket) batch size	$\{2, \dots, 200\}$	$\log \mu(1)$
batch_method	batch method	{`equal`, `HB`}	
η_{fid}	fidelity rate	$[2^{1/4}, 2^4]$	$\log \log \eta_{\text{fid}}$
η_{surv}	survival rate	$[1, \infty)$	$1/\eta_{\text{surv}}$
filter_method	SAMPLE method	{SAMPLETOURNAMENT, SAMPLEPROGRESSIVE}	
$\mathbb{P}_{\boldsymbol{\lambda}}(\mathcal{A})$	SAMPLE generating distribution	{`uniform`, `KDE`}	
$\mathcal{I}_{f_{\text{surr}}}$	surrogate learner	{`KNN1`, `KKNN7`, `TPE`, `RF`}	
$n_{\text{trn}}(0)$	filter sample per tournament round at $t = 0$	$\{1, \dots, 10\}$	$\log n_{\text{trn}}(0)$
$n_{\text{trn}}(1)$	filter sample per tournament round at $t = 1$	$\{1, \dots, 10\}$	$\log n_{\text{trn}}(1)$
$N_{\text{s}}^0(0)$	filtering rate of first point in batch at $t = 0$	$[1, 1000]$	$\log N_{\text{s}}^0(1)$
$N_{\text{s}}^0(1)$	filtering rate of first point in batch at $t = 1$	$[1, 1000]$	$\log N_{\text{s}}^0(1)$
$N_{\text{s}}^1(0)$	filtering rate of last point in batch at $t = 0$	$[1, 1000]$	$\log N_{\text{s}}^1(1)$
$N_{\text{s}}^1(1)$	filtering rate of last point in batch at $t = 1$	$[1, 1000]$	$\log N_{\text{s}}^1(1)$
$\rho(0)$	random interleave fraction at $t = 0$	$[0, 1]$	
$\rho(1)$	random interleave fraction at $t = 1$	$[0, 1]$	
filter_mb	surrogate prediction always with maximum r	{`TRUE`, `FALSE`}	
ρ_{random}	random interleave the same number in every batch	{`TRUE`, `FALSE`}	

APPENDIX D
ALL γ-VALUES

A table of all optimization γ-values is given in Table VI.

TABLE VI: Optimized configuration parameters, under some constraints. Top: restricted to *batch_method* `HB`, bottom: `equal`. γ_2, γ_3 are further restricted, as described in Table III in Section V. Shown is the overall result. Square brackets show range (for numeric parameters) or list (for discrete parameters) of values found in individual optimization runs when not aggregated as a rough indicator of uncertainty. "(!)" indicates the parameter was forced to the value by a restriction. The "(evals)" row indicates the number of performance evaluations (i.e. full second level optimization runs) that were performed in each setting.

Parameter Scenario	γ^* lcbench	γ^* rbv2_super	γ_2 lcbench	γ_2 rbv2_super	γ_3 lcbench	γ_3 rbv2_super
Optimized with *batch_method* `HB`:						
$\mu(1)$	5 [5, 23]	2 [2, 52]	126 [10, 126]	8 [8, 114]	5 [3, 68]	2 [2, 52]
η_{fid}	3.11 [1.25, 3.11]	1.97 [1.97, 6.73]	2.19 [1.68, 10.2]	4.4 [2.04, 4.4]	14.6 [1.45, 14.6]	5.19 [2.24, 5.19]
η_{surv}	2.22 [2.22, 6.1]	6.09 [1.65, 6.09]	3.42 [2.58, 9.19]	3.26 [3.26, 5]	1.15 [1.15, 3.07]	1.20 [1.03, 1.62]
filter_method	PROG [TRN]	PROG [TRN]	PROG [TRN]	PROG [TRN]	TRN (!)	TRN (!)
$\mathbb{P}_{\boldsymbol{\lambda}}(\mathcal{A})$	`KDE`	`uniform` [`KDE`]	`KDE`	`uniform` [`KDE`]	`KDE`	`uniform`
$\mathcal{I}_{f_{\text{surr}}}$	`KKNN7` [`KNN1`]	`KNN1`	`KKNN7` [`KNN1`]	`KNN1`	`KKNN7` [`KNN1`]	`KNN1`
$n_{\text{trn}}(0)$	2 [2, 8]	2 [1, 5]	5 [1, 8]	5 [1, 6]	1 (!)	1 (!)
$n_{\text{trn}}(1)$	1 [1, 5]	5 [1, 5]				
$N_{\text{s}}^0(0)$	101 [9.19, 124]	226 [2.03, 226]	39.6 [10.5, 76.7]	125 [125, 163]	570 [73, 570]	155 [155, 561]
$N_{\text{s}}^0(1)$	312 [56.3, 817]	495 [57.1, 533]				
$N_{\text{s}}^1(0)$	28.9 [4.84, 144]	256 [7.19, 256]	31.4 [18.2, 74.6]	481 [480, 563]		
$N_{\text{s}}^1(1)$	8 [8, 654]	99.7 [46.4, 890]				
$\rho(0)$	0.21 [0.12, 0.85]	0.86 [0.68, 0.86]	0.37 [0.2, 0.49]	0.71 [0.49, 0.71]	0.38 [0.12, 0.54]	0.34 [0.34, 0.45]
$\rho(1)$	0.08 [0.08, 0.55]	0.06 [0.01, 0.25]				
filter_mb	TRUE [FALSE]	FALSE [TRUE]	TRUE	TRUE	TRUE [FALSE]	FALSE
ρ_{random}	FALSE [TRUE]	TRUE [FALSE]	FALSE [TRUE]	TRUE [FALSE]	TRUE	TRUE [FALSE]
(evals)	1495	332	1071	249	981	337
Optimized with *batch_method* `equal`:						
$\mu(1)$	3 [2, 4]	15 [11, 15]	5 [2, 7]	5 [2, 5]	2 [2, 6]	85 [3, 93]
η_{fid}	2.71 [1.77, 12.2]	1.25 [1.22, 1.43]	2.63 [2.27, 6.01]	8 [1.28, 8]	2.59 [1.46, 2.59]	2.3 [1.36, 12.7]
η_{surv}	2.5 [1.23, 3.36]	18.8 [8.74, 18.8]	1.87 [1.84, 5.49]	3.45 [3.45, 5.53]	3.53 [1.29, 4.86]	6.5 [5.34, 11.3]
filter_method	TRN [PROG]	PROG	TRN [PROG]	PROG [TRN]	TRN (!)	TRN (!)
$\mathbb{P}_{\boldsymbol{\lambda}}(\mathcal{A})$	`KDE`	`KDE` [`uniform`]	`KDE`	`uniform` [`KDE`]	`KDE`	`uniform` [`KDE`]
$\mathcal{I}_{f_{\text{surr}}}$	`KKNN7` [`KNN1`]	`KNN1`	`KNN1` [`KNN7`]	`KNN1`	`KNN1` [`KNN7`]	`KNN1`
$n_{\text{trn}}(0)$	1 [1, 4]	2 [1, 3]	5 [1, 5]	3 [1, 3]	1 (!)	1 (!)
$n_{\text{trn}}(1)$	2 [1, 2]	9 [1, 9]				
$N_{\text{s}}^0(0)$	21.5 [1.63, 309]	39.5 [2.42, 39.5]	169 [43.4, 191]	212 [49.5, 212]	81.3 [24.8, 111]	583 [295, 777]
$N_{\text{s}}^0(1)$	941 [58.2, 991]	18.1 [11.5, 408]				
$N_{\text{s}}^1(0)$	35.4 [7.8, 280]	6.65 [5.43, 391]	4.76 [2.34, 273]	1.71 [1.71, 4.21]		
$N_{\text{s}}^1(1)$	264 [5, 474]	925 [25.4, 925]				
$\rho(0)$	0.32 [0.09, 0.68]	0.83 [0.49, 0.83]	0.34 [0.09, 0.37]	0.34 [0.34, 0.53]	0.27 [0.03, 0.27]	0.96 [0.38, 0.96]
$\rho(1)$	0.16 [0.06, 0.29]	0.03 [0.03, 0.5]				
filter_mb	TRUE	TRUE	TRUE	TRUE	TRUE [FALSE]	TRUE
ρ_{random}	TRUE [FALSE]	FALSE	TRUE [FALSE]	FALSE [TRUE]	TRUE	TRUE [FALSE]
(evals)	1751	450	1437	341	1070	368

APPENDIX E
ENLARGED OPTIMIZATION CURVE PLOTS

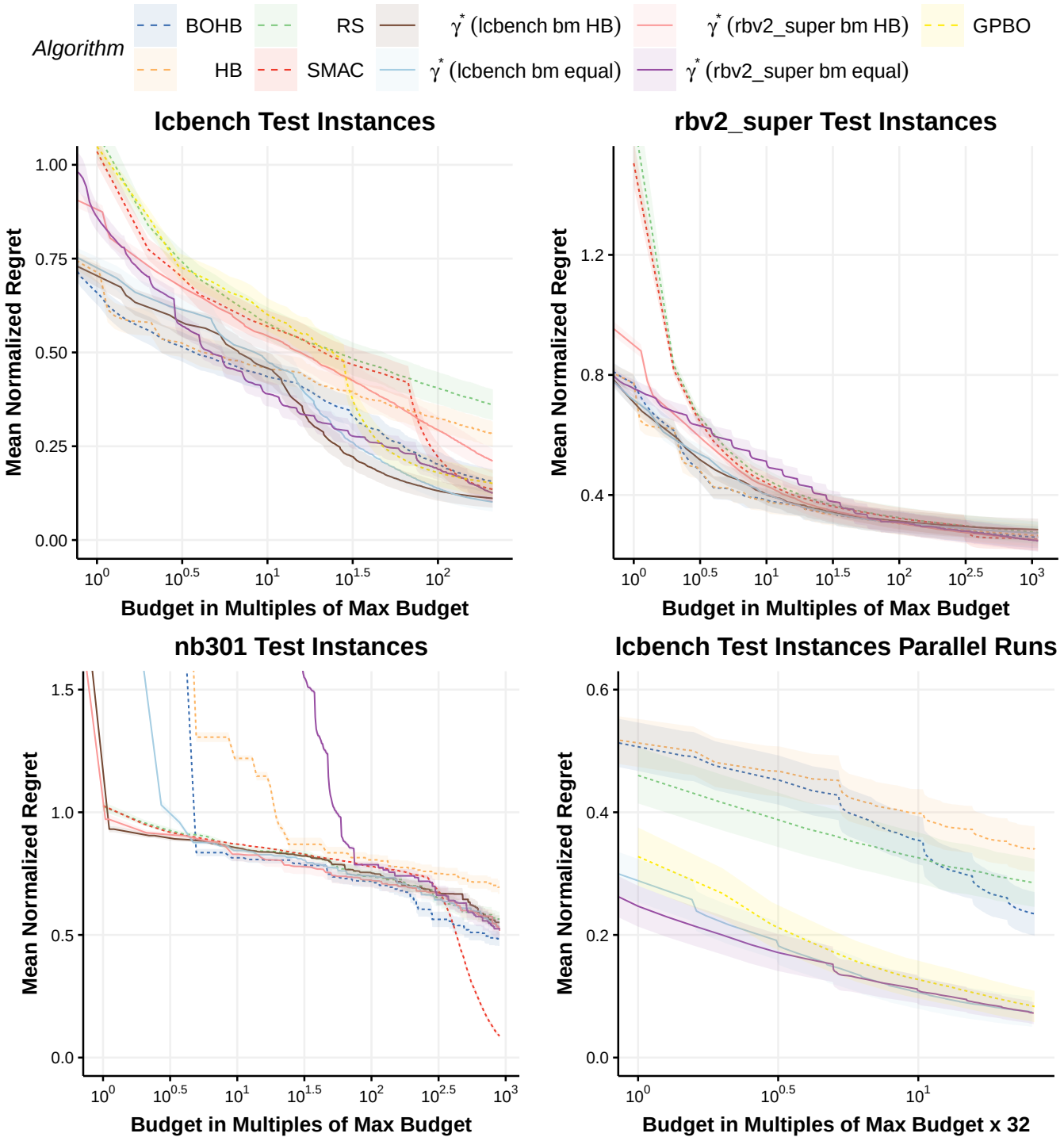


Fig. 6: Enlarged version of Figure 3.

REFERENCES

[1] L. Zimmer, M. Lindauer, and F. Hutter, "Auto-pytorch tabular: Multi-fidelity metalearning for efficient and robust autodl," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 43, no. 9, pp. 3079 – 3090, 2021.

[2] J. Siems, L. Zimmer, A. Zela, J. Lukasik, M. Keuper, and F. Hutter, "Nas-bench-301 and the case for surrogate benchmarks for neural architecture search," 2020.

[3] H.-T. Cheng, L. Koc, J. Harmsen, T. Shaked, T. Chandra, H. Aradhye, G. Anderson, G. Corrado, W. Chai, M. Ispir *et al.*, "Wide & deep learning for recommender systems," in *Proceedings of the 1st workshop on deep learning for recommender systems*, 2016, pp. 7–10.

[4] J. Vanschoren, J. N. van Rijn, B. Bischl, and L. Torgo, "Openml: networked science in machine learning," *SIGKDD Explor.*, vol. 15, no. 2, pp. 49–60, 2013.

[5] M. Binder, F. Pfisterer, and B. Bischl, "Collecting empirical data about hyperparameters for data driven automl," in *Proceedings of the 7th ICML Workshop on Automated Machine Learning (AutoML 2020)*, 2020.

[6] Y. A. Malkov and D. A. Yashunin, "Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs," *IEEE transactions on pattern analysis and machine intelligence*, vol. 42, no. 4, pp. 824–836, 2018.

[7] J. Friedman, T. Hastie, and R. Tibshirani, "Regularization paths for generalized linear models via coordinate descent," *Journal of Statistical Software*, vol. 33, no. 1, pp. 1–22, 2010.

[8] M. N. Wright and A. Ziegler, "ranger: A fast implementation of random forests for high dimensional data in C++ and R," *Journal of Statistical Software*, vol. 77, no. 1, pp. 1–17, 2017.

[9] L. Breiman, J. H. Friedman, R. A. Olshen, and C. J. Stone, *Classification And Regression Trees*. Routledge, Oct. 2017.

[10] B. E. Boser, I. Guyon, and V. Vapnik, "A training algorithm for optimal margin classifiers," in *Proceedings of the Fifth Annual ACM Conference on Computational Learning Theory, COLT 1992, Pittsburgh, PA, USA, July 27-29, 1992*, D. Haussler, Ed. ACM, 1992, pp. 144–152.

[11] T. Chen and C. Guestrin, "XGBoost: A scalable tree boosting system," in *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ser. KDD '16. New York, NY, USA: ACM, 2016, pp. 785–794.

[12] H. Liu, K. Simonyan, and Y. Yang, "DARTS: Differentiable Architecture Search," in *Proceedings of the International Conference on Learning Representations*, 2019.

[13] A. Krizhevsky, "Learning multiple layers of features from tiny images," University of Toronto, Tech. Rep., 2009.