Received: 9 July 2021 | Revised: 28 August 2022 | Accepted: 28 November 2022

DOI: 10.1002/widm.1484

ADVANCED REVIEW

WIREs DATA MINING AND KNOWLEDGE DISCOVERY WILEY

Hyperparameter optimization: Foundations, algorithms, best practices, and open challenges

Bernd Bischl[1,2] | **Martin Binder**[1,2] | **Michel Lang**[2,3] | **Tobias Pielok**[1] | **Jakob Richter**[1,4] | **Stefan Coors**[1] | **Janek Thomas**[1] | **Theresa Ullmann**[2,5] | **Marc Becker**[1] | **Anne-Laure Boulesteix**[2,5] | **Difan Deng**[6] | **Marius Lindauer**[6]

[1]Department of Statistics, Ludwig-Maximilians-Universität München, Munich, Germany

[2]Munich Center for Machine Learning, Munich, Germany

[3]Research Center Trustworthy Data Science and Security, TU Dortmund University, Dortmund, Germany

[4]Department of Statistics, TU Dortmund University, Dortmund, Germany

[5]Institute for Medical Information Processing, Biometry and Epidemiology, Ludwig-Maximilians-Universität München, Munich, Germany

[6]Institute of Artificial Intelligence, Leibniz University Hannover, Germany

Correspondence
Bernd Bischl, Department of Statistics, Ludwig-Maximilians-Universität München, Munich, Germany.
Email: bernd.bischl@stat.uni-muenchen.de

Martin Binder, Department of Statistics, Ludwig-Maximilians-Universität München, Munich, Germany.
Email: martin.binder@stat.uni-muenchen.de

Funding information
Bavarian Ministry for Economic Affairs, Infrastructure, Transport and Technology, Grant/Award Number: BAYERN DIGITAL II; Bundesministerium für Bildung und Forschung, Grant/Award Number: 01IS18036A; Deutsche Forschungsgemeinschaft (Collaborative Research Center), Grant/Award Number: SFB 876-A3; Federal Statistical Office of Germany; Research Center "Trustworthy Data Science and Security"

Edited by: Justin Wang, Associate Editor and Witold Pedrycz, Editor-in-Chief

[Correction added on 17 February 2023, after first online publication: The co-corresponding author has been added in this version.]

Abstract

Most machine learning algorithms are configured by a set of hyperparameters whose values must be carefully chosen and which often considerably impact performance. To avoid a time-consuming and irreproducible manual process of trial-and-error to find well-performing hyperparameter configurations, various automatic hyperparameter optimization (HPO) methods—for example, based on resampling error estimation for supervised machine learning—can be employed. After introducing HPO from a general perspective, this paper reviews important HPO methods, from simple techniques such as grid or random search to more advanced methods like evolution strategies, Bayesian optimization, Hyperband, and racing. This work gives practical recommendations regarding important choices to be made when conducting HPO, including the HPO algorithms themselves, performance evaluation, how to combine HPO with machine learning pipelines, runtime improvements, and parallelization.

This article is categorized under:

Algorithmic Development > Statistics

Technologies > Machine Learning

Technologies > Prediction

KEYWORDS

automl, hyperparameter optimization, machine learning, model selection, tuning

1 | INTRODUCTION

Machine learning (ML) algorithms are highly configurable by their hyperparameters (HPs). These parameters often substantially influence the complexity, behavior, speed as well as other aspects of the learner, and their values must be selected with care in order to achieve optimal performance. Human trial-and-error to select these values is time-consuming, often somewhat biased, error-prone and computationally irreproducible.

As the mathematical formalization of hyperparameter optimization (HPO) is essentially black-box optimization, often in a higher-dimensional space, this is better delegated to appropriate algorithms and machines to increase efficiency and ensure reproducibility. Many HPO methods have been developed to assist in and automate the search for well-performing hyperparameter configuration (HPCs) over the last 20–30 years. However, more sophisticated HPO approaches in particular are not as widely used as they could (or should) be in practice. We postulate that the reason for this may be a combination of the following factors:

- Poor understanding of HPO methods by potential users, who may perceive them as (too) complex "black-boxes";
- Poor confidence of potential users in the superiority of HPO methods over trivial approaches and resulting skepticism of the expected return on (time) investment;
- Missing guidance on the choice and configuration of HPO methods for the problem at hand;
- Difficulty to define the search space of an HPO process appropriately.

With these obstacles in mind, this paper formally and algorithmically introduces HPO, with many hints for practical application. Our target audience are scientists and users with a basic knowledge of ML and evaluation. To the extent possible, this article tries to provide practical advice on how to perform HPO depending on the circumstances, particularly in Section 6. There are, however, many open questions in the field of HPO, and a definite recommendation can often not be made. We therefore attempt to present the reader with the most widely used options for various aspects of HPO that are available to them, and we also present the challenges that remain. Both will help with making decisions that depend on the specific problems that the reader may be facing.

In this article, we mainly discuss HP for supervised ML, which is arguably the default scenario for HPO. We mainly do this to keep notation simple and to not overwhelm less experienced readers, especially for less experienced readers. Nevertheless, all covered techniques can be applied to practically any algorithm in ML in which the algorithm is trained on a collection of instances and performance is quantitatively measurable—for example, in semi-supervised learning, reinforcement learning, and potentially even unsupervised learning.[1]

Subsequent sections of this paper are organized as follows: Section 2 discusses related work. Section 3 introduces the concept of supervised ML and discusses the evaluation of ML algorithms. The principle of HPO is introduced in Section 4. Major classes of HPO methods are described, including their strengths and limitations. The problem of overtuning, the handling of noise in the context of HPO, and the topic of threshold tuning are also addressed. Section 5 introduces the most common preprocessing steps and the concept of ML pipelines, which enables us to include preprocessing and model selection within HPO. Section 6 offers practical recommendations on how to choose resampling strategies as well as define tuning search spaces, provides guidance on which HPO algorithm to use, and describes how HPO can be parallelized. In Section 7, we also briefly discuss how HPO directly connects to a much broader field of algorithm selection and configuration beyond ML and other related fields. Section 8 concludes with a discussion of relevant open issues in HPO.

2 | RELATED WORK

As one of the most studied sub-fields of automated ML (AutoML), there exist several previous surveys on HPO. Feurer and Hutter (2019) offered a thorough overview about existing HPO approaches, open challenges, and future research directions. In contrast to our paper, however, that work does not focus on specific advice for issues that arise in practice. Yang and Shami (2020) provide a very high-level overview of search spaces, HPO techniques, and tools. Although we expect that the paper by Yang and Shami (2020) will be a more accessible paper for first-time users of HPO compared with the survey by Feurer and Hutter (2019), it does not explain HPO's mathematical and algorithmic details or practical tips on how to apply HPO efficiently. Last but not least, Andonie (2019) provides an overview about HPO methods, but with a focus on computational complexity aspects. We see this work described here as filling the gap between these

papers by providing all necessary details both for first-time users of HPO as well as experts in ML and data science who seek to understand the concepts of HPO in sufficient depth.

Our focus is on providing a general overview of HPO without a special focus on concrete ML model classes. However, since the ML field has many large sub-communities by now, there are also several specialized HPO and AutoML surveys. For example, He et al. (2021) focus on AutoML for deep learning models, Khalid and Javaid (2020) on HPO for forecasting models in smart grids, and Zhang et al. (2021) on AutoML on graph models.

3 | SUPERVISED MACHINE LEARNING

3.1 | Terminology and notations

Supervised ML addresses the problem of inferring a model from labeled training data that is then used to predict data from the same underlying distribution with minimal error. Let $\mathcal{D}$ be a labeled data set with observations, where each observation $\left(\mathbf{x}^{(i)}, y^{(i)}\right)$ consists of a p-dimensional feature vector[2] $\mathbf{x}^{(i)} \in \mathcal{X}$ and its label $y^{(i)} \in \mathcal{Y}$. Hence, we define the data set[3] $\mathcal{D} = \left(\left(\mathbf{x}^{(1)},\ y^{(1)}\right), ..., \left(\mathbf{x}^{(n)},\ y^{(n)}\right)\right)$. We assume that $\mathcal{D}$ has been sampled i.i.d. from an underlying, unknown distribution, so $\mathcal{D} \sim \left(\mathrm{P}_{xy}\right)^n$. Each dimension of a p-dimensional $\mathbf{x}$ will usually be of numerical, integer, or categorical type. While some ML algorithms can handle all of these data types natively (e.g., tree-based methods), others can only work on numeric features and require encoding techniques for categorical types. The most common supervised ML tasks are *regression* and *classification*, where $\mathcal{Y} = \mathrm{R}$ for regression and $\mathcal{Y}$ is finite and categorical for classification with $|\mathcal{Y}| = g$ classes. Although we mainly discuss HPO in the context of regression and classification, all covered concepts easily generalize to other supervised ML tasks, such as Poisson regression, survival analysis, cost-sensitive classification, multi-output tasks, and many more. An ML model is a function $f : \mathcal{X} \to \mathrm{R}^g$ that assigns a prediction in R^g to a feature vector from $\mathcal{X}$. For regression, g is 1, while in classification the output represents the g *decision scores* or posterior probabilities of the g candidate classes. Binary classification is usually simplified to $g = 1$, with a single decision score in R or only the posterior probability for the positive class. The function space—usually parameterized—to which a model belongs is called the *hypothesis space* and denoted as $\mathcal{H}$.

The goal of supervised ML is to fit a model given n observations sampled from P_{xy}, so that it generalizes well to new observations from the same data generating process. Formally, an ML learner or *inducer* $\mathcal{I}$ configured by HPs $\lambda \in \Lambda$ maps a data set $\mathcal{D}$ to a model $\widehat{f}$ or equivalently to its associated parameter vector $\widehat{\theta}$, that is,

$$\mathcal{I} : \mathrm{D} \times \boldsymbol{\Lambda} \to \mathcal{H},\ \ (\mathcal{D}, \lambda) \mapsto \widehat{f}, \tag{1}$$

where $\mathrm{D} := \cup_{n \in N} (\mathcal{X} \times \mathcal{Y})^n$ is the set of all data sets. While model parameters $\widehat{\theta}$ are an output of the learner $\mathcal{I}$, HPs λ are an input. We also write $\mathcal{I}_\lambda$ for $\mathcal{I}$ or $\widehat{f}_{\mathcal{D},\lambda}$ for $\widehat{f}$ if we want to stress that the inducer was configured with λ or that the model was learned on $\mathcal{D}$ by an inducer configured by. A loss function $L : \mathcal{Y} \times \mathrm{R}^g \to \mathrm{R}_0^+$ measures the discrepancy between the prediction and the true label. Many ML learners use the concept of *empirical risk minimization* (ERM) in their training routine to produce their fitted model $\widehat{f}$, that is, they optimize $\mathcal{R}_{\text{emp}}(f)$ or $\mathcal{R}_{\text{emp}}(\theta)$ over all candidate models $f \in \mathcal{H}$

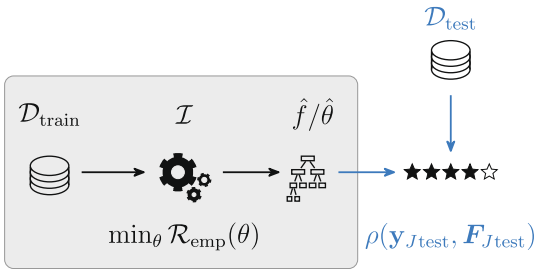


FIGURE 1 Learner $\mathcal{I}$ takes input data, performs ERM, and returns model $\widehat{f}$ and its parameters $\widehat{\theta}$. The GE of $\widehat{f}$ is evaluated on the fresh test set $\mathcal{D}_{\text{test}}$.

$$\mathcal{R}_{\text{emp}}(f) := \sum_{i=1}^{n} L\left(y^{(i)}, f\left(\mathbf{x}^{(i)}\right)\right), \quad \widehat{f} = \arg\min_f \mathcal{R}_{\text{emp}}(f) \tag{2}$$

on the training data $\mathcal{D}$ (c.f. Figure 1). This empirical risk is only a stochastic proxy for what we are actually interested in, namely the theoretical risk or true generalization error $\mathcal{R}(f) := \mathrm{E}_{(\mathbf{x},\, y)\sim \mathrm{P}_{xy}}[L(y,\ f(\mathbf{x}))]$. For many complex hypothesis spaces, $\mathcal{R}_{\text{emp}}(f)$ can become considerably smaller than its true risk $\mathcal{R}(f)$. This phenomenon is known as overfitting, which in ML is usually addressed by either constraining the hypothesis space or regularized risk minimization, that is, adding a complexity penalty $J(\theta)$ to (2).

3.2 | Evaluation of ML algorithms

After training an ML model $\widehat{f}$, a natural follow-up step is to evaluate its future performance given new, unseen data. We seek to use an unbiased, high-quality statistical estimator, which numerically quantifies the performance of our model when it is used to predict the target variable for new observations drawn from the same data-generating process P_{xy}.

3.2.1 | Performance metrics

A general performance measure ρ for an arbitrary data set of size m is defined as a two-argument function that maps the m-size vector of true labels $\mathbf{y}$ and the $m \times g$ matrix of prediction scores $\boldsymbol{F}$ to a scalar performance value:

$$\rho : \bigcup_{m \in \mathrm{N}} (\mathcal{Y}^m \times \mathrm{R}^{m \times g}) \to \mathrm{R}, \quad (\mathbf{y}, \boldsymbol{F}) \mapsto \rho(\mathbf{y}, \boldsymbol{F}). \tag{3}$$

This more general set-based definition is needed for performance measures—such as area under the ROC curve (AUC)—or for most measures from survival time analysis, where loss values cannot be computed with respect to only a single observation. For usual point-wise losses $L(y, f(\mathbf{x}))$, we can simply extend L to ρ by averaging over the size-m set used for testing:

$$\rho_L(\mathbf{y}, \boldsymbol{F}) = \frac{1}{m} \sum_{i=1}^{m} L\left(y^{(i)}, \boldsymbol{F}^{(i)}\right), \tag{4}$$

where $\boldsymbol{F}^{(i)}$ is the ith row of $\boldsymbol{F}$; this corresponds to estimating the theoretical risk $\mathcal{R}(f)$ corresponding to the given loss.

Furthermore, the introduction of ρ allows the evaluation of a learner with respect to a different performance metric than the loss used for risk minimization. Because of this, we call the loss used in (2) *inner loss*, and ρ the outer performance measure or *outer loss*.[4] Both can coincide, but quite often we select an outer performance measure based on the prediction task we would like to solve, and opt to approximate this metric with a computationally cheaper and possibly differentiable version during inner risk minimization.

3.2.2 | Generalization error

Due to potential overfitting, every predictive model should be evaluated on unseen test data to ensure unbiased performance estimation. Assuming (for now) dedicated train and test data sets $\mathcal{D}_{\text{train}}$ and $\mathcal{D}_{\text{test}}$ of sizes n_{train} and n_{test}, respectively, we define the *generalization error* of a learner $\mathcal{I}$ with HPs λ trained on n_{train} observations, w.r.t. measure ρ as:

$$\mathrm{GE}(\mathcal{I}, \lambda, n_{\text{train}}, \rho) := \lim_{n_{\text{test}} \to \infty} \mathrm{E}_{\mathcal{D}_{\text{train}}, \mathcal{D}_{\text{test}} \sim \mathrm{P}_{xy}} \left[\rho\left(\mathbf{y}_{\text{test}},\ \boldsymbol{F}_{\mathcal{D}_{\text{test}}, \mathcal{I}(\mathcal{D}_{\text{train}},\, \lambda)}\right)\right], \tag{5}$$

where we take the expectation over the data sets $\mathcal{D}_{\text{train}}$ and $\mathcal{D}_{\text{test}}$, both i.i.d. from P_{xy}, and $\boldsymbol{F}_{\mathcal{D}_{\text{test}},\mathcal{I}(\mathcal{D}_{\text{train}},\,\lambda)}$ is the matrix of predictions when the model is trained on $\mathcal{D}_{\text{train}}$ and predicts on $\mathcal{D}_{\text{test}}$. Note that in the simpler and common case of a point-wise loss $L(y, f(\mathbf{x}))$, the above trivially reduces to the more common form

$$\text{GE}(\mathcal{I},\,\lambda,\,n_{\text{train}},\,\rho_L) = \text{E}_{\mathcal{D}_{\text{train}},(\mathbf{x},\,y)\sim\text{P}_{xy}}[L(y,\,\mathcal{I}_\lambda(\mathcal{D}_{\text{train}})(\mathbf{x}))] \tag{6}$$

with expectation over data set $\mathcal{D}_{\text{train}}$ and test sample $(\mathbf{x}, y)$, both independently sampled from P_{xy}. This corresponds to the expectation of $\mathcal{R}(f)$—which references a given, fixed model—over all possible models fitted to different realizations of $\mathcal{D}_{\text{train}}$ of size n_{train} (see Figure 1).

3.2.3 | Data splitting and resampling

The generalization error must usually be estimated from a single given data set $\mathcal{D}$. For a simple estimator based on a single random split, $\mathcal{D}_{\text{train}}$ and $\mathcal{D}_{\text{test}}$ can be represented as index vectors $J_{\text{train}} \in \{1, ..., n\}^{n_{\text{train}}}$ and $J_{\text{test}} \in \{1, ..., n\}^{n_{\text{test}}}$, which usually partition the data set. For an index vector J of length m, one can define the corresponding vector of labels $\mathbf{y}_J \in \mathcal{Y}^m$, and the corresponding matrix of prediction scores $\boldsymbol{F}_{J,f} \in \text{R}^{m\times g}$ for a model f. The *holdout estimator* is then:

$$\widehat{\text{GE}}_{J_{\text{train}},J_{\text{test}}}(\mathcal{I},\,\lambda,\,n_{\text{train}},\,\rho) = \rho\left(\mathbf{y}_{J_{\text{test}}},\,\boldsymbol{F}_{J_{\text{test}},\mathcal{I}(\mathcal{D}_{\text{train}},\,\lambda)}\right). \tag{7}$$

The holdout approach has the following trade-off: (i) Because n_{train} must be smaller than n, the estimator is a pessimistically biased estimator of $\text{GE}(\mathcal{I},\,\lambda,\,n,\,\rho)$, as we do not use all available data for training. In a certain sense, we are estimating with respect to the wrong training set size. (ii) If $\mathcal{D}_{\text{train}}$ is large, $\mathcal{D}_{\text{test}}$ will be small, and the estimator (7) has a large variance. This trade-off not only depends on relative sizes of n_{train} and n_{test}, but also the absolute number of observations, so the complete sample size n,

Resampling methods offer a partial solution to this dilemma. These methods repeatedly split the available data into training and test sets, then apply an estimator (7) for each of these, and finally aggregate over all obtained ρ performance values. Formally, we can identify a resampling strategy with a vector of corresponding splits, that is, $\mathcal{J} = ((J_{\text{train},1},\,J_{\text{test},1}), ..., (J_{\text{train},B}, J_{\text{test},B}))$, where $J_{\text{train},i}$,$J_{\text{test},i}$ are index vectors and B is the number of splits. Hence, the estimator for Equation (5) is:

$$\begin{aligned}\widehat{\text{GE}}(\mathcal{I},\,\mathcal{J},\,\rho,\,\lambda) &= \text{agr}\left(\widehat{\text{GE}}_{J_{\text{train},1},J_{\text{test},1}}(\mathcal{I},\;\lambda,\;|J_{\text{train},1}|,\;\rho),\;...,\;\widehat{\text{GE}}_{J_{\text{train},B},J_{\text{test},B}}(\mathcal{I},\;\lambda,\;|J_{\text{train},B}|,\;\rho)\right)\\ &= \text{agr}\left(\rho\left(\mathbf{y}_{J_{\text{test},1}},\;\boldsymbol{F}_{J_{\text{test},1},\mathcal{I}(\mathcal{D}_{\text{train},1},\,\lambda)}\right),\;...,\;\rho\left(\mathbf{y}_{J_{\text{test},B}},\;\boldsymbol{F}_{J_{\text{test},B},\mathcal{I}(\mathcal{D}_{\text{train},B},\,\lambda)}\right)\right),\end{aligned} \tag{8}$$

where the aggregator agr is often chosen to be the mean. For Equation (8) to be a valid estimator of Equation (6), we must specify to what n_{train} training set size an estimator refers in $\text{GE}(\mathcal{I},\,\lambda,\,n_{\text{train}},\,\rho)$. As the training set sizes can be different during resampling (they usually do not vary much), it should at least hold that $n_{\text{train}} \approx n_{\text{train},1} \approx ... \approx n_{\text{train},B}$, and we could take the average for such a required reference size with $n_{\text{train}} = \frac{1}{B}\sum_{j=1}^{B} n_{\text{train},j}$.

Resampling uses the data more efficiently than a single holdout split, as the repeated estimation and averaging over multiple splits results in an estimate of generalization error with lower variance (Kohavi, 1995; Simon, 2007). Additionally, the pessimistic bias of simple holdout is also kept to a minimum and can be reduced to nearly 0 by choosing training sets of size close to n. The most widely used resampling technique is arguably k-fold-cross-validation (CV), which partitions the available data in k subsets of approximately equal size, and uses each partition to evaluate a model fitted on its complement. For small data sets, it makes sense to repeat CV with multiple random partitions and to average the resulting estimates in order to average out the variability, which results in repeated k-fold-cross-validation. Furthermore, note that performance values generated from resampling splits and especially CV splits are not statistically independent because of their overlapping training sets, so the variance of $\widehat{\text{GE}}(\mathcal{I},\,\mathcal{J},\,\rho,\,\lambda)$ is not proportional to $1/B$. Somewhat paradoxically, a leave-one-out strategy is not the optimal choice, and repeated cross-validation with many (but fewer than n) folds and many repetitions is often a better choice (Bengio & Grandvalet, 2004). An overview of existing resampling techniques can be found in Bischl et al. (2012) or Boulesteix et al. (2008).

Resampling is a crucial aspect of HPO, as it is the cost function for which hyperparameter are optimized, as shown in the upcoming Equation (10). In Section 6.1, we therefore give practical advice on how to choose resampling methods for the task at hand, possibly using HPO-methods that dynamically allocate resampling folds. Furthermore, in Section 4.4, we discuss that nested resampling is an integral part of the validation of HPO systems that ensures unbiased estimation of future predictive performance.

4 | HYPERPARAMETER OPTIMIZATION

4.1 | HPO problem definition

Most learners are highly configurable by HPs, and their generalization performance usually depends on this configuration in a non-trivial and subtle way. HPO algorithms automatically identify a well-performing HPC $\lambda \in \widetilde{\Lambda}$ for an ML algorithm $\mathcal{I}_\lambda$. The search space $\widetilde{\Lambda} \subset \Lambda$ contains all considered HPs for optimization and their respective ranges:

$$\widetilde{\Lambda} = \widetilde{\Lambda}_1 \times \widetilde{\Lambda}_2 \times \ldots \times \widetilde{\Lambda}_l, \tag{9}$$

where $\widetilde{\Lambda}_i$ is a bounded subset of the domain of the ith HP Λ_i, and can be either continuous, discrete, or categorical. This already mixed search space can also contain *dependent HPs*, leading to a hierarchical search space: An HP λ_i is said to be *conditional* on λ_j if λ_i is only active when λ_j is an element of a given subset of Λ_j and inactive otherwise, that is, not affecting the resulting learner (Thornton et al., 2013). Common examples are kernel HPs of a kernelized machine such as the SVM, when we tune over the kernel type and its respective hyperparameters as well. Such conditional HPs usually introduce tree-like dependencies in the search space, and may in general lead to dependencies that may be represented by directed acyclic graphs.

The general HPO problem as visualized in Figure 2 is defined as:

$$\lambda^* \in \arg\min_{\lambda \in \widetilde{\Lambda}} c(\lambda) = \arg\min_{\lambda \in \widetilde{\Lambda}} \widehat{\mathrm{GE}}(\mathcal{I}, \mathcal{J}, \rho, \lambda) \tag{10}$$

where λ^* denotes the theoretical optimum, and $c(\lambda)$ is a shorthand for the estimated generalization error when $\mathcal{I}$, $\mathcal{J}$, ρ are fixed. We therefore estimate and optimize the generalization error $\widehat{\mathrm{GE}}(\mathcal{I}, \mathcal{J}, \rho, \lambda)$ of a learner $\mathcal{I}_\lambda$, w.r.t. an HPC λ, based on a resampling split $\mathcal{J} = ((J_{\text{train},1}, J_{\text{test},1}), \ldots, (J_{\text{train},B}, J_{\text{test},B}))$.[5] Note that $c(\lambda)$ is a black-box, as it usually has no closed-form mathematical representation, and hence no analytic gradient information is available. Furthermore, the

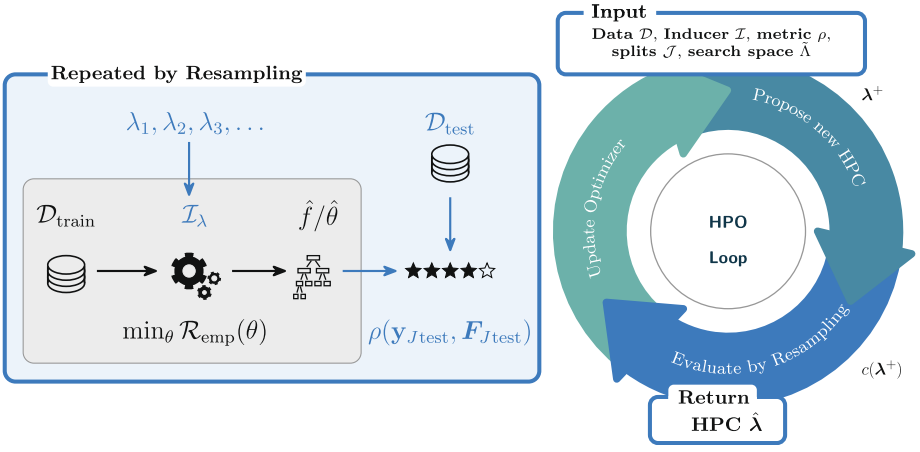


FIGURE 2 General HPO loop with inner risk minimization.

evaluation of $c(\lambda)$ can take a significant amount of time. Therefore, the minimization of $c(\lambda)$ forms an *expensive black-box* optimization problem.

Taken together, these properties define an optimization problem of considerable difficulty. Furthermore, they rule out many popular optimization methods that require gradients or entirely numerical search spaces or that must perform a large number of evaluations to converge to a well-performing solution, like many meta-heuristics. Furthermore, as $c(\lambda) = \widehat{\mathrm{GE}}(\mathcal{I}, \mathcal{J}, \rho, \lambda)$, which is defined via resampling and evaluates λ on randomly chosen validation sets, c should be considered a stochastic objective—although many HPO algorithms may ignore this fact or simply handle it by assuming that we average out the randomness through enough resampling replications.

We can thus define the HP tuner $\tau : \left(\mathcal{D}, \mathcal{I}, \widetilde{\Lambda}, \rho\right) \mapsto \widehat{\lambda}$ that proposes its estimate $\widehat{\lambda}$ of the true optimal configuration λ^* given a dataset $\mathcal{D}$, an inducer $\mathcal{I}$ with corresponding search space $\widetilde{\Lambda}$ to optimize, and a target measure ρ. The specific resampling splits $\mathcal{J}$ used can either be passed into τ as well or are internally handled to facilitate adaptive splitting or multi-fidelity optimization (e.g., as done in Klein et al., 2017).

4.2 | Well-established HPO algorithms

All HPO algorithms presented here work by the same principle: they iteratively *propose* HPCs λ^+ and then *evaluate* the performance on these configurations. We store these HPs and their respective evaluations successively in the so-called *archive* $\mathcal{A} = \left(\left(\lambda^{(1)}, c\left(\lambda^{(1)}\right)\right), \left(\lambda^{(2)}, c\left(\lambda^{(2)}\right)\right), \ldots\right)$, with $\mathcal{A}^{[t+1]} = \mathcal{A}^{[t]} \cup (\lambda^+, c(\lambda^+))$ if a single point is proposed by the tuner.

Many algorithms can be characterized by how they handle two different trade-offs: (a) The exploration versus exploitation trade-off refers to how much budget an optimizer spends on either attempting to directly exploit the currently available knowledge base by evaluating very close to the currently best candidates (e.g., local search) or exploring the search space to gather new knowledge (e.g., random search). (b) The inference versus 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*, that is, how many configurations a tuner can (reasonably) propose at the same time; *global* versus *local* behavior of the optimizer, that is, if updates are always quite close to already evaluated configurations; *noise handling*, that is, if the optimizer takes into account that the estimated generalization error is noisy; *multifidelity*, that is, if the tuner uses cheaper evaluations, for example on smaller subsets of the data, to infer performance on the full data; *search space complexity*, that is, if and how hierarchical search spaces as introduced in Section 5 can be handled.

4.2.1 | Grid search and random search

Grid search (GS) is the process of discretizing the range of each HP and exhaustively evaluating every combination of values. Numeric and integer HP values are usually equidistantly spaced in their box constraints. The number of distinct values per HP is called the *resolution* of the grid. For categorical HPs, either a subset or all possible values are considered. A second simple HPO algorithm is *random search* (RS). In its simplest form, values for each HP are drawn independently of each other

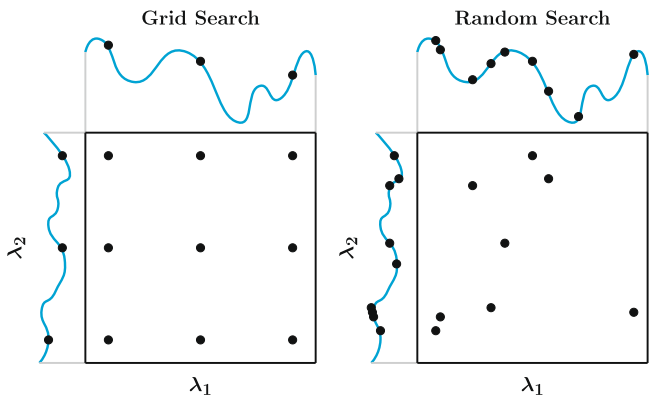


FIGURE 3 RS and GS where only HP λ_1 has a strong influence on c (figure based on Bergstra and Bengio (2012)).

and from a pre-specified (often uniform) distribution, which works for (box-constrained) numeric, integer, or categorical parameters (c.f. Figure 3). Due to their simplicity, both GS and RS can handle hierarchical search spaces.

RS often has much better performance than GS in higher-dimensional HPO settings (Bergstra & Bengio, 2012). GS suffers directly from the *curse of dimensionality* (Bellman, 2015), as the required number of evaluations increases exponentially with the number of HPs for a fixed grid resolution. This seems to be true as well for RS at first glance, and we certainly require an exponential number of points in $\dim\left(\widetilde{\Lambda}\right)$ to cover the space well. However, in practice, HPO problems often have *low effective dimensionality* (Bergstra & Bengio, 2012): The set of HPs that have an influence on performance is often a small subset of all available HPs. Consider the example illustrated in Figure 3, where an HPO problem with HPs λ_1 and λ_2 is shown. A GS with resolution 3 resulting in 9 HPCs is evaluated, and we discover that only HP λ_1 has any relevant influence on the performance, so only three of nine evaluations provided any meaningful information. In comparison, RS would have given us nine different configurations for HP λ_1, which results in a higher chance of finding the optimum. Another advantage of RS is that it can easily be extended by further samples; in contrast, the number of points on a grid must be specified beforehand, and refining the resolution of GS afterwards is more complicated. Altogether, this makes RS preferable to GS and a surprisingly strong baseline for HPO in many practical settings. Notably, there are sampling methods that attempt to cover the search space more evenly than the uniform sampling of RS, for example, *Latin Hypercube Sampling* (McKay et al., 1979), or Sobol sequences (Antonov & Saleev, 1979). However, these do not seem to significantly outperform naive i.i.d. sampling (Bergstra & Bengio, 2012).

4.2.2 | Evolution strategies

Evolution strategies (ES) are a class of stochastic population-based optimization methods inspired by the concepts of biological evolution, belonging to the larger class of *evolutionary algorithms*. They do not require gradients, making them generally applicable in black-box settings such as HPO. In ES terminology, an *individual* is a single HPC, the

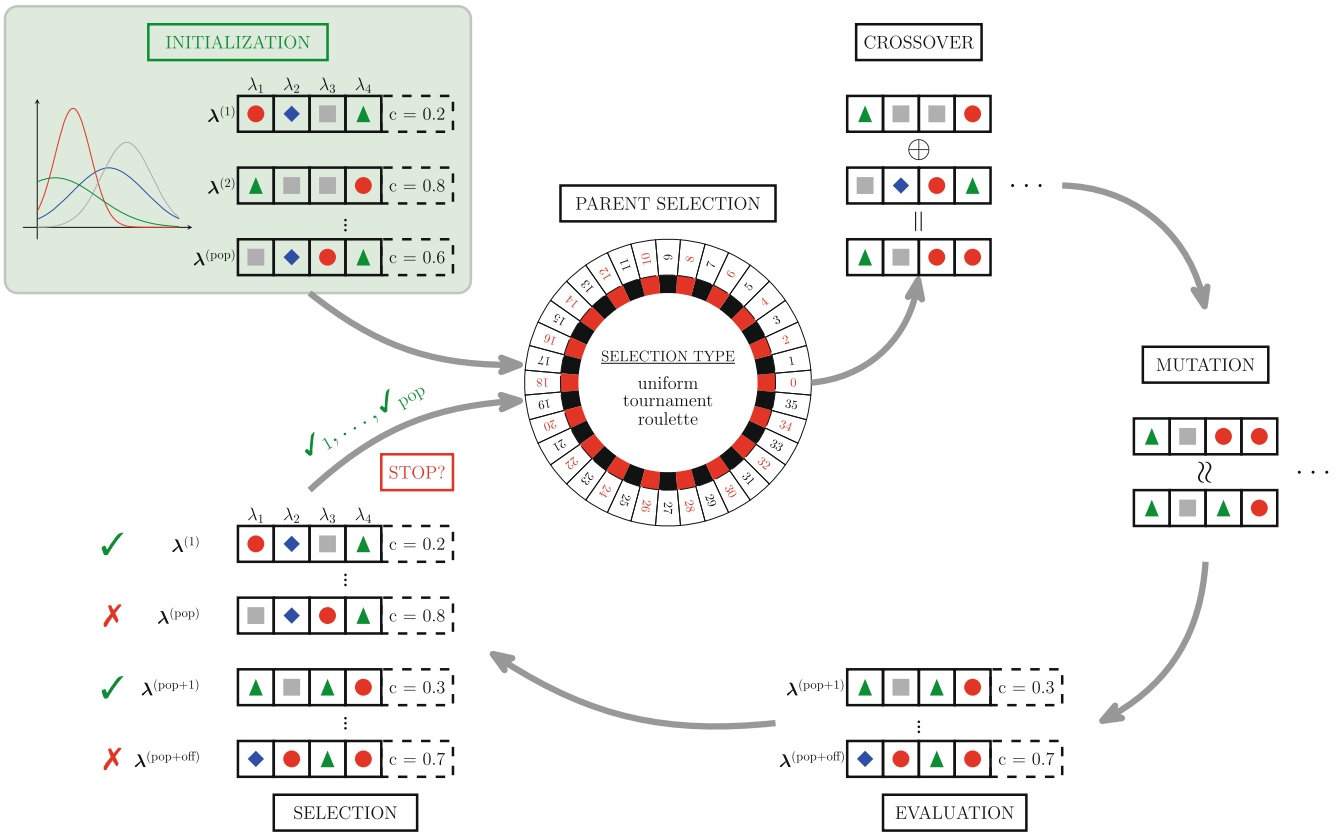


FIGURE 4 Schematic representation of a single iteration of an ES as a four dimensional discrete problem. Parameter values are symbolized by geometric shapes.

population is a currently maintained set of HPCs, and the *fitness* of an individual is its (inverted) generalization error $c(\lambda)$. *Mutation* is the (randomized) change of one or a few HP values in a configuration. *Crossover* creates a new HPC by (randomly) mixing the values of two other configurations. An ES follows iterative steps to find individuals with high fitness values (c.f. Figure 4): (i) An initial population is sampled at random. (ii) The fitness of each individual is evaluated. (iii) A set of individuals is selected as parents for reproduction.[6] (iv) The population is enlarged through *crossover* and *mutation* of the parents. (v) The offspring is evaluated. (vi) The *top-k* fittest individuals are selected.[7] (vii) Steps (ii) to (v) are repeated until a termination condition is reached. For a more comprehensive introduction to ES, see Beyer and Schwefel (2002).

ES were limited to numeric spaces in their original formulation, but they can easily be extended to handle mixed spaces by treating components of different types independently, for example, by adding a normally distributed random value to real-valued HPs while adding the difference of two geometrically distributed values to integer-valued HPs (Li et al., 2013). By defining mutation and crossover operations that operate on tree structures or graphs, it is even possible to perform optimization of preprocessing pipelines (Escalante et al., 2009; Olson et al., 2016) or neural network architectures (Real et al., 2019) using evolutionary algorithms. The properties of ES can be summarized as follows: ES have a low likelihood to get stuck in local minima, especially if so-called nested ES are used (Beyer & Schwefel, 2002). They can be straightforwardly modified to be robust to noise (Beyer & Sendhoff, 2006), and can also be easily extended to multi-objective settings (Coello Coello et al., 2007). Additionally, ES can be applied in settings with complex search spaces and can therefore work with spaces where other optimizers may fail (He et al., 2021). ES are more efficient than RS and GS but still often require a large number of iterations to find good solutions, which makes them unsatisfactory for expensive optimization settings like HPO.

4.2.3 | Bayesian optimization

Bayesian optimization (BO) has become increasingly popular as a global optimization technique for expensive black-box functions, and specifically for HPO (Hutter et al., 2011; Jones et al., 1998; Snoek et al., 2012).

BO is an iterative algorithm whose key strategy is to model the mapping $\lambda \mapsto c(\lambda)$ based on observed performance values found in the archive $\mathcal{A}$ via (non-linear) regression. This approximating model is called a *surrogate model*, for which a Gaussian process or a random forest are typically used. BO starts on an archive $\mathcal{A}$ filled with evaluated configurations, typically sampled randomly, using Latin Hypercube Sampling or the Sobol sampling (Bossek et al., 2020). BO then uses the archive to fit the surrogate model, which for each λ produces both an estimate of performance $\widehat{c}(\lambda)$ as well as an estimate of prediction uncertainty $\widehat{\sigma}(\lambda)$, which then gives rise to a predictive distribution for one test HPC or a joint distribution for a set of HPCs. Based on the predictive distribution, BO establishes a cheap-to-evaluate acquisition function $u(\lambda)$ that encodes a trade-off between *exploitation* and *exploration*: The former means that the surrogate model predicts a good, low c value for a candidate HPC λ, while the latter implies that the surrogate is very uncertain about $c(\lambda)$, likely because the surrounding area has not been explored thoroughly.

Instead of working on the true expensive objective, the acquisition function $u(\lambda)$ is then optimized in order to generate a new candidate λ^+ for evaluation. The optimization problem $u(\lambda)$ inherits most characteristics from $c(\lambda)$; so it is often still multi-modal and defined on a mixed, hierarchical search space. Therefore, $u(\lambda)$ may still be quite complex, but it is at least cheap to evaluate. This allows the usage of more budget-demanding optimizers on the acquisition function. If the space is real-valued and the combination of surrogate model and acquisition function supports it, even gradient information can be used.

Among the possible optimization methods are: iterated local search (as used by Hutter et al. (2009)), evolutionary algorithms (as in White et al. (2021)), ES using derivatives (as used by Sekhon and Mebane (1998) and Roustant et al. (2012)), and a focusing RS called *DIRECT* (Jones, 2009).

The true objective value $c(\lambda^+)$ of the proposed HPC λ^+—generated by optimization of $u(\lambda)$—is finally evaluated and added to the archive $\mathcal{A}$. The surrogate model is updated, and BO iterates until a predefined budget is exhausted, or a different termination criterion is reached. These steps are summarized in Algorithm 1. BO methods can use different ways of deciding which λ to return, referred to as the *identification step* by Jalali et al. (2017). This can either be the best observed λ during optimization, the best (mean, or quantile) predicted λ from the archive according to the surrogate model (Jalali et al., 2017; Picheny et al., 2013), or the best predicted λ overall (Scott et al., 2011). The latter options serve as a way of smoothing the observed performance values and reducing the influence of noise on the choice of $\widehat{\lambda}$.

ALGORITHM 1 **BO for a black-box objective $c(\lambda)$.**

Generate $\boldsymbol{\lambda}^{(1)}, \ldots, \boldsymbol{\lambda}^{(k)}$ with sampling scheme or fixed design
Initialize archive $\mathcal{A}^{[0]} = ((\boldsymbol{\lambda}^{(1)}, c(\boldsymbol{\lambda}^{(1)})), \ldots, (\boldsymbol{\lambda}^{(k)}, c(\boldsymbol{\lambda}^{(k)})))$
for $t = 1, 2, 3, \ldots$ *until termination* **do**
  1: Fit surrogate model $(\hat{c}(\boldsymbol{\lambda}), \hat{\sigma}(\boldsymbol{\lambda}))$ on $\mathcal{A}^{[t-1]}$
  2: Build acquisition function $u(\boldsymbol{\lambda})$ from $(\hat{c}(\boldsymbol{\lambda}), \hat{\sigma}(\boldsymbol{\lambda}))$
  3: Obtain proposal $\boldsymbol{\lambda}^{+}$ by optimizing u: $\boldsymbol{\lambda}^{+} \in \arg\max_{\boldsymbol{\lambda} \in \tilde{\Lambda}} u(\boldsymbol{\lambda})$
  4: Evaluate $c(\boldsymbol{\lambda}^{+})$
  5: Obtain $\mathcal{A}^{[t]}$ by augmenting $\mathcal{A}^{[t-1]}$ with $(\boldsymbol{\lambda}^{+}, c(\boldsymbol{\lambda}^{+}))$
end
Result: $\hat{\boldsymbol{\lambda}}$: Best-performing $\boldsymbol{\lambda}$ from archive or according to surrogates prediction.

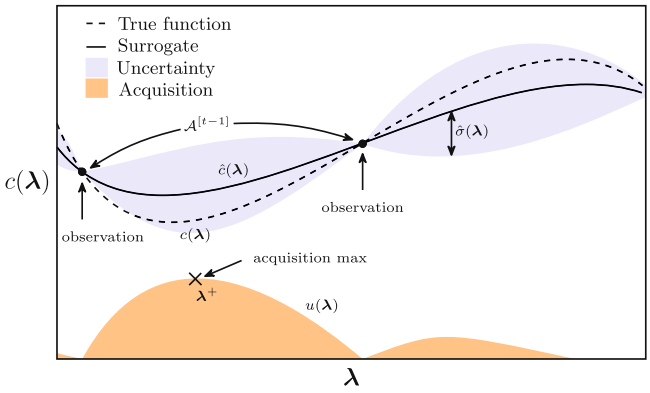


FIGURE 5 Illustration of how BO generates the proposal by maximizing an acquisition function (figure inspired by Hutter et al. (2019)).

Surrogate model

The choice of surrogate model has great influence on BO performance and is often linked to properties of $\widetilde{\Lambda}$. If $\widetilde{\Lambda}$ is purely real-valued, Gaussian process (GP) regression (Rasmussen & Williams, 2006)—sometimes referred to as Kriging—is used most often. In its basic form, BO with a GP does not support HPO with non-numeric or conditional HPs, and tends to show deteriorating performance when $\widetilde{\Lambda}$ has more than roughly 10 dimensions. Dealing with integer-valued or categorical HPs requires special care (Garrido-Merchán & Hernández-Lobato, 2020). Extensions for mixed-hierarchical spaces that are based on special kernels (Swersky et al., 2014) exist, and the use of random embeddings has been suggested for high-dimensional spaces (Nayebi et al., 2019; Wang et al., 2016). Most importantly, standard GPs have runtime complexity that is cubic in the number of samples, which can result in a significant overhead when the archive $\mathcal{A}$ becomes large (Figure 5).

McIntire et al. (2016) propose to use an adapted, sparse GP that restrains training data from uninteresting areas. Local Bayesian optimization (Eriksson et al., 2019) is implemented in the TuRBO algorithm and has been successfully applied to various black-box problems.

Random forests, most notably used in SMAC (Hutter et al., 2011), have also shown good performance as surrogate models for BO. Their advantage is their native ability to handle discrete HPs and, with minor modifications, for example, in Hutter et al. (2011), even dependent HPs without the need for preprocessing. Standard random forest implementations are still able to handle dependent HPs by treating infeasible HP values as missing and performing imputation. Random forests tend to work well with larger archives and introduce less overhead than GPs. SMAC uses the standard deviation of tree predictions as a heuristic uncertainty estimate $\widehat{\sigma}(\lambda)$ (Hutter et al., 2011). However, more sophisticated alternatives exist to provide unbiased estimates (Sexton & Laake, 2009). Since trees are not distance-based spatial models, the uncertainty estimator does not increase the further we extrapolate away from observed training points. This might be one explanation as to why tree-based surrogates are outperformed by GP regression on purely numerical search spaces (Eggensperger et al., 2013).

Neural networks (NNs) have shown good performance in particular with nontrivial input spaces, and they are thus increasingly considered as surrogate models for BO (Snoek et al., 2015). Discrete inputs can be handled by one-hot encoding or by automatic techniques, for example, entity embedding where a dense representation is learned from the output of a simple, direct encoding, such as one-hot encoding by the NN. (Hancock & Khoshgoftaar, 2020). NNs offer efficient and versatile implementations that allow the use of gradients for more efficient optimization of the acquisition function. Uncertainty bounds on the predictions can be obtained, for example, by using Bayesian neural networks (BNNs), which combine NNs with a probabilistic model of the network weights or adaptive basis regression where only a Bayesian linear regressor is added to the last layer of the NN (Snoek et al., 2015).

Acquisition function

The acquisition function balances out the surrogate model's prediction $\widehat{c}(\lambda)$ and its posterior uncertainty $\widehat{\sigma}(\lambda)$ to ensure both exploration of unexplored regions of $\widetilde{\Lambda}$, as well as exploitation of regions that have performed well in previous evaluations. A very popular acquisition function is the *expected improvement* (EI) (Jones et al., 1998):

$$\begin{aligned} u_{EI}(\lambda) &= \mathrm{E}_{C\sim\mathcal{N}\left(\widehat{c}(\lambda),\, \widehat{\sigma}(\lambda)^2\right)}\left[\max\left\{c_{\min} - C(\lambda),\ 0\right\}\right] \\ &= (c_{\min} - \widehat{c}(\lambda))\Phi\left(\frac{c_{\min} - \widehat{c}(\lambda)}{\widehat{\sigma}(\lambda)}\right) + \widehat{\sigma}(\lambda)\phi\left(\frac{c_{\min} - \widehat{c}(\lambda)}{\widehat{\sigma}(\lambda)}\right), \end{aligned} \tag{11}$$

where $c_{\min}$ denotes the best observed outcome of c so far, and Φ and ϕ are the cumulative distribution function and density of the standard normal distribution, respectively. The EI was introduced in connection with GPs that have a Bayesian interpretation, expressing the posterior distribution of the true performance value given already observed values as a Gaussian random variable $C(\lambda)$ with $C(\lambda) \sim \mathcal{N}\left(\widehat{c}(\lambda), \widehat{\sigma}(\lambda)^2\right)$. Under this condition, Equation (11) can be analytically expressed as above, and the resulting formula is often heuristically applied to other surrogates that supply $\widehat{c}(\lambda)$ and $\widehat{\sigma}(\lambda)$.

A further, very simple acquisition function is the *lower confidence bound* (LCB) (Jones, 2001):

$$u_{LCB}(\lambda) = (-1) \cdot (\widehat{c}(\lambda) - \kappa \cdot \widehat{\sigma}(\lambda)), \tag{12}$$

here negated to yield a maximization problem for Algorithm 1. The LCB treats local uncertainty as an additive bonus at each λ to enforce exploration, with κ being a control parameter that is not easy to set.

Adaptive explore-exploit tradeoffs

In a theoretical analysis without a limit on evaluations, Srinivas et al. (2010) suggest to increase the κ-parameter of LCB over time to encourage exploration in later phases of optimization. However, in the context of practical HPO with a finite budget for evaluations, it seems plausible to *decrease* κ over time to enforce exploration in the beginning and exploitation at the end in a similar cooldown scheme as in simulated annealing, such as, for example, suggested by Zheng et al. (2016), which uses a further cyclical scheme to escape local minima. Sasena et al. (2002) propose a cooldown scheme for expected improvement: They use the "generalized expected improvement" $u_{\mathrm{GEI}}(\lambda) = \mathrm{E}[\max\{c_{\min} - C(\lambda),\ 0\}^g]$, where larger values for exponent g also enforce more exploration. They suggest to start with a large g value in the beginning and to gradually decrease it to enforce exploitation at the end. Jasrasaria and Pyzer-Knapp (2018) propose to dynamically give exploitation more weight as a function of mean model-uncertainty in what they call "contextual improvement." This has a similar effect of encouraging late exploitation, as model uncertainty generally decreases over the course of optimization. Finally, de Ath et al. (2021) show that a very simply ϵ-greedy BO strategy can perform well or even better than established acquisition functions. They simply propose HPCs that maximize the predicted mean, but interleave random HPCs with ϵ probability. They show that this strategy performs well in settings with a low evaluation budget or with many dimensions, which is consistent with the other proposed methods, which adaptively emphasize exploitation more when remaining budget is low.

Multi-point proposal

In its original formulation, BO only proposes one candidate HPC per iteration and then waits for the performance evaluation of that configuration to conclude. However, in many situations, it is preferable to evaluate multiple HPCs in

parallel by proposing multiple configurations at once, or by asynchronously proposing HPCs while other proposals are still being evaluated.

While in the sequential variant, the best point can be determined unambiguously from the full information of the acquisition function. In the parallel variant, many points must be proposed at the same time without information about how the other points will perform. The objective here is to some degree to ensure that the proposed points are sufficiently different from each other.

The proposal of $n_{\text{batch}} > 1$ configurations in one BO iteration is called *batch proposal* or *synchronous parallelization* and works well if the runtimes of all black-box evaluations are somewhat homogeneous. If the runtimes are heterogeneous, one may seek to spontaneously generate new proposals whenever an evaluation thread finishes in what is called *asynchronous parallelization*. This offers some advantages to synchronous parallelization, but is more complicated to implement in practice.

The simplest option to obtain n_{batch} proposals is to use the LCB criterion in Equation (12) with different values for κ. For this so-called qLCB (also referred to as qUCB) approach, Hutter et al. (2012) propose to draw κ from an exponential distribution with rate parameter 1. This can work relatively well in practice but has the potential drawback of creating proposals that are too similar to each other (Bischl et al., 2014). Bischl et al. (2014) instead propose to maximize both $\widehat{c}(\lambda)$ and $\widehat{\sigma}(\lambda)$ simultaneously, using multi-objective optimization, and to choose n_{batch} points from the approximated Pareto-front. Further ways to obtain n_{batch} proposals are constant liar, Kriging believer (both described in Ginsbourger et al., 2010), and q-EI (Chevalier & Ginsbourger, 2013). Constant liar sets fake constant response values for the first points proposed in the batch to generate additional one via the normal EI principle and the approach; Kriging believer does the same but uses the GP model's mean prediction as fake value instead of a constant. The qEI optimizes a true multivariate EI criterion and is computationally expensive for larger batch sizes, but Balandat et al. (2020) implement methods to efficiently calculate the qEI (and qNEI for noisy observations) through MC simulations.

Efficient performance evaluation

While BO models only optimize the HPC prediction performance in its standard setup, there are several extensions that aim to make optimization more efficient by considering runtime or resource usage. These extensions mainly modify the acquisition function to influence the HPCs that are being proposed. Snoek et al. (2012) suggests the *expected improvement per second* (EIPS) as a new acquisition function. The EIPS includes a second surrogate model that predicts the runtime of evaluating a HPC in order to compromise between expected improvement and required runtime for evaluation. Most methods that trade off between runtime and information gain fall under the category of multi-fidelity methods, which is further discussed in Section 4.2.4. Acquisition functions that are especially relevant here consider information gain-based criteria like *Entropy Search* (Hennig & Schuler, 2012) or *Predictive Entropy Search* (Hernández-Lobato et al., 2016). These acquisition functions can be used for selective subsample evaluation (Klein et al., 2017), reducing the number of necessary resampling iterations (Swersky et al., 2013), and stopping certain model classes, such as NNs, early.

4.2.4 | Multifidelity and hyperband

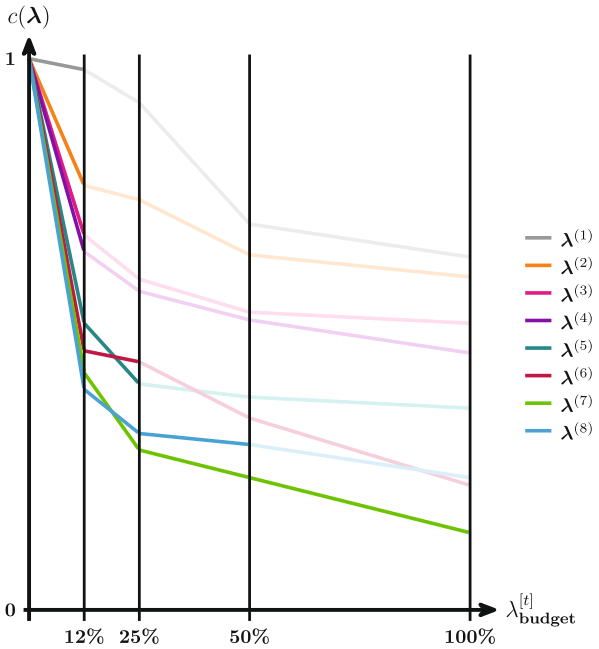
The multifidelity (MF) concept in HPO refers to all tuning approaches that can efficiently handle a learner $\mathcal{I}(\mathcal{D}, \lambda)$ with a fidelity HP λ_{fid} as a component of λ, which influences the computational cost of the fitting procedure in a monotonically increasing manner. Higher λ_{fid} values imply a longer runtime of the fit. This directly implies that the lower we set λ_{fid}, the more points we can explore in our search space, albeit with much less reliable information w.r.t. their true performance. If λ_{fid} has a linear relationship with the true computational costs, we can directly sum the λ_{fid} values for all evaluations to measure the computational costs of a complete optimization run. We assume to know box-constraints of λ_{fid} in form of a lower and upper limit, so $\lambda_{\text{fid}} \in \left[\lambda_{\text{fid}}^{\text{low}}, \lambda_{\text{fid}}^{\text{upp}}\right]$, where the upper limit implies the highest fidelity returning values closest to the true objective value at the highest computational cost. Usually, we expect higher values of λ_{fid} to be better in terms of predictive performance yet naturally more computationally expensive. However, overfitting can occur at some point, for example when λ_{fid} controls the number of training epochs when fitting an NN. Furthermore, we assume that the relationship of the fidelity to the prediction performance changes somewhat smoothly. Consequently, when evaluating multiple HPCs with small λ_{fid}, this at least indicates their true ranking. Typically, this implies a sequential fitting procedure, where λ_{fid} is, for example, the number of (stochastic) gradient descent steps or the number of sequentially added (boosting) ensemble members. A further, generally applicable option

is to subsample the training data from a small fraction to 100% before training and to treat this as a fidelity control (Klein et al., 2017). HPO algorithms that exploit such a λ_{fid} parameter—usually by spending budget on cheap HPCs with low λ_{fid} values earlier for exploration, and then concentrating on the most promising ones later—are called *multifidelity methods*. One can define two versions of the MF-HPO problem. (a) If overfitting can occur with higher values of λ_{fid} (e.g., if it encodes training iterations), simply minimizing $\min_{\lambda \in \Lambda} c(\lambda)$ is already appropriate. (b) If the assumption holds that a higher fidelity always results in a better model (e.g., if λ_{fid} controls the size of the training set), we are interested in finding the configuration λ^* for which the inducer will return the best model given the full budget, so $\min_{\lambda \in \Lambda, \lambda_{\text{fid}} = \lambda_{\text{fid}}^{upp}} c(\lambda)$. Of course, in both versions, the optimizer can make use of cheap HPCs with low settings of λ_{fid} on its path to its result.

Hyperband (Li et al., 2018) can best be understood as repeated execution of the *successive halving* (SH) procedure (Jamieson & Talwalkar, 2016). SH assumes a fidelity-budget B for the sum of λ_{fid} for all evaluations. It starts with a given, fixed number of candidates $\lambda^{(i)}$ that we denote with $p^{[0]}$ and "races them down" in stages t to a single best candidate by repeatedly evaluating all candidates with increased fidelity in a certain schedule. Typically, this is controlled by the η_{HB} control multiplier of Hyperband with $\eta_{\text{HB}} > 1$ (typically set to 2 or 3): After each batch evaluation t of the current population of size $p^{[t]}$, we reduce the population to the best $\frac{1}{\eta_{\text{HB}}}$ fraction and set the new fidelity for a candidate evaluation to $\eta_{\text{HB}} \times \lambda_{\text{fid}}$. Thus, promising HPCs are assigned a higher fidelity overall, and sub-optimal ones are discarded early on. The starting fidelity $\lambda_{\text{fid}}^{[0]}$ and the number of stages $s+1$ are computed in a way such that each batch evaluation of an SH population has approximately $B/(s+1)$ amount of fidelity units spent. Overall, this ensures that approximately, but not more than, B fidelity units are spent in SH:

$$\sum_{t=0}^{s} \left\lfloor p^{[0]} \eta_{\text{HB}}^{-t} \right\rfloor \lambda_{\text{fid}}^{[0]} \eta_{\text{HB}}^{t} \leq B. \tag{13}$$

However, the efficiency of SH strongly depends on a sensible choice of the number of starting configurations and the resulting schedule. If we assume a fixed fidelity-budget for HPO, the user has the choice of running either (a) more configurations but with less fidelity, or (b) fewer configurations, but with higher fidelity. While the former naturally explores more, the latter schedules evaluations with stronger correlation to the true objective value and more informative evaluations. As an example, consider how $\lambda^{(6)}$ is discarded in favor of $\lambda^{(8)}$ at 25% in Figure 6. Because their



bracket 3

t	$\lambda_{\text{budget}}^{[t]}$	$p_3^{[t]}$
0	1	8
1	2	4
2	4	2
3	8	1

bracket 2

t	$\lambda_{\text{budget}}^{[t]}$	$p_2^{[t]}$
0	2	6
1	4	3
2	8	1

bracket 1

t	$\lambda_{\text{budget}}^{[t]}$	$p_1^{[t]}$
0	4	4
1	8	2

bracket 0

t	$\lambda_{\text{budget}}^{[t]}$	$p_0^{[t]}$
0	8	4

FIGURE 6 Right: Bracket design of HB with $\lambda_{\text{fid}}^{\text{upp}} = 8$ and $\eta_{\text{HB}} = 2$ (resulting in four brackets). Left: Exemplary bracket run (figure inspired by Hutter et al. (2019)). Faint lines represent future performance of HPCs that were discarded early.

performance lines would have crossed close to 100%, $\lambda^{(6)}$ is ultimately the better configuration. However, in this case, the superiority of $\lambda^{(6)}$ was only observable after full evaluation.

As we often have no prior knowledge regarding this effect, HB simply runs SH for different numbers of starting configurations $p_s^{[0]}$, and each SH run or schedule is called a *bracket*. As input, HB takes η_{HB} and the maximum fidelity $\lambda_{\text{fid}}^{\text{upp}} > \eta_{\text{HB}}$. HB then constructs the target fidelity budget B for each bracket by considering the most explorative bracket: Here, the number of batch evaluations $s_{\max}+1$ is chosen to be $\left\lfloor \log_{\eta_{\text{HB}}}\left(\lambda_{\text{fid}}^{\text{upp}}\right)\right\rfloor + 1$ for which $\lambda_{\text{fid}}^{[0]} = \lambda_{\text{fid}}^{\text{upp}} \eta_{\text{HB}}^{-s_{\max}} \in \left(\eta_{\text{HB}}^{-1}, \eta_{\text{HB}}\right)$, $\lambda_{\text{fid}}^{[s_{\max}]} = \lambda_{\text{fid}}^{\text{upp}}$, and we collect these values in $\boldsymbol{r} = \left(\lambda_{\text{fid}}^{\text{upp}} \eta_{\text{HB}}^{-s_{\max}}, \lambda_{\text{fid}}^{\text{upp}} \eta_{\text{HB}}^{-s_{\max}+1}, \lambda_{\text{fid}}^{\text{upp}} \eta_{\text{HB}}^{-s_{\max}+2}, \ldots, \lambda_{\text{fid}}^{\text{upp}}\right) \in \mathrm{R}^{s_{max}+1}$. Since we want to spend approximately the same total fidelity and reduce the candidates to one winning HPC in every batch evaluation, the fidelity budget of each bracket is $B = (s_{\max}+1)\lambda_{\text{fid}}^{\text{upp}}$. For every $s \in \{0, \ldots, s_{\max}\}$, a bracket is defined by setting the starting fidelity $\lambda_{\text{fid}}^{[0]} \geq \lambda_{\text{fid}}^{\text{low}}$ of the bracket to $\mathbf{r}^{(1+s_{\max}-s)}$, resulting in $s_{\max}+1$ brackets and an overall fidelity budget of $(s_{\max}+1)B$ spent by HB. Consequently, every bracket s consists of $s+1$ batch evaluations, and the starting population size $p_s^{[0]}$ is the maximum value that fulfills Equation (13). The full algorithm is outlined in Algorithm 2, and the bracket design of HB with $\lambda_{\text{fid}}^{\text{upp}} = 8$ and $\eta_{\text{HB}} = 2$ is shown in Figure 6.

Starting configurations are usually sampled uniformly, but Li et al., 2018 also show that any stationary sampling distribution is valid. Because HB is a random-sampling-based method, it can trivially handle hierarchical HP spaces in the same manner as RS.

Multifidelity Bayesian optimization

The idea behind Hyperband—trying to discard HPCs that do not perform well early on—is somewhat orthogonal to the idea behind BO, that is, intelligently proposing HPCs that are likely to improve performance or to otherwise gain information about the location of the optimum. It is therefore natural to combine these two methods. This has first been achieved with BOHB by Falkner et al. (2018), who progressively increase λ_{fid} of suggested HPCs as in Hyperband. However, instead of proposing HPCs randomly, they use a model-based approach equivalent to maximizing expected improvement. They show that BOHB performs similar to HB in the low-budget regime, where it is superior to normal BO methods, but outperforms HB and perform similar or better to BO when enough budget for tens of full-budget evaluations are available.

Hyperband-based multi-fidelity methods have a control parameter that functions similar to η_{HB} described above, which determines the fraction of configurations that are discarded at every λ_{fid} value for which evaluations are

ALGORITHM 2 Hyperband algorithm (Li et al., 2018) where

- get_HPCs(p) uses a stationary sampling distribution to generate the initial HPC population of size p,
- top_k(Λ_s, C, k) selects the k HPCs in Λ_s associated to the k best performances in C as the next HPC population.

input: maximum fidelity per HPC $\lambda_{\text{fid}}^{\text{upp}}, \eta_{\text{HB}}$
initialization: $s_{\max} = \left\lfloor \log_{\eta_{\text{HB}}}(\lambda_{\text{fid}}^{\text{upp}}) \right\rfloor$, $B = (s_{\max}+1)\lambda_{\text{fid}}^{\text{upp}}$
  $\boldsymbol{r} = (\lambda_{\text{fid}}^{\text{upp}} \eta_{\text{HB}}^{-s_{\max}}, \lambda_{\text{fid}}^{\text{upp}} \eta_{\text{HB}}^{-s_{\max}+1}, \lambda_{\text{fid}}^{\text{upp}} \eta_{\text{HB}}^{-s_{\max}+2}, \ldots, \lambda_{\text{fid}}^{\text{upp}})$
for $s = s_{max}, s_{max}-1, \ldots, 0$ **do**
  $p_s = \left\lceil \frac{B}{\lambda_{\text{fid}}^{\text{upp}}} \frac{\eta_{\text{HB}}^s}{s+1} \right\rceil$
  $\Lambda_s^{[0]} = \text{get\_HPCs}(p_s) \quad (= \{\boldsymbol{\lambda}_s^{(1)}, \ldots, \boldsymbol{\lambda}_s^{(p_s)}\},\ \boldsymbol{\lambda}_s^{(i)} \in \tilde{\Lambda})$
  // Successive Halving inner loop
  for $t = 0, \ldots, s$ **do**
    $p_s^{[t]} = \left\lfloor p_s \eta_{\text{HB}}^{-t} \right\rfloor$
    Set λ_{fid} components of entries of $\Lambda_s^{[t]}$ to $\boldsymbol{r}^{(1+s_{\max}-s+t)} \quad (= (\lambda_{\text{fid}}^{\text{upp}} \eta_{\text{HB}}^{-s}) \cdot \eta_{\text{HB}}^{t})$
    $C^{[t]} = \{c(\boldsymbol{\lambda}) : \boldsymbol{\lambda} \in \Lambda_s^{[t]}\}$
    $\Lambda_s^{[t+1]} = \text{top\_k}(\Lambda_s^{[t]}, C^{[t]}, \left\lfloor p_s^{[t]} / \eta_{\text{HB}} \right\rfloor)$
  end
end
Result: HPC with best performance

performed. However, the optimal proportion of configurations to discard may vary depending on how strong the correlation is between performance values at different fidelities. An alternative approach is to use the surrogate model from BO to make adaptive decisions about what λ_{fid} values to use, or what HPCs to discard. Algorithms following this approach typically use a method first proposed by Swersky et al. (2013), who use a surrogate model for both the performance, as well as the resources used to evaluate an HPC. Entropy search (Hennig & Schuler, 2012) is then used to maximize the information gained about the maximum for when $\lambda_{\text{fid}} = \lambda_{\text{fid}}^{\text{upp}}$ per unit of predicted resource expenditure. Low-fidelity HPCs are evaluated whenever they contribute disproportionately large amounts of information to the maximum compared with their needed resources. A special challenge that needs to be solved by these methods is the modeling of performance with varying λ_{fid}, which often has a different influence than other HPs and is therefore often considered as a separate case. Another HPO method that specifically considers the training set size as fidelity HP is FABOLAS (Klein et al., 2017), which actively decides the training set size for each evaluation by trading off computational cost of an evaluation with a lot of data against the information gain on the potential optimal configuration.

In general, there could be other proposal mechanisms instead of random sampling as in Hyperband or BO as in BOHB. For example, Awad et al. (2021) showed that differential evolution can perform even better; however the evolution of population members across fidelities needs to be adjusted accordingly.

4.2.5 | Iterated racing

The iterated racing (IR, Birattari et al., 2010) procedure is a general algorithm configuration method that optimizes for a configuration of a general (not necessarily ML) algorithm that performs well over a given distribution of (arbitrary) problems. In most HPO algorithms, HPCs are evaluated using a resampling procedure such as CV, so a noisy function (error estimate for single resampling iterations) is evaluated multiple times and averaged. In order to connect racing to HPO, we now define a problem as a single holdout resampling split for a given ML data set, as suggested in Thornton et al. (2013) and Lang et al. (2015), and we will from now on describe racing only in terms of HPO.

The fundamental idea of *racing* (Maron & Moore, 1994) is that HPCs that show particularly poor performance when evaluated on the first problem instances (in our case: resampling folds) are unlikely to catch up in later folds and can be discarded early to save computation time for more interesting HPCs. This is determined by running a (paired) statistical test w.r.t. HPC performance values on folds. This allows for an efficient and dynamic allocation of the number of folds in the computation of $c(\lambda)$—a property of IR that is unique, at least when compared with the algorithms covered in this article.

Racing is similar to HB in that it discards poorly performing HPCs early. Like HB, racing must also be combined with a sampling metaheuristic to initialize a race. Particularly well-suited for HPO are iterated races (López-Ibáñez et al., 2016), and we will use the terminology of that implementation to explain the main control parameters of IR. IR starts by racing down an initial population of randomly sampled HPCs and then uses the surviving HPCs of the race to stochastically initialize the population of the subsequent race to focus on interesting regions of the search space.

Sampling is performed by first selecting a parent configuration λ among the N^{elite} survivors of the previous generation, according to a categorical distribution with probabilities $p_\lambda = 2\left(N^{elite} - r_\lambda + 1\right) / \left(N^{elite}\left(N^{elite}+1\right)\right)$, where r_λ is the rank of the configuration λ. A new HPC is then generated from this parent by mutating numeric HPs using a truncated normal distribution, always centered at the numeric HP value of the parent. Discrete parameters use a discrete probability distribution. This is visualized in Figure 7. The parameters of these distributions are updated as the optimization continues: The standard deviation of the Gaussian is narrowed to enforce exploitation and convergence, and the categorical distribution is updated to more strongly favor the values of recent ancestors. IR is able to handle search spaces with dependencies by sampling HPCs that were inactive in a parent configuration from the initial (uniform) distribution.

This algorithmic principle of having a distribution that is centered around well-performing candidates, is continuously sampled from and updated, is close to an estimation-of-distribution algorithm (EDA), a well-known template for ES (Larrañaga & Lozano, 2001). Therefore, IR could be described as an EDA with racing for noise handling.

IR has several control parameters that determine how the racing experiments are executed. We only describe the most important ones here; many of these have heuristic defaults set in the implementation introduced by López-Ibáñez et al. (2016). *Niter* (nbIterations) determines the number of performed races, defaulting to $\left\lfloor 2 + \log_2 \dim\left(\widetilde{\Lambda}\right) \right\rfloor$ (with $\dim\left(\widetilde{\Lambda}\right)$ the number of HPs being optimized). Within a race, each HPC is first evaluated on *Tfirst* (firstTest) folds before a first comparison test is made. Subsequent tests are then made after every *Teach*

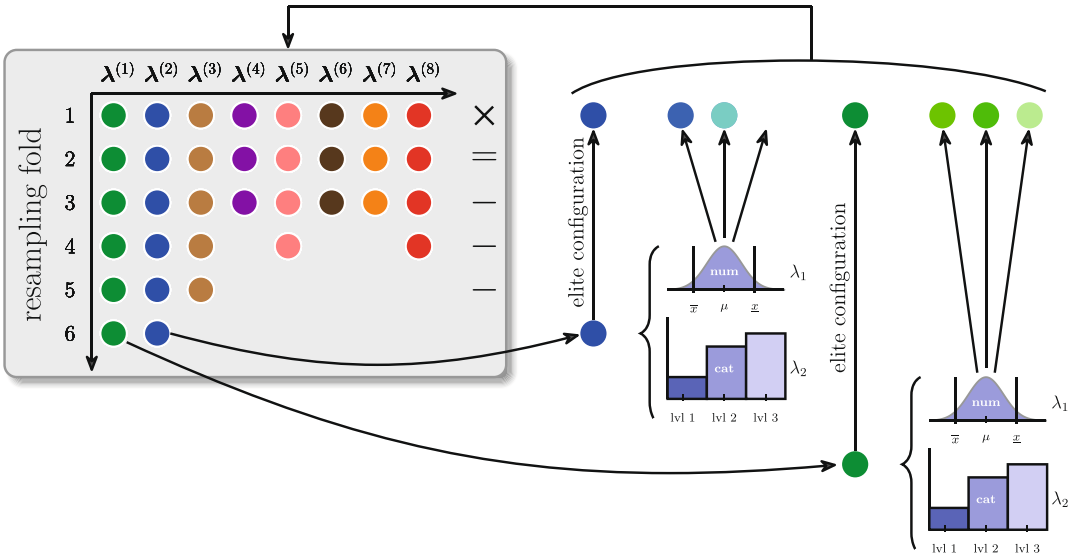


FIGURE 7 Scheme of the iterated racing algorithm (figure based on López-Ibáñez et al. (2016)).

(eachTest) evaluation. IR can be performed as *elitist*, which means surviving configurations from a generation are part of the next generation. The statistical test that discards individuals can be the Friedman test or the *t*-test (Birattari et al., 2002), the latter possibly with multiple testing correction. López-Ibáñez et al. (2016) recommend the *t*-test when when performance values for evaluations on different instances are commensurable and the tuning objective is the mean over instances, which is usually the case for our resampled performance metric where instances are simply resampling folds.

4.2.6 | Gradient-based optimization

Optimization methods that make use of gradients are often used successfully when optimizing model parameters with millions of dimensions (Soydaner, 2020). As discussed in Section 4.1, no gradient information with respect to hyperperameters is available in general, so these methods are not typically used for HPO. However, there is recent work aiming to make gradients available for certain model classes, in particular deep learning models, often making use of automatic differentiation capabilities of deep learning libraries. One approach by Lorraine et al. (2020) is based on implicit differentation along the path of the best response model parameters to the hyperparameters, and was shown to train neural networks with millions of hyperparameters. More generally, nested automatic differentiation can be used to derive *hypergradients*, that is, derivatives of a training objective with respect to the hyperparameters (Franceschi et al., 2017; Maclaurin et al., 2015). For a more general introduction to the topic of nested automatic differention and hypergradients we refer to Baydin et al. (2018).

4.3 | HPO: a Bilevel inference perspective

As discussed in Section 4.1, HPO produces an approximately optimal HPC $\widehat{\lambda} = \tau\left(\mathcal{D}, \mathcal{I}, \widetilde{\Lambda}, \rho\right)$ by optimizing it w.r.t. the resampled performance $c(\lambda) = \widehat{\mathrm{GE}}(\mathcal{I}, \mathcal{J}, \rho, \lambda)$. This is still risk minimization w.r.t. (hyper)parameters, where we search for optimal parameters $\widehat{\lambda}$ so that the risk of our predictor $\widehat{f}_{\widetilde{\lambda,\boldsymbol{\theta}}}$ becomes minimal when measured on validation data via

$$\rho_L\left(\mathbf{y}, \boldsymbol{F}_{\widetilde{\lambda,\boldsymbol{\theta}}}\right) = \sum_{i=1}^{m} L\left(y^{(i)}, \boldsymbol{F}^{(i)}_{\widetilde{\lambda,\theta}}\right), \tag{14}$$

where $\widehat{f}_{\widetilde{\lambda,\boldsymbol{\theta}}} = \mathcal{I}\left(\mathcal{D}_{\text{train}}, \widehat{\lambda}\right)$ and $\boldsymbol{F}_{\widetilde{\lambda,\theta}}$ is the prediction matrix of $\widehat{f}$ on validation data, for a pointwise loss function. The above is formulated for a single holdout split $(J_{\text{train}}, J_{\text{test}})$ in order to demonstrate the tight connection between (first level) risk minimization and HPO; Equation (8) provides the generalization for arbitrary resampling with multiple

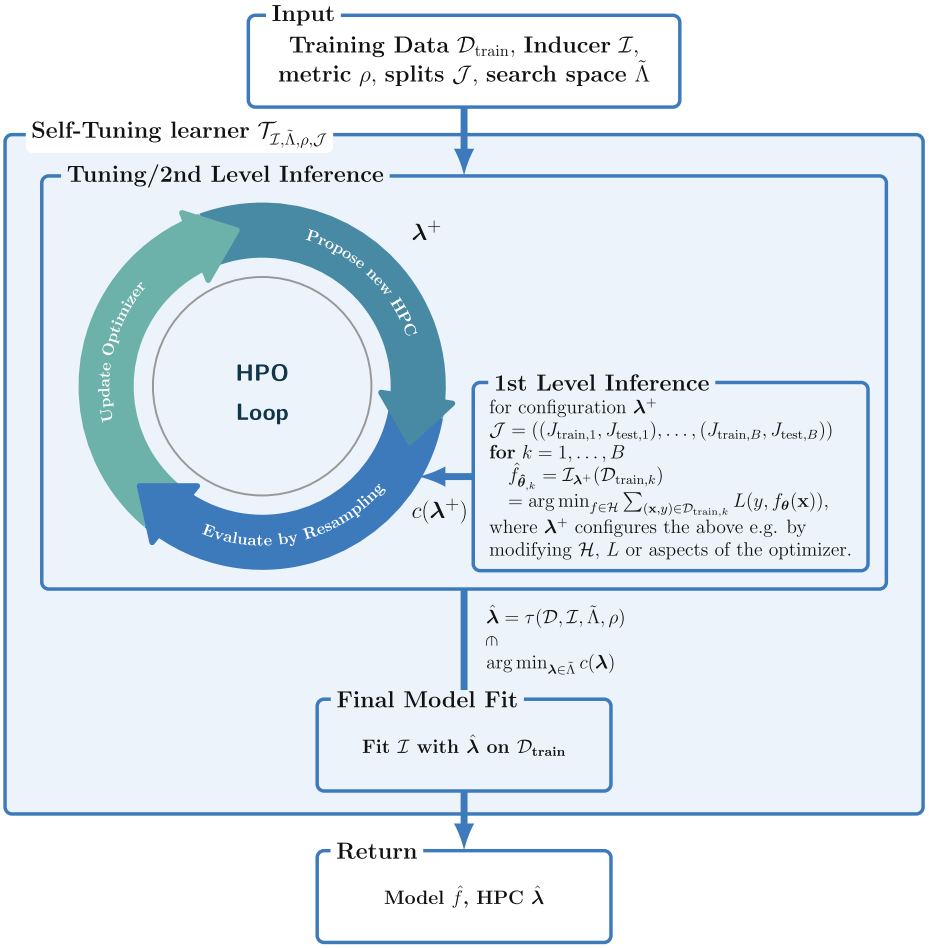


FIGURE 8 Self-tuning learner with integrated HPO wrapped around the inducer.

folds. This is somewhat obfuscated and complicated by the fact that we cannot evaluate Equation (8) in one go, but must rather fit one or multiple models $\widehat{f}$ during its computation (hence also its black-box nature). It is useful to conceptualize this as a bilevel inference mechanism; while the parameters $\widehat{\boldsymbol{\theta}}$ of f for a given HPC are estimated in the first level, in the second level we infer the HPs $\widehat{\lambda}$. However, both levels are conceptually very similar in the sense that we are optimizing a risk function for model parameters which should be optimal for the the data distribution at hand. In case of the second level, this risk function is not $\mathcal{R}_{\text{emp}}(\boldsymbol{\theta})$, but the harder-to-evaluate generalization error $\widehat{\text{GE}}$. An intuitive, alternative term for HPO is *second level inference* (Guyon et al., 2010), visualized in Figure 8.

There are mainly two reasons why such a bilevel optimization is preferable to a direct, joint risk minimization of parameters and HPs (Guyon et al., 2010):

- Typically, learners are constructed in such a way that optimized first-level parameters can be more efficiently computed for fixed HPs, for example, often the first-level problem is convex, while the joint problem is not.
- Since the generalization error is eventually optimized for the bilevel approach, the resulting model should be less prone to overfitting.

Thus, we can define a learner with integrated tuning as a mapping $\mathcal{T}_{\mathcal{I},\tilde{\Lambda},\rho,\mathcal{J}} : \mathrm{D} \to \mathcal{H}$, $\mathcal{D} \to \mathcal{I}_{\tau\left(\mathcal{D},\, \mathcal{I},\, \tilde{\Lambda},\, \rho\right)}(\mathcal{D})$, which maps a data set $\mathcal{D}$ to the model $\widehat{f}_{\widehat{\lambda}}$ that has the HPC set to $\widehat{\lambda}$ as optimized by τ on $\mathcal{D}$ and is then itself trained on the whole of $\mathcal{D}$; all for a given inducer $\mathcal{I}$, performance measure ρ, and search space $\tilde{\Lambda}$. Algorithmically, this learner has a 2-step training procedure (see Figure 8), where tuning is performed before the final model fit. This "self-tuning" learner $\mathcal{T}$ "shadows" the tuned HPs of its search space $\tilde{\Lambda}$ from the original learner and integrates their configuration into the

training procedure.[8] If such a learner is cross-validated, we naturally arrive at the concept of *nested CV*, which is discussed in the following Section 4.4.

4.4 | Nested resampling and meta-overfitting

As discussed in Section 3.2, the evaluation of any learner should always be performed via resampling on independent test sets to ensure non-biased estimation of its generalization error. This is necessary because evaluating $\widehat{f}$ on the data set $\mathcal{D}$ that was used for its construction would lead to an optimistic bias. In the general HPO problem as in Equation (10), we already minimize this generalization error by resampling:

$$\widehat{\lambda} \in \arg\min_{\lambda \in \Lambda} \widehat{c}(\lambda) = \arg\min_{\lambda \in \Lambda} \widehat{\mathrm{GE}}(\mathcal{I}, \mathcal{J}, \rho, \lambda). \tag{15}$$

If we simply report the estimated $c\left(\widehat{\lambda}\right)$ value of the returned best HPC, this also creates an optimistically biased estimator of the generalization error, as we have violated the fundamental "untouched test set" principle by optimizing on the test set(s) instead.

To better understand the necessity of an additional resampling step, we consider the following example in Figure 9, introduced by Bischl et al. (2012). Assume a balanced binary classification task and an inducer $\mathcal{I}_\lambda$ that ignores the data. Hence, λ has no effect, but rather "predicts" the class labels in a balanced but random manner. Such a learner always has a true misclassification error of $\mathrm{GE}(\mathcal{I}, \lambda, n_{\text{train}}, \rho) = 0.5$ (using ρ_{CE} as a metric), and any normal CV-based estimator will provide an approximately correct value as long as our data set is not too small. We now "tune" this learner, for example, by RS—which is meaningless, as λ has no effect. The more tuning iterations are performed, the more likely it becomes that some model from our archive will produce partially correct labels simply by random chance, and the (only randomly) "best" of these is selected by our tuner at the end. The more we tune, the smaller our data set, or the more variance our GE estimator exhibits, the more expressed this optimistic bias will be.

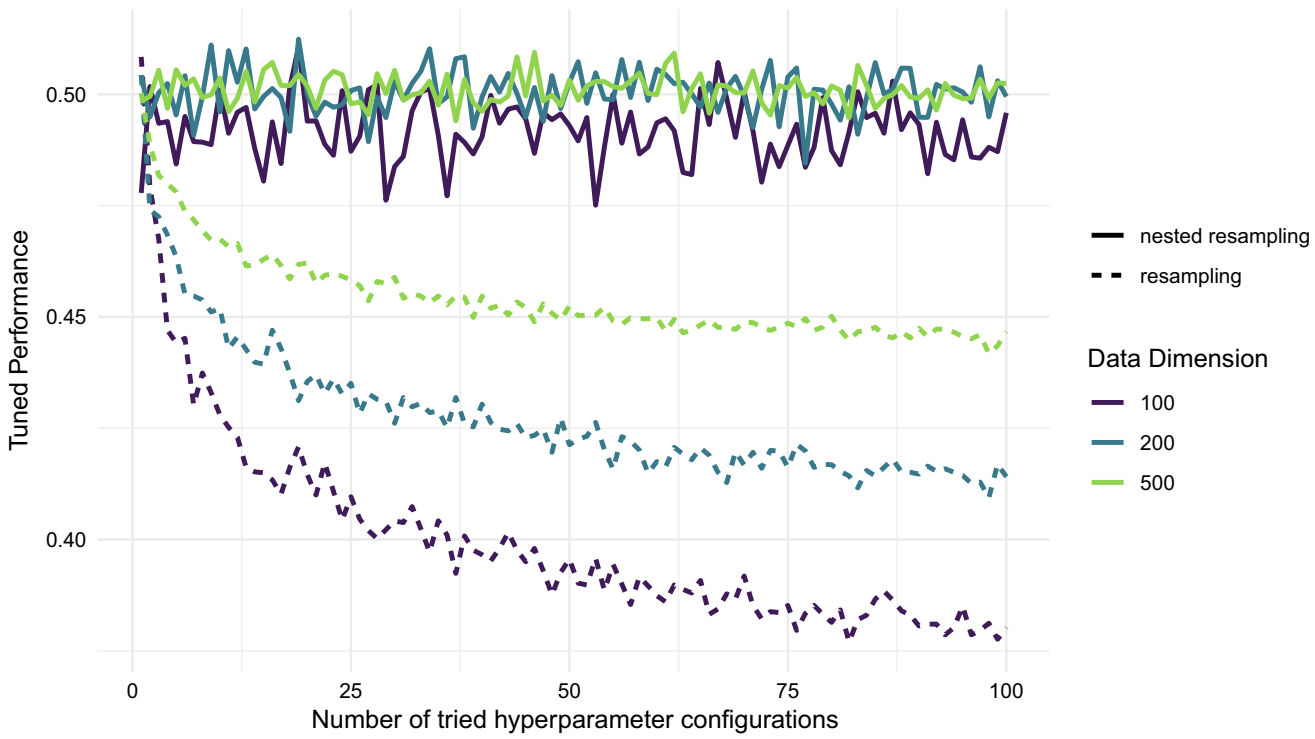


FIGURE 9 While nested resampling delivers correct results for the performance around 0.5, taking the tuning result directly results in a biased, optimistic estimator, especially on smaller data sets.

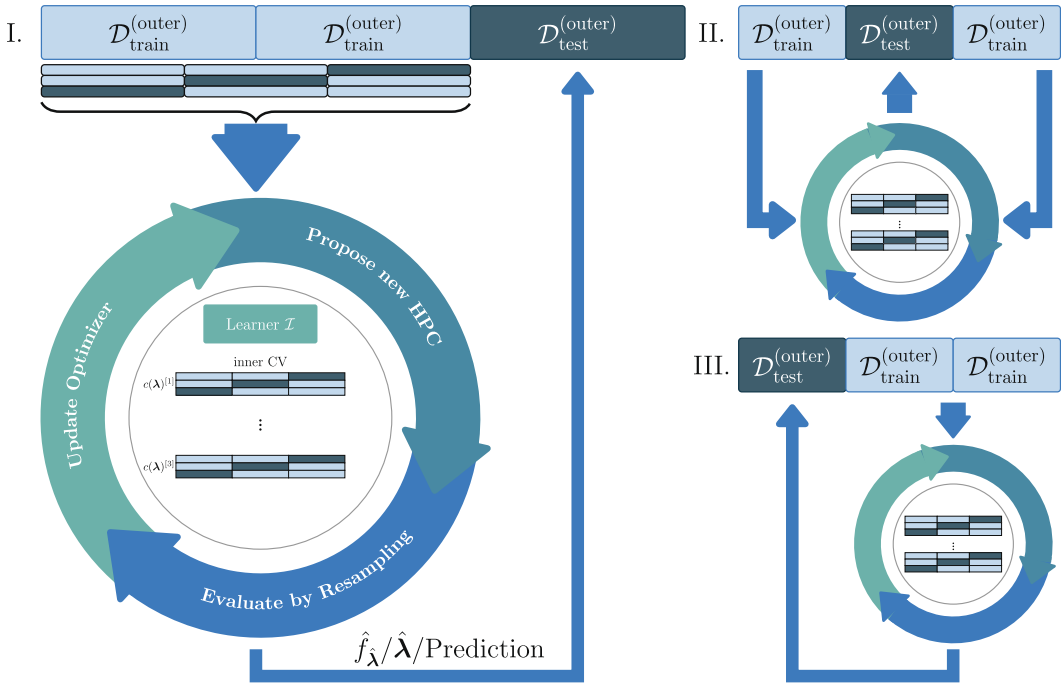


FIGURE 10 Nested CV with three inner and outer folds: Each HPC is evaluated on an inner CV, while the resulting tuned model is evaluated on the outer test set.

To avoid this bias, we introduce an additional outer resampling loop around this inner HPO-resampling procedure—or as discussed in Section 4.3, we simply regard this as cleanly cross-validating the self-tuned learner $\mathcal{T}_{\mathcal{I},\widetilde{\Lambda},\rho,\mathcal{J}}$. This is called *nested resampling*, which is illustrated in Figure 10.

The procedure works as follows: In the outer loop, an outer model-building or training set is selected, and an outer test set is cleanly set aside. Each proposed HPC λ^+ during tuning is evaluated via inner resampling on the outer training set. The best performing HPC $\widehat{\lambda}$ returned by tuning is then used to fit a final model for the current outer loop on the outer training set, and this model is then cleanly evaluated on the test set. This is repeated for all outer loops, and all outer test performances are aggregated at the end.

Some further comments on this general procedure: (i) Any resampling scheme is possible on the inside and outside, and these schemes can be flexibly combined based on statistical and computational considerations. Nested CV and nested holdout are most common. (ii) Nested holdout is often called the *train-validation-test* procedure, with the respective terminology for the generated three data sets resulting from the 3-way split. (iii) Many users often wonder which "optimal" HPC $\widehat{\lambda}$ they are supposed to report or study if nested CV is performed, with multiple outer loops, and hence multiple outer HPCs $\widehat{\lambda}$. However, the learned HPs that result from optimizations within CV are considered temporary objects that merely exist in order to estimate $\widehat{\text{GE}}(\mathcal{I}, \mathcal{J}, \rho, \lambda)$. The comparison to first-level risk minimization from Section 4.3 is instructive here: The formal goal of nested CV is simply to produce the performance distribution on outer test sets; the $\widehat{\lambda}$ can be considered as the fitted HPs of the self-tuned learner $\mathcal{T}$. If the parameters of a final model are of interest for further study, the tuner $\mathcal{T}$ should be fitted one final time on the complete data set. This would imply a final tuning run on the complete data set for second-level inference.

Nested resampling ensures unbiased outer evaluation of the HPO process, but, as CV for the first level, it is only a process that is used to estimate performance—it does not directly help in constructing a better model. The biased estimation of performance values is not a problem for the optimization itself, as long as all evaluated HPCs are still ranked correctly. But after a considerably large amount of evaluations, wrong HPCs might be selected due to stochasticity or overfitting to the splits of the inner resampling. This effect has been called either overtuning, meta-overfitting or oversearching (Ng, 1997; Quinlan & Cameron-Jones, 1995). At least parts of this problem seem directly related to the problem of multiple hypothesis testing. However, it has not been analyzed very well yet, and unlike regularization for (first level) ERM, not many counter measures are currently known for HPO.

4.5 | Threshold tuning

Most classifiers do not directly output class labels, but rather probabilities or real-valued decision scores, although many metrics require predicted class labels. A score is converted to a predicted label by comparing it to a threshold t so that $\widehat{y} = [f(\mathbf{x}) \geq t]$, where we use the Iverson bracket [] with $\widehat{y} = 1$ if $f(\mathbf{x}) \geq t$ and $\widehat{y} = 0$ in all other cases. For binary classification, the default thresholds are $t = 0.5$ for probabilities and $t = 0$ for scores. However, depending on the metric and the classifier, different thresholds can be optimal. *Threshold tuning* (Sheng & Ling, 2006) is the practice of optimizing the classification threshold t for a given model to improve performance. Strictly speaking, the threshold constitutes a further HP that must be chosen carefully. However, since the threshold can be varied freely after a model has been built, it does not need to be tuned jointly with the remaining HPs and can be optimized in a separate, subsequent, and cheaper step for each proposed HPC λ^+.

After models have been fitted and predictions for all test sets have been obtained when $c(\lambda^+)$ is computed via resampling, the vector of joint test set scores $\widetilde{\boldsymbol{F}}$ can be compared against the joint vector of test set labels $\mathbf{y}$ to optimize for an optimal threshold $\widehat{t} = \arg\min_t \rho\left(\widetilde{\mathbf{y}}, \left[\widetilde{\boldsymbol{F}} \geq t\right]\right)$, where the Iverson bracket is evaluated component-wise and $t \in [0, 1]$ for a binary probabilistic classifier and $t \in R$ for a score classifier. Since t is scalar and evaluations are cheap, $\widehat{t}$ can be found easily via a line search. This two-step approach ensures that every HPC is coupled with its optimal threshold. $c(\lambda^+)$ is then defined as the optimal performance value for λ^+ in combination with $\widehat{t}$.

The procedure can be generalized to multi-class classification. Class probabilities or scores $\widehat{\pi}_k(\mathbf{x}), k = 1,...,g$ are divided by threshold weights w_k, $k = 1,...,g$, and the k that yields the maximal $\frac{\widehat{\pi}_k(\mathbf{x})}{w_k}$ is chosen as the predicted class label. The weights $w_k, k = 1,...,g$ are optimized in the same way as t in the binary case. Generally, threshold tuning can be performed jointly with any HPO algorithm. In practice, threshold tuning is implemented as a post-processing step of an ML pipeline (Sections 5.1 and 5.2).

5 | PIPELINING, PREPROCESSING, AND AUTOML

ML typically involves several data transformation steps before a learner can be trained. If one or multiple preprocessing steps are executed successively, the data flows through a linear graph, also called pipeline. Section 5.1 explains why a pipeline forms an ML algorithm itself, and why its performance should be evaluated accordingly through resampling. Finally, Section 5.2 introduces the concept of flexible pipelines via hierarchical spaces.

5.1 | Linear pipelines

In the following, we will extend the HPO problem for a specific learner towards configuring a full pipeline including preprocessing. A *linear ML pipeline* is a succession of preprocessing methods followed by a learner at the end, all arranged as nodes in a linear graph. Each node acts in a very similar manner as the learner, and has an associated training and prediction procedure. During training, each node learns its parameters (based on the training data) and sends a

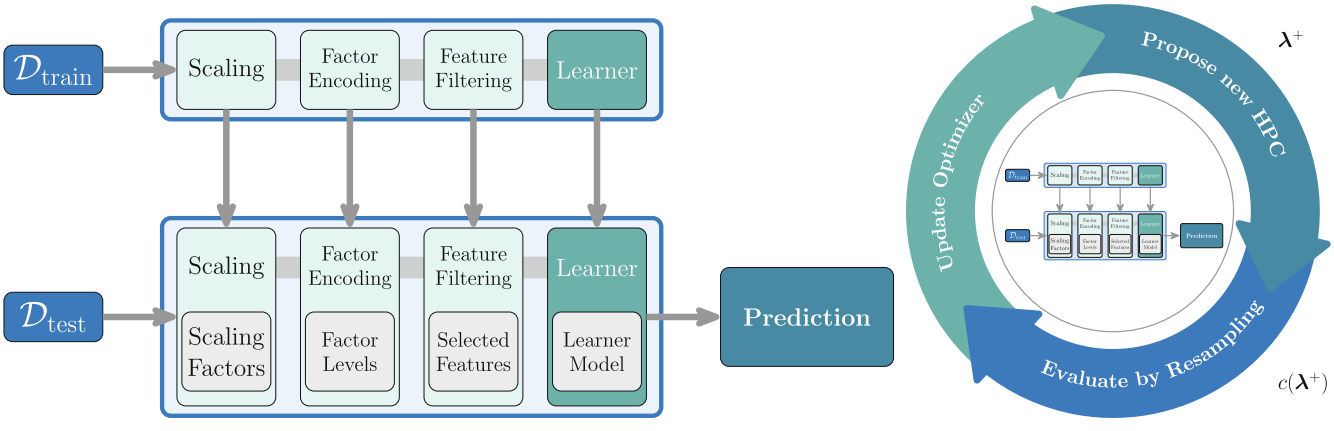
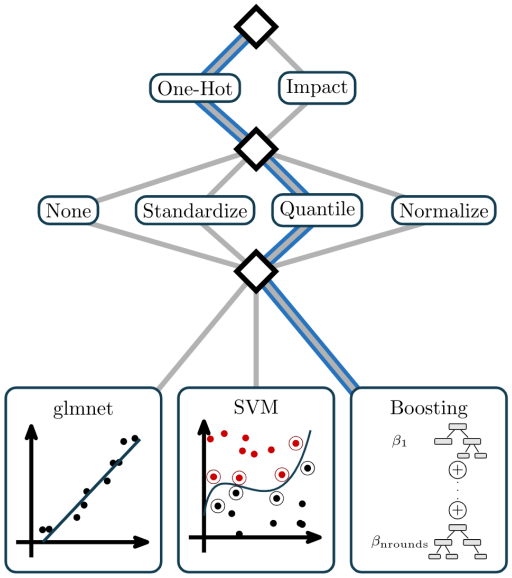


FIGURE 11 Example of a linear pipeline, including node parameters.



Search Space $\tilde{\Lambda}$

Name	Type	Bounds/Values	Trafo
◇ `encoding`	C	One-Hot, Impact	
◇ `scaling`	C	None, Standardize, Normalize, Quantile Transformation	
◇ `learner`	C	glmnet, SVM, Boosting	
`if learner = glmnet`			
`s`	R	$[-12, 12]$	2^x
`alpha`	R	$[0, 1]$	–
`if learner = SVM`			
`cost`	R	$[-12, 12]$	2^x
`gamma`	R	$[-12, 12]$	2^x
`if learner = Boosting`			
`eta`	R	$[-4, 0]$	10^x
`nrounds`	I	$\{1, \ldots, 5000\}$	–
`max_depth`	I	$\{1, \ldots, 20\}$	–

FIGURE 12 Example for a graph pipeline with operator selection via branches.

potentially transformed version of the training data to its successor. Afterwards, a pipeline can be used to obtain predictions: The nodes operate on new data according to their model parameters. Figure 11 shows a simple example.

Pipelines are important to properly embed the full model building procedure, including preprocessing, into cross-validation, so every aspect of the model is only inferred from the training data. This is necessary to avoid overfitting and biased performance evaluation (Bischl et al., 2012; Hornung et al., 2015), as it is for basic ML. As each node represents a configurable piece of code, each node can have HPs, and the HPs of the pipeline are simply the joint set of all HPs of its contained nodes. Therefore, we can model the whole pipeline as a single HPO problem with the combined search space $\widetilde{\Lambda} = \widetilde{\Lambda}_{\text{op},1} \times \cdots \times \widetilde{\Lambda}_{\text{op},k} \times \widetilde{\Lambda}_{\mathcal{F}}$.

5.2 | Operator selection and AutoML

More flexible pipelining, and especially the selection of appropriate nodes in a data-driven manner via HPO, can be achieved by representing our pipeline as a directed acyclic graph. Usually, this implies a single source node that accepts the original data and a single sink node that returns the predictions. Each node represents a preprocessing operation, a learner, a postprocessing operation, or a *directive* operation that directs how the data is passed to the child node(s).

One instance of such a pipeline is illustrated in Figure 12.

Here, we consider the choice between multiple mutually exclusive preprocessing steps, as well as the choice between different ML algorithms. Such a choice is represented by a branching operator, which can be configured through a categorical parameter and can determine the flow of the data, resulting in multiple "modeling paths" in our graph.

The HP space induced by a flexible pipeline is potentially more complex. Depending on the setting of the branching HP, different nodes and therefore different HPs are active, resulting in a very hierarchical search space. If we build a graph that includes a sufficiently large selection of preprocessing steps combined with sufficiently many ML models, the result can be flexible enough to work well on a large number of data sets—assuming it is correctly configured in a data-dependent manner. Combining such a graph with an efficient tuner is the key principle of AutoML (Feurer, Klein, et al., 2015a; Mohr et al., 2018; Olson et al., 2016).

6 | PRACTICAL ASPECTS OF HPO

In this section, we discuss practical aspects of HPO, which are more qualitative in nature and less often discussed in an academic context. Some of these recommendations are more "rules of thumb", based solely on experience, while others are at least partially confirmed by empirical benchmarks—keeping in mind that empirical benchmarks are only as good

as the selection of data sets. Even if a proper empirical study cannot be cited for every piece of advice in this section, we still propose that such a compilation is highly valuable for HPO practitioners.

6.1 | Choosing resampling and performance metrics

A resampling procedure is usually chosen based on two fundamental properties of the available data: (i) the number of observations, that is, to what degree do we face a small sample size situation, and (ii) whether the i.i.d. assumption for our data sampling process is violated.

For large sample sizes, simple holdout splitting can be used. A typical rule of thumb for medium to larger n is to choose $n_{\text{train}} = \frac{2}{3}n$ (Dobbin & Simon, 2011; Kohavi, 1995).[9] Many modern deep learning applications, for example image recognition, often use data sets with millions of observations, making a much smaller relative n_{test} reasonable.

For data sets of "medium" size with $500 \leq n \leq 50000$, 5- or 10-fold CV is usually recommended. The larger the data set, the fewer splits are necessary.

In other scenarios, for example, in the medical (image) domain, one often has to work with much smaller sample sizes, even if (strongly regularized) deep learning techniques are applied. In such cases, for example, with $n < 500$, repeated CV with a high number of repetitions should be used to reduce the variance while keeping the pessimistic bias small (Bischl et al., 2012). A typical instantiation that is often recommended is 10 times repeated 10-fold CV (Molinaro et al., 2005). *Stratified* sampling, which ensures that the relative class frequencies for each train/test split are consistent with the original data set, also helps in such a case.

One fundamental assumption about our data is that observations are i.i.d., that is, $\left(\mathbf{x}^{(i)}, y^{(i)}\right) \overset{i.i.d}{\sim} \mathrm{P}_{xy}$. This assumption is often violated in practice. A typical example is *repeated measurements*, where observations occur in "blocks" of multiple, correlated data, for example, from different hospitals, cities or persons. In such a scenario, we are usually interested in the ability of the model to generalize to new blocks. We must then perform CV with respect to the blocks, for example, "leave" one block "out." A related problem occurs if data are collected sequentially over a period of time. In such a setting, we are usually interested in how the model will generalize in the future, and the rolling or expanding window forecast must be used for evaluation (Bergmeir et al., 2018) instead of regular CV. However, discussing these special cases is out of scope for this work.

In HPO, resampling strategies must be specified for the inner as well as the outer level of nested resampling. The outer level is simply regular ML evaluation, and all comments from above hold. We advise readers to study further material such as Japkowicz and Shah (2011). The inner level concerns the evaluation of $c(\lambda)$ through resampling during tuning. While the same general comments from above apply, in order to reduce runtime, repetitions can also be scaled down. We are not particularly interested in very accurate numerical performance estimates at the inner level, and we must only ensure that HPCs are *ranked* properly during tuning to achieve correct selection, as discussed in Section 4.4. Hence, it might be appropriate to use a 10-fold CV on the outside to ensure proper generalization error estimation of the tuned learner, but to use only two folds or simple holdout on the inside.

In general, controlling the number of resampling repetitions on the inside should be considered an aspect of the tuner and should probably be automated away from the user (without taking away flexible control in cases of i.i.d. violations, or other deviations from standard scenarios). An alternative to statically choosing a fixed resampling strategy here is to dynamically allocate the number of train/test splits to do for each HPC. These ideas are common in algorithm configuration, and iterated racing, introduced in Section 4.2.5, does exactly this. BO can be extended with this idea as well (Lindauer et al., 2022), where *intensification* is applied to the currently best performing configuration, called *incumbent*. Intensification in case of HPO simply means to add additional resampling iterations. By only increasing the number of evaluations on the incumbent, compute budget is used efficiently as badly performing configurations are only evaluated by a comparatively small number of folds. Optimal Computing Budget Allocation (OCBA) can be used to optimize the probability of correct selection of the best HPC (Chen, 1995) from noisy evaluations. Bartz-Beielstein et al. (2011) combine OCBA with BO to iteratively distribute an evaluation budget between new HPCs proposed by BO and increasing the number of evaluations of previously evaluated ones.

The choice of the performance measure should be guided by the costs that suboptimal predictions by the model and subsequent actions in the real-word context of applying the model would incur. Often, popular but simple measures like accuracy do not meet this requirement. Misclassification of different classes can imply different costs. Failing to detect an illness may for example have a higher associated cost than mistakenly admitting a person to the hospital. There exists a plethora of performance measures that attempt to emphasize different aspects of misclassification with

respect to prediction probabilities and class imbalances, c.f. Japkowicz and Shah (2011). For other applications, it might be necessary to design a performance measure from scratch or based on underlying business key performance indicators (KPIs).

While a further discussion of metrics is again out of scope for this article, two pieces of advice are pertinent. First, as HPO is a black-box, no real constraints exist regarding the mathematical properties of ρ (or the associated outer loss). For first-level risk minimization, on the other hand, we usually require differentiability and convexity of L. If this is not fulfilled, we must approximate the KPI with a more convenient version. Second, for many applications, it is quite unclear whether there is a single metric that captures all aspects of model quality in a balanced manner. In such cases, it can be preferable to optimize multiple measures simultaneously, resulting in a multi-criteria optimization problem (Horn & Bischl, 2016).

6.2 | Choosing a pipeline and corresponding search space

For HPO, it is necessary to define the *search space* $\widetilde{\Lambda}$ over which the optimization is to be performed. This choice can have a large impact on the performance of the tuned model. A simple search space is a (lower dimensional) Cartesian product of individual HP sets that are either numeric (continuous or integer-valued) or categorical. Encoding categorical values as integers is a common mistake that degrades the performance of optimizers that rely on information about distances between HPCs, such as BO. The search intervals of numeric HPs typically must be bounded within a region of plausibly well-performing values for the given method and data set.

Many numeric HPs are often either bounded in a closed interval (e.g., $[0, 1]$) or bounded from below (e.g., $[0, \infty)$). The former can usually be tuned without modifications. HPs bounded by a left-closed interval should often be tuned on a *logarithmic* scale with a generous upper bound, as the influence of larger values often diminishes. For example, the decision whether k-NN uses $k = 2$ versus three neighbors will have a larger impact than whether it uses $k = 102$ versus $k = 103$ neighbors. The logarithmic scale can either be defined in the tuning software or must be set up manually by adjusting the algorithm to use *transformations*: If the desired range of the HP is $[a, b]$, the tuner optimizes on $[\log a, \log b]$, and any proposed value is transformed through an exponential function before it is passed to the ML algorithm. The logarithm and exponentiation must refer to the same base here, but which base is chosen does not influence the tuning process.

The size of the search space will also considerably influence the quality of the resulting model and the necessary budget of HPO. If chosen too small, the search space may not contain a particularly well-performing HPC. Choosing too wide HP intervals or including inadequate HPs in the search space can have an adverse effect on tuning outcomes in the given budget. If $\widetilde{\Lambda}$ is simply too large, it is more difficult for the optimizer to find the optimum or promising regions within the given budget. Furthermore, restricting the bounds of an HP may be beneficial to avoid values that are a priori known to cause problems due to unstable behavior or large resource consumption. If multiple HPs lead to poor performance throughout a large part of their range—for example, by resulting in a degenerate ML model or a software crash—the fraction of the search space that leads to fair performance then shrinks exponentially in the number of HPs with this problem. This effect can be viewed as a further manifestation of the so-called *curse of dimensionality*.

Due to this curse of dimensionality and the considerable runtime costs of HPO, we would like to tune as few HPs as possible. If no prior knowledge from earlier experiments or expert knowledge exists, it is common practice to leave other HPs at their software default values with the assumption that the developers of the algorithm chose values that work well under a wide range of conditions, although this is not necessarily given and it is often not documented how these defaults were specified. Recent approaches have studied how to empirically find optimal "default" values that work well under many conditions, by tuning ranges and HPC prior distributions based on extensive meta-data (Feurer, Klein, et al., 2015a; Gijsbers et al., 2021; Perrone et al., 2019; Pfisterer et al., 2021; Probst et al., 2019; Van Rijn & Hutter, 2018; Wistuba et al., 2015a, 2015b).

It is possible to optimize several learning algorithms in combination, as described in Section 5.2, but this introduces HP dependencies. The question then arises of which of the large number of ML algorithms (or preprocessing operations) should be considered. However, Fernández-Delgado et al. (2014) showed that in many cases, only a small but diverse set of learners is sufficient to choose one ML algorithm that performs reasonably well.

6.3 | Choosing an HPO algorithm

The number of HPs considered for optimization has a large influence on the difficulty of an HPO problem. Particularly large search spaces typically arise from optimizing over large pipelines or multiple learners (Section 5). With very few HPs (up to about 2–3) and well-defined discretization, GS may be useful due to its interpretable, deterministic, and reproducible nature; however, it is not recommended beyond this (Bergstra & Bengio, 2012). Standard BO with GPs work well for up to around 10 HPs. However, more HPs typically require more function evaluations—which in turn is problematic runtime-wise, since runtime complexity of GPs scales cubically in the number of dimensions. On the other hand, BO with RFs have been used successfully on search spaces with hundreds of HPs (Thornton et al., 2013) and can usually handle mixed hierarchical search spaces better. Pure sampling-based methods, such as RS and Hyperband, work well even for very large HP spaces as long as the "effective" dimension (i.e., the number of HPs that have a large impact on performance) is low, which is often observed in ML models (Bergstra & Bengio, 2012). Evolutionary algorithms (and those using similar metaheuristics, such as racing) can also work with truly large search spaces, and even with search spaces of arbitrarily complex structure if one is willing to use non-custom mutation and crossover operators. Evolutionary algorithms may also require fewer objective evaluations than RS. Therefore, they occupy a middle ground between (highly complex, sample-efficient) BO and (very simple but possibly wasteful) RS.

When performance evaluations are particularly expensive, which can happen when models are trained on large, complex datasets, then it is often beneficial to consider multi-fidelity optimization algorithms (Section 4.2.4). Simple multi-fidelity algorithms like Hyperband can then outperform more elaborate algorithms that do not take advantage of multi-fidelity evaluations; the advantage of multi-fidelity is greater when a single full model evaluation takes up a large fraction of the total evaluation budget available for HPO, that is, when only few full-fidelity evaluations are possible (Falkner et al., 2018). Algorithms that combine BO with multi-fidelity evaluation can be even more sample-efficient, although the advantage vis-a-vis Hyperband only appears after a few full-fidelity evaluations (Falkner et al., 2018). A natural way of doing multi-fidelity evaluation with DL methods is to stop training after a variable number of epochs, but even other ML methods can be evaluated in a multi-fidelity way by sub-sampling training data.

Another property of algorithms that is especially relevant to practitioners with access to large computational resources is *parallelizability*, which is discussed in Section 6.8.2. Furthermore, HPO algorithms differ in their simplicity, both in terms of their algorithmic principles and in terms of usability of available implementations, which can often have implications for their usefulness in practice. While more complex optimization methods, such as those based on BO, are often more sample-efficient or have other desirable properties compared with simple methods, they also have more components that can fail, which is particularly undesirable in large scale experiments. Here, numerical and algorithmic stability as well as robust error handling techniques might be worth more than theoretical efficiency. When performance evaluations are cheap and the search space is small, it may therefore be beneficial to fall back on simple and robust approaches such as RS, Hyperband, or any tuner with minimal inference overhead. The availability of any implementation at all (and the quality of that implementation) is also important; there may be optimization algorithms based on beautiful theoretical principles that have a poorly maintained implementation or are based on out-of-date software. The additional cost of having to port an algorithm to the software platform being used, or even implement it from scratch, could be spent on running more evaluations with an existing algorithm.

6.4 | Benchmarking HPO algorithms

One might wish to select an HPO algorithm based on its performance in prior benchmarks and competitions. Recent work made substantial progress in providing extensive comparison studies of popular HPO algorithms and several benchmark suites are made available to the community. These make it possible to investigate the advantages of newly developed HPO algorithms, or to tune strategy meta-parameters of HPO algorithms, either in general or for a specific domain.

A benchmarking suite for general continuous black box optimization is COCO (Hansen et al., 2021), which contains a well-curated collection of various synthetic benchmark functions and has been used in the general black box optimization community for many years. Kurobako[10] provides various general black-box optimizers and benchmark problems, Bayesmark[11] is especially suitable for benchmarking BO on real-world tasks, and LassoBench (Šehić et al., 2022) is suitable for benchmarking on high-dimensional optimization problems. One of the first benchmarks that specifically targets single-objective HPO algorithms is HPOlib (Eggensperger et al., 2013), which contains real-world HPO

problems, tabular and surrogate benchmarks, as well as synthetic test functions, and makes them available via a common API. HPOBench (Eggensperger et al., 2021) is the successor of HPOlib that focuses on reproducible, containerized benchmarks and multi-fidelity optimization problems. Pineda Arango et al. (2021) recently introduced HPO-B, a reproducible large-scale benchmark for transfer-HPO methods focusing on tabular benchmarks using data available on OpenML (Vanschoren et al., 2014). In contrast, YAHPO Gym (Pfisterer et al., 2022) focuses on surrogate-based benchmark problems, containing a large number of HPO problems that are suitable for benchmarking both multi-fidelity and multi-objective algorithms. Another focus is on efficiently, as the surrogates are provided in the form of compressed ONNX models that induce minimal memory and latency overhead. In contrast to other benchmarking suites, YAHPO Gym also contains many problems with higher dimensional and hierarchical search spaces. Finally, the benchmarks included in YAHPO Gym contain surrogates for a large number of tasks per scenario, which makes YAHPO Gym also suitable for evaluating transfer-HPO approaches. Klein et al. (2019) illustrate how a meta-surrogate model for HPO tasks can be used to generate new benchmark problems.

Published results from specific benchmarks or challenges can help as an orientation on what algorithms to use in what kinds of settings. It is important, however, to be aware of the scope of specific benchmarks: They indicate how well an algorithm works for a selected sample of data sets, a predefined budget, a specific parallelization setup, and specific learners and search spaces. Turner et al. (2021) have conducted an optimization challenge for different HPO-implementations. Although they find performance differences between specific implementations, they report that BO consistently outperforms RS. They find that, in practice, BO software profits from making use of ensembles of techniques (e.g., various surrogate models or various acquisition functions). This is in contrast to BO methods that are typically presented in scientific publications, which often try to prove a scientific point about the advantage of one aspect (surrogate model, acquisition function) over another. Eggensperger et al. (2013) showed that (i) BO with GPs were a strong approach for small continuous spaces with few evaluations, (ii) BO with Random Forests performed well for mixed spaces with a few hundred evaluations, and (iii) for large spaces and many evaluations, ES were the best optimizers. On HPOBench, BO with Random Forests (as implemented in SMAC, Lindauer et al., 2022) and differential evolution (if given enough budget) are well performing full-fidelity algorithms (Eggensperger et al., 2021). When also taking into account multi-fidelity methods, SMAC-HB (combining BO with Random Forests and Hyperband) and DEHB (combining differential evolution with Hyperband, Awad et al., 2021) are well-performing approaches, strongly improving upon vanilla Hyperband. However, if the optimization budget is sufficiently large, full-fidelity algorithms like BO with Random Forests catch up and eventually outperform multi-fidelity methods. On YAHPO Gym's single objective benchmark (YAHPO-SO v1), the central findings of HPOBench can be confirmed. For smaller optimization budgets, SMAC-HB is the best multi-fidelity method. But with increasing optimization budget, the full-fidelity method BO with Random Forests starts to outperform the multi-fidelity methods (Pfisterer et al., 2022). In contrast to findings reported in Eggensperger et al. (2021), DEHB did not outperform BOHB on the YAHPO-SO v1 benchmark. This may be due to (i) the inclusion of hierarchical search spaces in YAHPO Gym, which DEHB currently may not be able to handle well, and (ii) the choice of the optimization budget: Pfisterer et al. (2022) used a smaller optimization budget compared with the relatively large optimization budget used in Eggensperger et al. (2021). The advantage of fast surrogate benchmarks such as YAHPO Gym is that they make it feasible to optimize the configuration of HPO-algorithms themselves. It was for example used by Moosbauer et al. (2022) to create a new HPO algorithm by optimizing over the space of various algorithm components. Their analysis showed that an alternative, simpler approach to scheduling multi-fidelity evaluations performed as well as the approach used by Hyperband. In addition, the KDE-based sampling approach used by BOHB, which is not usually presented as one of BOHB's key features, was shown to have a large impact disproportionate to its limited prominence. Finally, they find that a KNN-based surrogate learner performs surprisingly well in comparison to more established choices such as RF.

6.5 | Choosing an implementation

In addition to the choice of ML and HPO algorithm, the choice of the specific implementation is also important in practice. Although it is surely possible to *manually* combine the implementation of one or multiple learner libraries with an HPO implementation, we strongly recommend utilizing an standard ML software framework instead. These frameworks are well-tested and simplify the process of model fitting, evaluation, and HPO, among other things, while managing the most common pitfalls and helping with error handling and parallel execution. In this subsection, we give an

TABLE 1 Feature matrix of tuning capabilities of *R*'s ML frameworks

	mlr3	caret	tidymodels	h2o
Tuners				
Random search	✓	✓	✓	✓
Grid search	✓	✓	✓	✓
Evolutionary	✓	×	✓	✓
Bayesian optimization	✓	✓	✓	✓
Hyperband	✓	✓	✓	✓
Racing	*	✓	✓	✓
Other tuners	NLOpt, SA	–	SA	–
Search space				
Transformations	✓	✓	✓	✓
Default search spaces	✓	i	✓	✓
Joint preproc tuning	✓	✓	✓	✓

Note: *not yet released on CRAN but prototype available, i: see description in main text.

overview of the available implementations and frameworks for ML and HPO for both R and Python and try to briefly categorize them on an abstract level.

6.5.1 | ML frameworks in R

Here, we summarize the HPO capabilities of the most important ML frameworks for R. The feature matrix in Table 1 gives an encompassing overview of the implemented tuning capabilities.

mlr3 (Lang et al., 2019) is a flexible toolbox for model fitting, resampling and evaluation and offers a sophisticated system for ML pipelines in mlr3pipelines (Binder et al., 2021). mlr3tuning provides various HPO methods, as well as a flexible description language for hyperparameter spaces. It includes grid search, random search, simulated annealing via GenSA, CMA-ES via cmaes, and non-linear optimization via nloptr. Additionally, mlr3hyperband provides Hyperband for multifidelity HPO, and the miesmuschel package implements a flexible toolbox for optimization using ES. Many BO variants are available through mlr3mbo.

caret (Kuhn, 2008) ships with grid search, random search, and adaptive resampling (AR)—a racing approach where an iterative optimization algorithm favors regions of empirically good predictive performance. Default search spaces are provided for each ML algorithm. However, if a custom search space is required, the user must specify a custom design to evaluate. Embedding preprocessing into the tuning process is possible. However, tuning the HPs of preprocessing methods is not supported.

tidymodels is the successor of caret. In combination with the tune package it is possible to perform grid search, random search, and Bayesian optimization. The AR racing approach and simulated annealing can be found in the finetune package. HP defaults and transformations are supported via dials. HPs of preprocessing operations using tidymodels' recipes pipeline system can be tuned jointly with the HPs of the ML algorithm.

h2o connects to the H2O cross-platform ML framework written in Java. Unlike the other discussed frameworks, which connect third-party packages from CRAN, h2o ships with its own implementations of ML models. The package supports grid search and random grid search, a variant of random search where points to be evaluated are randomly sampled from a discrete grid. The possibilities for preprocessing are limited to imputation, different encoders for categorical features, and correcting for class imbalances via under- and oversampling. H2O automatically constructs a search space for a given set of learning algorithms and preprocessing methods, and HPs of both can be tuned jointly. It is worth mentioning that h2o was developed with a strong focus on AutoML and offers the functionality to perform random search over a pre-defined grid, evaluating configurations of generalized linear models with elastic net penalty, xgboost models, gradient boosting machines, and deep learning models.

The best performing configurations found by the AutoML system are then stacked together (van der Laan et al., 2007) for the final model.

6.5.2 | Black-box and HP optimizers in R

Evolution strategies

Popular R packages that implement different ES are rgenoud (Mebane Jr. & Sekhon, 2011), cmaes, adagio, DEoptim (Mullen et al., 2011), mco, and ecr (Bossek, 2017) and mosmafs (Binder, Moosbauer, et al., 2020a).

Bayesian optimization

tune, rBayesianOptimization and DiceOptim (Roustant et al., 2012) are R packages that implement BO with Gaussian processes and for purely numerical search spaces. mlrMBO and its successor mlr3mbo offer to construct a flexible class of BO variants with arbitrary regression models and acquisition functions. These packages can work with mixed and hierarchical search spaces.

Other methods

Hyperband is implemented in the package mlr3hyperband. Iterated F-racing is provided by irace (López-Ibáñez et al., 2016).

6.5.3 | ML frameworks in python

Scikit-learn (Pedregosa et al., 2011, sklearn) is a general ML framework implemented in Python and the default choice for ML projects without the need for deep learning. The framework provides the most important traditional ML models like SVM, RFs, Boosting. This is further complemented by a variety of preprocessing and postprocessing methods, for example, ensembling. The clean and well-documented API allows users to easily build fairly complex pipelines, which can provide strong performance on tabular data, for example, see results on auto-sklearn (Feurer, Klein, et al., 2015a). It provides implementations of random forests and Gaussian processes that can be used as surrogate models for BO.

Modern DL frameworks such as Tensorflow,[12] PyTorch (Paszke et al., 2019), MXNet[13] and JAX[14] accelerate large-scale computation task, in particular matrix multiplication, by allowing to execute them on GPUs. Furthermore, GPflow (de G. Matthews et al., 2017) and GPyTorch (Gardner et al., 2018) are two GP libraries built on top of Tensorflow and PyTorch, respectively, and thus can be used to implement surrogate models for BO.

6.5.4 | Black-box and HP optimizers in python

The rapid development of the package landscape in Python has enabled many developers to also provide HPO tools in Python. Here, we will give a selective overview of commonly used and well-known open source packages.

Spearmint[15] (Snoek et al., 2012) was one of the first successful open-source BO packages for HPO. As proposed by Snoek et al. (2012), Spearmint implements standard BO with Markov chain Monte Carlo (MCMC) integration of the acquisition function for a fully Bayesian treatment of GP's HPs. Additionally, the second version of Spearmint allowed warping the input space with a Beta cumulative distribution function (Snoek et al., 2014) to deal with non-stationary loss landscapes. Spearmint also supports constrained BO (Gelbart et al., 2014) and parallel optimization (Snoek et al., 2012).

Hyperopt (Bergstra et al., 2013) is a distributed HPO framework. Unlike other frameworks that estimate the potential performance given a configuration, Hyperopt implements a Tree-structured Parzen Estimators (TPE) (Bergstra et al., 2011) that estimates the distribution of well-performing configurations in relation to underperforming configurations. This approach allows to easily run different configurations in parallel, as multiple density estimators can be executed at the same time.

Scikit-optimize[16] builds on top of scikit-learn and is a simple, yet efficient library to optimize expensive and noisy black-box functions with BO. It supports different surrogate models, incl. RF, grading-boosting trees and the classical

GP models. Similar to sklearn, Scikit-optimize allows for several ways of preprocessing input-features, for example, one-hot encoding, log transformation, normalization, label encoding.

SMAC (Hutter et al., 2011; Lindauer et al., 2022) was originally developed as a tool for algorithm configurations (Eggensperger et al., 2019). However, in recent years, it has also been successfully used as a key component of several AutoML tools, such as auto-sklearn (Feurer, Klein, et al., 2015a) and Auto-PyTorch (Zimmer et al., 2021). SMAC implements both RF and GP as surrogate models to handle various sorts of HP spaces. Additionally, BOHB (Falkner et al., 2018) is implemented as a multi-fidelity approach in SMAC; however, instead of the TPE model in BOHB, SMAC utilized RF as a surrogate model for BO part, showing better performance than the original BOHB (Eggensperger et al., 2021). The most recent version of SMAC (Lindauer et al., 2022) also allows for multi-objective optimization, for parallel optimization with Dask (Rocklin, 2015) and for specification of expert priors to make use of existing human expertise (Hvarfner et al., 2022).

HyperMapper (Nardi et al., 2019) also builds an RF as a surrogate model; thus, it also supports mixed and structured HP configuration spaces. However, HyperMapper does not implement a BO approach in a strict sense. In addition, HyperMapper incorporates previous knowledge by rescaling the samples with a beta distribution and handles unknown constraints by training another RF model as a probabilistic classifier. Furthermore, it supports multi-objective optimization. As HyperMapper cannot be allocated to a distributed computing environment, it might not be applicable to large-scale HPO problems.

OpenBox (Li et al., 2021) is a general framework for black-box optimization—including HPO—and supports multi-objective optimization, multi-fidelity, early-stopping, transfer learning, and parallel BO via distributed parallelization under both synchronous parallel settings and asynchronous parallel settings. OpenBox further implements an ensemble surrogate model to integrate the information from previously seen similar tasks.

Dragonfly (Kandasamy et al., 2020) is an open source library for scalable BO. Dragonfly extends standard BO in the following ways to scale to higher dimensional and expensive HPO problems: it implements GPs with additive kernels and additive GP-UCB (Kandasamy et al., 2015) as an acquisition function. In addition, it supports a multi-fidelity routine to scale to expensive HPO problems. To increase the robustness of the system w.r.t. its HPs, Dragonfly acquires its GP HP by either sampling a set of GP HPs from the posterior, conditioned on the data or optimizing the likelihood to handle different degrees of smoothness of the optimized loss landscape. In addition, Dragonfly maximizes its acquisition function with an evolutionary algorithm, which enables the system to work on different sorts of configuration spaces, for example, different variable types and constraints.

The previously described BO-based HPO tools fix their search space during the optimization phases. The package BayesianOptimization[17] can concentrate a domain around the current optimal values and adjust this domain dynamically (Stander & Craig, 2002). A similar idea is implemented in TurBO (Eriksson et al., 2019), which only samples inside a trust region while keeping the configuration space fixed. The trust region shrinks or extends based on the performance of TurBO's suggested configuration.

GPflowOpt[18] and BoTorch (Balandat et al., 2020) are the BO-based HPO frameworks that build on top of GPflow and GPyTorch, respectively. The automatic differentiation feature facilitates the users to freely extend their own approaches to existing models. Ax adds an easy-to-use API on top of BoTorch.

DEHB (Awad et al., 2021) uses differential evolution (DE) and combines it with a multi-fidelity HPO framework, inspired by BOHB's combination of Hyperband (Li et al., 2018) and BO. DEHB overcomes several drawbacks of BOHB: (i) it does not require a surrogate model, and thus, its runtime overhead does not grow over time; (ii) DEHB can have stronger final performance compared with BOHB, especially for discrete-valued and high-dimensional problems, where BO usually fails, as it tends to suggest points on the boundary of the search space (Oh et al., 2018); and (iii) parallel DE algorithms potentially scale better, since they can be parallelized more efficiently than BO. However, while DEHB is a well-working choice for discrete-valued search spaces, its performance decreases if the search space becomes hierarchical (Pfisterer et al., 2022). This effect might be explained by DEHB's mutation and crossover operations not appropriately handling HP dependencies.

One of the most popular and well-maintained HPO tools is Optuna (Akiba et al., 2019). It is known for its easy way of dynamically constructing complex HPO search spaces by its so-called Define-by-run API. It implements TPE (Bergstra et al., 2011) as a BO variant, CMA-ES as an evolutionary strategy, grid search and random search. In addition, it is straightforward to parallelize the HPO search and it comes with several tools to inspect and visualize the optimization history of the HPO process.

There are several other general HPO tools. For instance, Oríon[19] is an asynchronous framework for black-box function optimization. Tune[20] is a scalable HP tuning framework that provides APIs for several optimizers

mentioned before, for example, Dragonfly, SKopt, or HyperOpt. Similarly, Syne Tune (Salinas et al., 2022) is a modular HPO framework with different backends. It aims at facilitate reproducible and faster benchmarking of HPO algorithms.

6.5.5 | AutoML tools in python

In the previous section, we discussed several HPO tools in Python, which allow for flexible applications of different algorithms, search spaces and data sets. Here, we will briefly discuss several AutoML-packages.

Auto-Weka (Thornton et al., 2013) is one of the first AutoML tools that automates the design choice of entire ML packages, namely Weka, with SMAC. Similarly, the same optimizer is applied to Auto-sklearn (Feurer, Klein, et al., 2015a), while a meta-learning and ensemble approach further boost the performance of auto-sklearn. Similarly, Auto-keras (Jin et al., 2019) and Auto-PyTorch (Zimmer et al., 2021) use BO to automate the design choices of deep NNs. TPOT (Olson et al., 2016) automates the design choice of an ML pipeline with genetic algorithms, while a probabilistic grammatical evolution is implemented in AutoGOAL (Estévez-Velarde et al., 2020) for HPO problems. H2O automl (LeDell & Poirier, 2020) and AutoGluon (Erickson et al., 2020) train a set of base models and ensemble them with a single layer (H2O) or multi-layer stacks (AutoGluon). Additionally, TransmogrifAI[21] is an AutoML tool built on the top of Apache Spark and MLBox.[22]

6.6 | When to terminate HPO

Choosing an appropriate budget for HPO or dynamically terminating HPO based on collected archive data is a difficult practical challenge that is not readily solved. The user typically has these options: (i) A certain amount of runtime is specified before HPO is started, solely based on practical considerations and intuition. This is what currently happens nearly exclusively in practice. A simple rule-of-thumb might be scaling the budget of HPC evaluations with search space dimensionality in a linear fashion like $50 \times l$ or $100 \times l$, which would also define the number of full budget units in a multi-fidelity setup. The downside is that while it is usually simpler to define when a result is needed, it is nearly impossible to determine whether this computational budget is enough to solve the HPO problem reasonably well. (ii) The user can set a lower bound regarding the required generalization error, that is, what model quality is deemed good enough in order to put the resulting model to practical use. Notably, if this lower bound is set too optimistically, the HPO process might never terminate or simply take too long. (iii) If no (significant) progress is observed for a certain amount of time during tuning, HPO is considered to have converged or stagnated and stopped. This criterion bears the risk of terminating too early, especially for large and complex search spaces. Furthermore, this approach is conceptually problematic, as it mainly makes sense for iterative, local optimizers. No HPO algorithm from this article, except for maybe ES, belongs to this category, as they all contain a strong exploratory, global search component. This often implies that search performance can flatline for a longer time and then abruptly change. (iv) For BO, a small maximum value of the EI acquisition function (see Equation 11) indicates that further progress is estimated to be unlikely (Jones et al., 1998). This criterion could even be combined with other, no-BO type tuners. It implies that the surrogate model is trustworthy enough for such an estimate, including its local uncertainty estimate. This is somewhat unlikely for large, complex search spaces. In practice, it is probably prudent to set up a combination of all mentioned criteria (if possible) as an "and" or "or" combination, with somewhat liberal thresholds to continue the optimization for (ii), (iii), (iv), and (v), and an absolute bound on the maximal runtime for (i).

With many HPO tuners, it is possible to continue the optimization even after it has been terminated. One can even use (part of) the archive as an initialization of the next HPO run, for example, as an initial design for BO or as the initial population for an ES.

Some methods, for example, ES and under some circumstances BO, may get stuck in a subspace of the search space and fail to explore other regions. While some means of mitigation exist, such as interleaving proposed points with randomly generated points, there is always the possibility of re-starting the optimization from scratch multiple times, and selecting the best performance from the combined runs as the final result. The decision regarding the termination criterion is itself always a matter of cost–benefit calculation of potential increased performance against cost of additional evaluations. One should also consider the possibility that, for more local search strategies like ES, terminating early and restarting the optimization can be more cost-effective than letting a single optimization run continue for much longer. For more global samplers like BO and HB, it is much less clear whether and how such an efficient restart could be executed.

6.7 | Warm-starts

HPO may require substantial computational resources, as a large optimization search space may require many performance evaluations, and individual performance evaluations can also be very computationally expensive. One way to reduce the computational time are warm-starts, where information from previous experiments are used as a starting solution.

Warm-starting evaluations

Certain model classes may offer specific methods that reduce the computational resources needed for training a single model by transferring model parameters from other, similar configurations that were already trained. NNs with similar architectures can be initialized from trained networks to reduce training time—a concept known as weight sharing. Some neural architecture search algorithms are specifically designed to make use of this fact (Elsken et al., 2019).

Warm-starting optimization

Many HPO algorithms do not take any input regarding the relative merit of different HPCs beyond box-constraints on the search space $\widetilde{\Lambda}$. However, it is often the case that some HPCs work relatively well on a large variety of data sets (Probst et al., 2019). At other times, HPO may have been performed on other problems that are similar to the problem currently being considered. In both cases, it is possible to warm-start the HPO process itself—for example, by choosing the initial design as HPCs that have performed well in the past or by choosing HPCs that are known to perform well in general (Lindauer & Hutter, 2018). Large systematic collections of HPC performance on different data sets, such as those collected in Binder, Pfisterer, and Bischl (2020b) or on OpenML (van Rijn et al., 2013), can be used to build these initial designs (Feurer, Klein, et al., 2015a; Pfisterer et al., 2021).

Transfer learning

More generally, warm-starting can also be regarded as a transfer learning mechanism for HPO. For Bayesian optimization specifically, joint surrogate models across multiple tasks can be trained. This can be achieved by multi-task optimization (Perrone et al., 2018) or by formulating BO as a few-shot learning problem (Wistuba & Grabocka, 2021).

6.8 | Control of execution

HP tuning is usually computationally expensive, and running optimization in parallel is one way to reduce the overall time needed for a satisfactory result. During parallelization, *computational jobs* (which can be arbitrary function calls) are mapped to a pool of *workers* (usually distributed among individual physical CPU cores, or remote computational nodes). In practice, a *parallelization backend* orchestrates starting up, shutting down, and communication with the workers. The achieved speedup can be proportional to the number of workers in ideal cases, but less than linear scaling is typically observed, depending on the algorithm used (*Amdahl's law*, Rodgers, 1985). Besides parallelization, ensuring a fail-safe program flow often also requires special care. The fact that HPO tries many, sometimes extreme, configurations makes it more likely to trigger software bugs. Some model fitting and performance evaluation steps can then potentially crash or run for an unacceptably long time. It is recommended to launch individual evaluation steps in external processes, and to terminate these after a pre-specified amount of time has passed, to ensure that the main optimization process continues running even when individual model evaluations fail.

6.8.1 | Job hierarchy for HPO with nested resampling

HPO can be parallelized at different granularity or *parallelization levels*. These levels result from the nested loop outlined in Algorithm 3 and described in detail in the following (from coarse to fine granularity):

(a) One iteration of the *outer resampling loop* (Line 1), that is, tuning an ML algorithm on the respective training set of the outer resampling split. The result is the outer test set performance of the best HPC found and trained on the outer training set.

(b) The execution of one *tuning iteration* (Line 3), that is, the proposal and evaluation of a batch of new HPCs. The result is an updated optimization archive.

ALGORITHM 3 **Nested resampling as nested loops.**

```
1  foreach outer resampling iteration do
2  |  Initialize archive A = {};
3  |  foreach tuning iteration do
4  |  |  {λ^{+,1}, ..., λ^{+,n_batch}} = propose_points(A);
5  |  |  foreach i ∈ {1, ..., n_batch} do
6  |  |  |  c = {};
7  |  |  |  foreach inner resampling iteration do
8  |  |  |  |  c = c ∪ eval(λ^{+,i}, D_inner_train, D_inner_test);
9  |  |  |  end
10 |  |  |  A = A ∪ (λ^{+,i}, agr(c));
11 |  |  end
12 |  end
13 |  λ* = get_best(A);
14 |  eval(λ*, D_train, D_test);
15 end
```

(c) One *evaluation* of a single proposed HPC (Line 5), that is, one inner resampling with configuration $\lambda^{+,i}$. The result is an aggregated performance score.

(d) One iteration of the *inner resampling* loop (Line 7). The result is the performance score for a single inner train / test split.

(e) The *model fit* for the evaluation of the model itself is sometimes also parallelizable (Line 8 and 14). For example, the individual trees of a random forest can be fitted independently and as such are an obvious target for parallelization.

Note that the k_{inner} resampling iterations created in (d) are independent between the n_{batch} HPC evaluations created in (c). Therefore, they form $n_{\text{batch}} \cdot k_{\text{inner}}$ independent jobs that can be executed in parallel.

An HPO problem can now be distributed to the workers in several ways. For example, if one wants to perform a 10-fold (outer) CV of *BO* that proposes one point per tuning iteration, with a budget of 50 HP evaluations done via a 3-fold CV (inner resampling), one can decide to (i) spawn 10 parallel jobs (level [a]) with a long runtime once, or to (ii) spawn three parallel jobs (level [d]) with a short runtime 50 times.

Consider another example with an *ES* running for 50 generations with an offspring size of 20. This translates to 50 tuning iterations with 20 HPC proposals per iteration. If this is evaluated using nested CV, with 10 outer and three inner resampling loops, then the parallelization options are: (i) spawn 10 parallel jobs with a long runtime once (level [a]), (ii) spawn 20 parallel jobs (level [c]) $10 \cdot 50$ times with a medium runtime, (iii) spawn three parallel jobs (level [d]) with a short runtime $10 \cdot 50 \cdot 20$ times, or (iv) spawn $20 \cdot 3 = 60$ parallel jobs (level [d] and [c] combined) with a short runtime $10 \cdot 50$ times.

6.8.2 | Parallelizability

These examples demonstrate that the choice of the parallelization level also depends on the choice of the tuner. Parallelization for RS and GS is so straightforward that it is also called "embarrassingly parallel", since all HPCs to be evaluated can be determined in advance as they have only one (trivial) tuning iteration (level [b]). Algorithms based on iterations over batch proposals (ES, BO, IR, Hyperband) are limited in how many parallel workers they can use and profit from: they can be parallelized up to their (current) batch size, which decreases for HB and IR during a bracket/race. The situation looks very different for BO; by construction, it is a sequential algorithm with a batch size of 1. Multi-point proposals (see Section 4.2.3) must be used to parallelize multiple configurations in each iteration (level [c]). However, parallelization does not scale very well for this level. If possible, a different parallelization level should be chosen to achieve a similar speedup but avoid the problems of multi-point proposal.

Therefore, if a truly large number of parallel resources is available, then the best optimization method may be RS due to its relatively boundless parallelizability and low synchronization overhead. With fewer parallel resources, more efficient algorithms have a larger relative advantage.

6.8.3 | Parallelization tweaks

The more jobs generated, the more workers can be utilized in general, and the better the tuning tends to scale with available computational resources. However, there are some caveats and technical difficulties to consider. First, the process of starting a job, communicating the inputs to the worker, and communicating the results back often comes with considerable overhead, depending on the parallelization backend used. To maximize utilization and minimize overhead, some backends therefore allow chunking multiple jobs together into groups that are calculated sequentially on the workers. Second, whenever a batch of parallel jobs is started, the main process must wait for all of them to finish, which leads to *synchronization overhead* if nodes that finish early are left idling because they wait for longer running jobs to finish. This can be a particular problem for HPO, where it is likely that the jobs have heterogeneous runtimes. For example, fitting a boosting model with 10 iterations will be significantly faster than fitting a boosting model with 10,000 iterations. Unwittingly chunking many jobs together can exacerbate synchronization overhead and lead to a situation where most workers idle and wait for one worker to finish a relatively large chunk of jobs. While there are approaches to mitigate such problems, as briefly discussed in Section 4.2.3 for BO, it is often sufficient—as a rule of thumb—to aim for as many jobs as possible, as long as each individual job has an expected average runtime of ≥ 5 min. Additionally, randomizing the order of jobs can increase the utilization of search strategies such as GS.

Budget and overhead

For an unbiased comparison, it is mandatory that all tuners are granted the same budget. However, if the budget is defined by a fixed number evaluations, it is unclear how to count multi-fidelity evaluations and how to compare them against "regular" evaluations. We recommend comparing evaluations by wall-clock time, and to explicitly report overhead for model inference and point proposal.

Anytime behavior

In practice, it is unclear how long tuners will and should be run. Thus, when benchmarking tuners, the performance of the optimization process should be considered at many stages. To achieve this objective, one should not (only) compare the final performance at the maximum budget, but rather compare the whole optimization traces from initialization to termination. Furthermore, tuning runs must be terminated if the wall-clock time is exceeded, which may leave the tuner in an undefined state and incapable of extracting a final result.

Parallelization

The effect of parallelization must be factored in.[23] While some tuners scale linearly with the number of workers without a degradation in performance (e.g., GS), other tuners (e.g., BO) show diminishing returns from excessive parallelism. A fair comparison w.r.t. a time budget is also hampered by technical aspects like cache interference or difficult to control parallelization of the ML algorithms to tune.

6.9 | Simple sequential approach to model selection

In practical model selection, it is often the case that practitioners care not only about predictive performance, but also about other aspects of the model such as its simplicity and interpretability. If a practitioner considers only a low number of different model classes, then a practical alternative to AutoML on a full pipeline space of different ML models is to construct a manual sequence of preferred models of increasing complexity, to separately tune them, and to compare their performance. An informed choice can then be made about how much additional complexity is acceptable for a certain amount of improved predictive performance. Hastie et al. (2009) suggest the "one-standard-error rule" and use the simplest model within one standard error of the performance of the best model. The specific sequence of learners will differ for each use case but might look something like this: (1) linear model with only a single feature or tree stump, (2) L1/L2 regularized linear model or a CART tree, (3) component-

wise boosting of a GAM, (4) random forest, (5) gradient tree boosting, and (6) full AutoML search space with pipelines. If even more focus is put on predictive performance (multi-level), stacking a diverse set of models can give a final (often rather small) boost (Wolpert, 1992).

7 | RELATED PROBLEMS

Besides HPO, there are many related scientific fields that face very similar problems on an abstract level and nowadays resort to techniques that are very similar to those described in this work. Although detailed coverage of these related areas is out of scope for this paper, we nevertheless briefly summarize and compare them to HPO.

7.1 | Neural architecture search

A specific type of HPO is neural architecture search (NAS) (Elsken et al., 2019), where the task is to find a well-performing architecture of a deep NN for a given data set. Although NAS can also be formulated as an HPO problem (e.g., Zimmer et al., 2021), it is usually approached as a bi-level optimization problem that can be simultaneously solved while training the NN (Liu et al., 2019). Although it is common practice to optimize architecture and HPCs in sequence, recent evidence suggests that they should be jointly optimized (Lindauer & Hutter, 2020; Zela et al., 2018).

7.2 | Algorithm selection and traditional meta-learning

In HPO, we actively *search* for a well-performing HPC through iterative optimization. In the related problem of algorithm selection (Rice, 1976), we usually train a meta-learning model offline to select an optimal element from a finite set of algorithms or HPCs (Bischl et al., 2016). This model can be trained on empirical meta-data of the candidates' performances on a large set of problem instances or data sets. While this allows an instantaneous prediction of a model class and/or a configuration for a new data set without any time investment, these meta-models are rarely sufficiently accurate in practice, and modern approaches use them mainly to warmstart HPO (Feurer, Springenberg, & Hutter, 2015b) in a hybrid manner.

7.3 | Algorithm configuration

Algorithm configuration (AC) is the general task of searching for a well-performing parameter configuration of an arbitrary algorithm for a given, finite, arbitrary set of problem instances. We assume that the performance of a configuration can be quantified by some scoring metric for any given instance (Hutter et al., 2006). Usually, this performance can only be accessed empirically in a black-box manner and comes at the cost of running this algorithm. This is sometimes called offline configuration for a set of instances, and the obtained configuration is then kept fixed for all future problem instances from the same domain. Alternatively, algorithms can also be configured per instance. AC is much broader than HPO, and has a particularly rich history regarding the configuration of discrete optimization solvers (Xu et al., 2008). In HPO, we typically search for a well-performing HPC *on a single data set*, which can be seen as a case of per-instance configuration. The case of finding optimal pipelines or default configurations (Pfisterer et al., 2021) across a larger domain of ML tasks is much closer to traditional AC across sets of instances. The field of AutoML as introduced in Section 5 originally stems from AC and was initially introduced as the *Combined Algorithm Selection and Hyperparameter Optimization* (CASH) problem (Thornton et al., 2013).

7.4 | Dynamic algorithm configuration

In HPO (and AC), we assume that the HPC is chosen *once* for the entire training of an ML model. However, many HPs can actually be adapted while training. A well-known example is the learning rate of an NN optimizer, which might be

adapted via heuristics (Kingma & Ba, 2015) or pre-defined schedules (Loshchilov & Hutter, 2017). However, both are potentially sub-optimal. A more adaptive view on HPO is called dynamic algorithm configuration (Biedenkapp et al., 2020). This allows a policy to be learned (e.g., based on reinforcement learning) for mapping a state of the learner to an HPC, for example, the learning rate (Daniel et al., 2016). We note that dynamic algorithm configuration is a generalization of algorithm selection and configuration (and thus HPO) (Speck et al., 2021).

7.5 | Learning to learn and to optimize

Chen et al. (2017) and Li and Malik (2017) considered methods beyond simple HPO for fixed hyperparameters, and proposed replacing entire components of learners and optimizers. For example, Chen et al. (2017) proposed using an LSTM to learn how to update the weights of an NN. In contrast, Li and Malik (2017) applied reinforcement learning to learn where to sample next in black-box optimization such as HPO. Both of these approaches attempt to learn these meta-learners on diverse sets of applications. While it is already non-trivial to select an HPO algorithm for a given problem, such meta-learning approaches face the even greater challenge of generalizing to new (previously unseen) tasks.

8 | CONCLUSION AND OPEN CHALLENGES

This work has sought to provide an overview of the fundamental concepts and algorithms for HPO in ML. Although HPO can be applied to a single ML algorithm, state-of-the-art systems typically optimize the entire predictive pipeline—including pre-processing, model selection, and post-processing. By using a structured search space, even complex optimization tasks can be efficiently optimized by HPO techniques. This final section now outlines several open challenges in HPO that we deem particularly important.

8.1 | General vs. narrow HPO frameworks

For HPO tools, there is a general trade-off between handling many tasks (such as auto-sklearn; Feurer, Klein, et al., 2015a) and a specializing for few and narrow-focused tasks. The former has the advantage that it can be applied more flexibly, but comes with the disadvantages that (i) it requires more development time to initially set it up and (ii) that the search space is larger such that the efficiency might be sub-optimal compared with a task-specific approach. The latter has the advantage that a more specialized search space can lead to a higher efficiency on a specific task but might not be applicable to a different task. When a specialized tool for an HPO problem can be found, it can often preferred to generalized tools.

8.2 | Interactive HPO

It is still unclear how HPO tools can be fully integrated into the exploratory and prototype-driven workflow of data scientists and ML experts. On the one hand, it is desirable to support these experts in tuning HPs to avoid this tedious and error-prone task. On the other hand, this can lead to lengthy periods of waiting that interrupt the workflow of the data scientist, ranging from a few minutes on smaller data sets and simpler search spaces, or for hours or even days for large-scale data and complex pipelines. Therefore, the open problems remains of how HPO approaches can be designed so that users can monitor progress in an efficient and transparent manner and how to enable interactions with a running HPO tool in case the desired progress or solutions are not achieved.

8.3 | HPO for deep learning and reinforcement learning

For many applications of reinforcement learning and deep learning, especially in the field of natural language processing and large computer vision models, expensive training often prohibits several training runs with different

HPCs. Iterative HPO for such computationally extremely expensive tasks is infeasible even with efficient approaches like BO and multifidelity variants. There are two ways to address this issue. First, some models do not have to be trained from scratch by applying transfer learning or few-shot learning, for example, Finn et al. (2017). This allows cost-effective applications of these models without incredibly expensive training. Using meta-learning techniques for HPO is an option and can also be further improved if gradient-based HPO or NAS is feasible (Elsken et al., 2020; Franceschi et al., 2018). Second, dynamic configuration approaches allow application of HPO on the fly while training. A very prominent example is population-based training (PBT) (Li et al., 2019), which uses a population of training runs with different settings and applies ideas from evolutionary algorithms (especially mutation and tournament selection) from time to time while the model training makes progress. The disadvantage is that this method requires a substantial amount of computational resources for training several models in parallel. This can be reduced by combining PBT with ideas from bandits and BO (Parker-Holder et al., 2020).

8.4 | Overtuning and regularization for HPO

As discussed in Section 4.4, long HPO runs can lead to biased performance estimators, which in the end can also lead to incorrect HPC selection. It seems plausible that this effect is increasingly exacerbated the smaller the data and the less iterations of resampling the user has configured.[24] It seems even more plausible that better testing of results (on even more separate validation data) can mitigate or resolve the issue, but this makes data usage even less efficient in HPO, as we are already operating with 3-ways splits. HPO tools should probably more intelligently control resampling by increasing the number of folds more and more the longer the tuning process runs. However, determining how to ideally set up such a schedule seems to be an under-explored issue.

8.5 | Interpretability of HPO process

Many HPO approaches mainly return a well-performing HPC and leave users without insights into decisions of the optimization process. Not all data scientists trust the outcome of an AutoML system due to the lack of transparency (Drozdal et al., 2020). Consequently, they might not deploy an AutoML model despite all performance gains. In addition, larger performance gains could potentially be generated (especially by avoiding meta-configuration problems of HPO) if a user is presented with a better understanding of the functional landscape of the objective $c(\lambda)$, the sampling evolution of the HPO algorithm, the importance of HPs, or their effects on the resulting performance. Hence, in future work, HPO may need to be combined more often with approaches from interpretable ML and sensitivity analysis.

8.6 | Multi-criteria HPO and model simplicity

Until here, we mainly considered the scenario of having one well-defined metric for predictive performance available to guide HPO. However, in practical applications, there is often an unknown trade-off between multiple metrics at play, even when only considering predictive performance (e.g., consider an imbalanced multiclass task with unknown misclassification costs). Additionally, there often exists an inherent preference towards simple, interpretable, efficient, and sparse solutions. Such solutions are easier to debug, deploy and maintain, as well as assist in overcoming the "last-mile" problem of integrating ML systems in business workflows (Chui et al., 2018). Since the optimal trade-off between predictive performance and simplicity is usually unknown a-priori, an attractive approach is multi-criteria HPO in order to learn the HPCs of all Pareto optimal trade-offs (Binder, Moosbauer, et al., 2020a; Horn & Bischl, 2016). Such multi-criteria approaches introduce a variety of additional challenges for HPO that go beyond the scope of this paper. Recently, model distillation approaches have been proposed to extract a simple(r) model from a complex ensemble (Fakoor et al., 2020).

8.7 | Optimizing for interpretable models and sparseness

A different challenge regarding interpretability is to bias the HPO process towards more interpretable models and return them if their predictive performance is not significantly and/or relevantly worse than that of a less

understandable model. To integrate the search for interpretable models into HPO, it would be beneficial if the interpretability of an ML model could be quantitatively measured. This would allow direct optimization against such a metric (Molnar et al., 2019), for example, using multi-criteria HPO methods.

8.8 | HPO beyond supervised learning

This paper discussed HPO for supervised ML. Most algorithms from other sub-fields of ML, such as clustering (unsupervised) or anomaly detection (semi- or unsupervised), are also configurable by HPs. In these settings, the HPO techniques discussed in this paper can hardly be used effectively, since performance evaluation, especially with a single, well-defined metric, is much less clear. Nevertheless, HPO is ready to be used. With the seminal paper by Bergstra and Bengio (2012), HPO again gained exceptional attention in the last decade. Furthermore, this development sparked tremendous progress both in terms of efficiency and in applicability of HPO. Currently, there are many HPO packages in various programming languages readily available that allow easy application of HPO to a large variety of problems.

AUTHOR CONTRIBUTIONS

Bernd Bischl: Conceptualization (lead); funding acquisition (lead); investigation (lead); methodology (lead); project administration (lead); supervision (lead); writing – original draft (lead); writing – review and editing (lead). **Martin Binder:** Writing – original draft (supporting); writing – review and editing (supporting). **Michel Lang:** Writing – original draft (supporting); writing – review and editing (supporting). **Tobias Pielok:** Writing – original draft (supporting); writing – review and editing (supporting). **Jakob Richter:** Writing – original draft (supporting); writing – review and editing (supporting). **Stefan Coors:** Visualization (lead); writing – original draft (supporting); writing – review and editing (supporting). **Janek Thomas:** Writing – original draft (supporting); writing – review and editing (supporting). **Theresa Ullmann:** Writing – original draft (supporting); writing – review and editing (supporting). **Marc Becker:** Software (lead); writing – review and editing (supporting). **Anne-Laure Boulesteix:** Writing – original draft (supporting); writing – review and editing (supporting). **Difan Deng:** Writing – original draft (supporting); writing – review and editing (supporting). **Marius Lindauer:** Writing – original draft (supporting); writing – review and editing (supporting).

ACKNOWLEDGMENT

We would like to thank Eyke Hüllermeier, Lars Kothoff, Jürgen Branke, and Eduardo C. Garrido-Merchán for their valuable feedback on the manuscript. Open Access funding enabled and organized by Projekt DEAL.

FUNDING INFORMATION

The authors of this work take full responsibilities for its content. This work was supported by the Federal Statistical Office of Germany; the Deutsche Forschungsgemeinschaft (DFG) within the Collaborative Research Center SFB 876, A3; the Research Center "Trustworthy Data Science and Security", one of the Research Alliance centers within the https://uaruhr.de; the German Federal Ministry of Education and Research (BMBF) under Grant No. 01IS18036A; and the Bavarian Ministry for Economic Affairs, Infrastructure, Transport and Technology through the Center for Analytics-Data-Applications (ADA-Center) within the framework of "BAYERN DIGITAL II."

DATA AVAILABILITY STATEMENT

Data sharing is not applicable to this article as no new data were created or analyzed in this study.

ORCID

Bernd Bischl https://orcid.org/0000-0001-6002-6980
Michel Lang https://orcid.org/0000-0001-9754-0393
Jakob Richter https://orcid.org/0000-0003-4481-5554
Stefan Coors https://orcid.org/0000-0002-7465-2146
Theresa Ullmann https://orcid.org/0000-0003-1215-8561
Marc Becker https://orcid.org/0000-0002-8115-0400
Anne-Laure Boulesteix https://orcid.org/0000-0002-2729-0947
Marius Lindauer https://orcid.org/0000-0002-9675-3175

RELATED WIREs ARTICLES

Resampling methods
Evolutionary Algorithms
Hyperparameters and tuning strategies for random forest

ENDNOTES

[1] where the measurement of performance is arguably much less straightforward, especially via a single metric.
[2] More generally, with a slight abuse of notation, the feature vector $\mathbf{x}^{(i)}$ could be taken as a tensor (for example when the feature is an image).
[3] More precisely, $\mathcal{D}$ is an indexed tuple, but we will continue to use common terminology and call it a data *set*.
[4] *Surrogate loss* for the inner loss and *target loss* for the outer loss are also commonly used terminologies.
[5] Note again that optimizing the resampling error will result in biased estimates, which are problematic when reporting the generalization error; use nested CV for this, see Section 4.4.
[6] Either completely at random, or with a probability according to their fitness, the most popular variants being roulette wheel and tournament selection.
[7] In ES-terminology this is called a $(\mu + \lambda)$ elite survival selection, where μ denotes the population size, and λ the number of offspring; other variants like (μ,λ) selection exist.
[8] The self-tuning learner actually also adds new HPs that are the control parameters of the HPO procedure.
[9] But beware of hidden small sizes, e.g., in imbalanced classification scenarios with large n but one or several classes with few samples.
[10] https://github.com/optuna/kurobako
[11] https://github.com/uber/bayesmark
[12] https://www.tensorflow.org/
[13] https://mxnet.apache.org/
[14] https://github.com/google/jax
[15] https://github.com/HIPS/Spearmint
[16] https://scikit-optimize.github.io/
[17] https://github.com/fmfn/BayesianOptimization
[18] https://github.com/GPflow/GPflowOpt
[19] https://github.com/Epistimio/orion
[20] https://github.com/ray-project/ray
[21] https://github.com/salesforce/TransmogrifAI
[22] https://github.com/AxeldeRomblay/MLBox
[23] At least if the budget is defined by available wall-clock time
[24] Issues like imbalance of a classification task and non-standard resampling and evaluation metrics can complicate this matter further.

REFERENCES

Akiba, T., Sano, S., Yanase, T., Ohta, T., & Koyama, M. (2019). Optuna: A next-generation hyperparameter optimization framework. In A. Teredesai, V. Kumar, Y. Li, R. Rosales, E. Terzi, & G. Karypis (Eds.), *KDD '19: Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining* (pp. 2623–2631). ACM.

Andonie, R. (2019). Hyperparameter optimization in learning systems. *Journal of Membrane Computing*, *1*(4), 279–291.

Antonov, I. A., & Saleev, V. (1979). An economic method of computing lp τ-sequences. *USSR Computational Mathematics and Mathematical Physics*, *19*(1), 252–256.

Awad, N. H., Mallik, N., & Hutter, F. (2021). DEHB: Evolutionary hyberband for scalable, robust and efficient hyperparameter optimization. In Z. Zhou (Ed.), *Proceedings of the Thirtieth International Joint Conference on Artificial Intelligence (IJCAI-21)* (pp. 2147–2153).

Balandat, M., Karrer, B., Jiang, D. R., Daulton, S., Letham, B., Wilson, A. G., & Bakshy, E. (2020). BoTorch: A framework for efficient Monte-Carlo Bayesian optimization. In H. Larochelle, M. Ranzato, R. Hadsell, M. F. Balcan, & H. Lin (Eds.), *Advances in Neural Information Processing Systems 33 (NeurIPS 2020)*. Curan Associates, Inc.

Bartz-Beielstein, T., Friese, M., Zaefferer, M., Naujoks, B., Flasch, O., Konen, W., & Koch, P. (2011). Noisy optimization with sequential parameter optimization and optimal computational budget allocation. Proceedings of the 13th Annual Conference Companion on Genetic and Evolutionary Computation, pp. 119–120.

Baydin, A. G., Pearlmutter, B. A., Radul, A. A., & Siskind, J. M. (2018). Automatic differentiation in machine learning: A survey. *Journal of Machine Learning Research*, *18*, 1–43.

Bellman, R. E. (2015). *Adaptive control processes*. Princeton University Press.

Bengio, Y., & Grandvalet, Y. (2004). No unbiased estimator of the variance of k-fold crossvalidation. *Journal of Machine Learning Research*, *5*(Sep), 1089–1105.

Bergmeir, C., Hyndman, R. J., & Koo, B. (2018). A note on the validity of cross-validation for evaluating autoregressive time series prediction. *Computational Statistics & Data Analysis*, *120*, 70–83.

Bergstra, J., Bardenet, R., Bengio, Y., & Kégl, B. (2011). Algorithms for hyper-parameter optimization. In J. Shawe-Taylor, R. Zemel, P. Bartlett, F. Pereira, & K. Weinberger (Eds.), *Advances in Neural Information Processing Systems 24 (NeurIPS 2011)* (pp. 2546–2554). Curan Associates, Inc.

Bergstra, J., & Bengio, Y. (2012). Random search for hyper-parameter optimization. *Journal of Machine Learning Research*, *13*(10), 281–305.

Bergstra, J., Yamins, D., & Cox, D. (2013). Making a science of model search: Hyperparameter optimization in hundreds of dimensions for vision architectures. In S. Dasgupta & D. McAllester (Eds.), *Proceedings of the 30th international conference on machine learning* (pp. 115–123) PMLR. JMLR Inc.

Beyer, H.-G., & Schwefel, H.-P. (2002). Evolution strategies: A comprehensive introduction. *Natural Computing*, *1*, 3–52.

Beyer, H.-G., & Sendhoff, B. (2006). Evolution strategies for robust optimization. *IEEE International Conference on Evolutionary Computation*, *2006*, 1346–1353.

Biedenkapp, A., Bozkurt, H. F., Eimer, T., Hutter, F., & Lindauer, M. (2020). Dynamic algorithm configuration: Foundation of a new meta-algorithmic framework. In G. D. Giacomo, A. Catala, B. Dilkina, M. Milano, S. Barro, A. Bugarin, & J. Lang (Eds.), *ECAI 2020: 24th European conference on artificial intelligence, 29 August–8 September 2020, Santiago de compostela, Spain, August 29–September 8, 2020—including 10th conference on prestigious applications of artificial intelligence (PAIS 2020)* (pp. 427–434). IOS Press.

Binder, M., Moosbauer, J., Thomas, J., & Bischl, B. (2020a). Multi-objective hyperparameter tuning and feature selection using filter ensembles. Proceedings of the 2020 Genetic and Evolutionary Computation Conference, pp. 471–479.

Binder, M., Pfisterer, F., & Bischl, B. (2020b). Collecting empirical data about hyperparameters for data driven AutoML. Proceedings of the 7th ICMLWorkshop on Automated Machine Learning (AutoML 2020).

Binder, M., Pfisterer, F., Lang, M., Schneider, L., Kotthoff, L., & Bischl, B. (2021). mlr3pipelines: Flexible machine learning pipelines in R. *Journal of Machine Learning Research*, *22*(184), 1–7.

Birattari, M., Stützle, T., Paquete, L., & Varrentrapp, K. (2002). A racing algorithm for configuring metaheuristics. Proceedings of the 4th Annual Conference on Genetic and Evolutionary Computation, pp. 11–18.

Birattari, M., Yuan, Z., Balaprakash, P., & Stutzle, T. (2010). F-race and iterated F-race: An overview. In T. Bartz-Beielstein, M. Chiarandini, L. Paquete, & M. Preuss (Eds.), *Experimental methods for the analysis of optimization algorithms* (pp. 311–336). Springer.

Bischl, B., Mersmann, O., Trautmann, H., & Weihs, C. (2012). Resampling methods for meta- model validation with recommendations for evolutionary computation. *Evolutionary Computation*, *20*(2), 249–275.

Bischl, B., Kerschke, P., Kotthoff, L., Lindauer, M., Malitsky, Y., Fréchette, A., Hoos, H. H., Hutter, F., Leyton-Brown, K., Tierney, K., & Vanschoren, J. (2016). Aslib: A benchmark library for algorithm selection. *Artificial Intelligence*, *237*, 41–58.

Bischl, B., Wessing, S., Bauer, N., Friedrichs, K., & Weihs, C. (2014). MOI-MBO: Multiobjective infill for parallel model-based optimization. Learning and Intelligent Optimization Conference, pp. 173–186.

Bossek, J. (2017). ecr 2.0: A modular framework for evolutionary computation in R. Proceedings of the Genetic and Evolutionary Computation Conference Companion, pp. 1187–1193.

Bossek, J., Doerr, C., & Kerschke, P. (2020). Initial design strategies and their effects on sequential model-based optimization: An exploratory case study based on bbob. Proceedings of the 2020 Genetic and Evolutionary Computation Conference, pp. 778–786.

Boulesteix, A.-L., Strobl, C., Augustin, T., & Daumer, M. (2008). Evaluating microarray-based classifiers: An overview. *Cancer Informatics*, *6*, 77–97.

Chen, C.-H. (1995). An effective approach to smartly allocate computing budget for discrete event simulation. Proceedings of 1995 34th IEEE Conference on Decision and Control, 3, pp. 2598–2603.

Chen, Y., Hoffman, M. W., Colmenarejo, S. G., Denil, M., Lillicrap, T. P., Botvinick, M., & de Freitas, N. (2017). Learning to learn without gradient descent by gradient descent. In D. Precup & Y. W. Teh (Eds.), *Proceedings of the 34th international conference on machine learning, ICML 2017, Sydney, nsw, Australia, 6–11 August 2017* (pp. 748–756) PMLR. JMLR Inc.

Chevalier, C., & Ginsbourger, D. (2013). Fast computation of the multi-points expected improvement with applications in batch selection. *Learning and Intelligent Optimization*, 7997, 59–69.

Chui, M., Manyika, J., Miremadi, M., Henke, N., Chung, R., Nel, P., & Malhotra, S. (2018). *Notes from the ai frontier: Insights from hundreds of use cases*. McKinsey Global Institute.

Coello Coello, C. A., Lamont, G. B., & Van Veldhuizen, D. A. (2007). *Evolutionary algorithms for solving multi-objective problems*. Springer.

Daniel, C., Taylor, J., & Nowozin, S. (2016). Learning step size controllers for robust neural network training. In D. Schuurmans & M. P. Wellman (Eds.), *Proceedings of the thirtieth AAAI conference on artificial intelligence, February 12–17, 2016, phoenix, Arizona, USA (pp. 1519–1525)* (Vol. 30). AAAI Press.

de Ath, G., Everson, R. M., Rahat, A. A., & Fieldsend, J. E. (2021). Greed is good: Exploration and exploitation trade-offs in Bayesian optimization. *ACM Transactions on Evolutionary Learning and Optimization*, *1*(1), 1–22.

de Matthews, A. G., van der Wilk, M., Nickson, T., Fujii, K., Boukouvalas, A., León-Villagrá, P., Ghahramani, Z., & Hensman, J. (2017). GPflow: A Gaussian process library using TensorFlow. *Journal of Machine Learning Research*, *18*(40), 1–6.

Dobbin, K. K., & Simon, R. M. (2011). Optimally splitting cases for training and testing high dimensional classifiers. *BMC Medical Genomics*, *4*(1), 1–8.

Drozdal, J., Weisz, J. D., Wang, D., Dass, G., Yao, B., Zhao, C., Muller, M. J., Ju, L., & Su, H. (2020). Trust in automl: Exploring information needs for establishing trust in automated machine learning systems. In F. Paternò, N. Oliver, C. Conati, L. D. Spano, & N. Tintarev (Eds.), *IUI'20: 25th international conference on intelligent user interfaces, Cagliari, Italy, March 17–20, 2020* (pp. 297–307). ACM.

Eggensperger, K., Lindauer, M., & Hutter, F. (2019). Pitfalls and best practices in algorithm configuration. *Journal of Artificial Intelligence Research*, *64*, 861–893.

Eggensperger, K., Feurer, M., Hutter, F., Bergstra, J., Snoek, J., Hoos, H., & Leyton-Brown, K. (2013). Towards an empirical foundation for assessing Bayesian optimization of hyperparameters. NIPS workshop on Bayesian Optimization in Theory and Practice, p. 3.

Eggensperger, K., Müller, P., Mallik, N., Feurer, M., Sass, R., Klein, A., Awad, N. H., Lindauer, M., & Hutter, F. (2021). Hpobench: A collection of reproducible multi-fidelity benchmark problems for HPO. In J. Vanschoren & S. Yeung (Eds.), *Proceedings of the neural information processing systems track on datasets and benchmarks*. Curan Associates, Inc.

Elsken, T., Staffler, B., Metzen, J. H., & Hutter, F. (2020). Meta-learning of neural architectures for few-shot learning. *2020* IEEE/CVF Conference on Computer Vision and Pattern Recognition, CVPR 2020, Seattle, WA, USA, June 13–19, 2020, pp. 12362–12372.

Elsken, T., Metzen, J. H., & Hutter, F. (2019). Neural architecture search: A survey. *Journal of Machine Learning Research*, *20*(55), 1–21.

Erickson, N., Mueller, J., Shirkov, A., Zhang, H., Larroy, P., Li, M., Smola, A., Klein, A., Tiao, L., Lienart, T., Archambeau, C., Seeger, M., Perrone, V., Donini, M., Zafar, B. M., Schmucker, R., Kenthapadi, K., & Archambeau, C. (2020). *Autogluon-tabular: Robust and accurate automl for structured data*. 7th ICML Workshop on Automated Machine Learning (AutoML 2020).

Eriksson, D., Pearce, M., Gardner, J., Turner, R. D., & Poloczek, M. (2019). Scalable global optimization via local Bayesian optimization. *Advances in Neural Information Processing Systems*, *32*, 5496–5507.

Escalante, H. J., Montes, M., & Sucar, L. E. (2009). Particle swarm model selection. *Journal of Machine Learning Research*, *10*(2), 405–440.

Estévez-Velarde, S., Gutiérrez, Y., Almeida-Cruz, Y., & Montoyo, A. (2020). General-purpose hierarchical optimisation of machine learning pipelines with grammatical evolution. *Information Sciences*, *543*, 58–71.

Fakoor, R., Mueller, J. W., Erickson, N., Chaudhari, P., & Smola, A. J. (2020). Fast, accurate, and simple models for tabular data via augmented distillation. *Advances in Neural Information Processing Systems*, *33*, 8671–8681.

Falkner, S., Klein, A., & Hutter, F. (2018). BOHB: Robust and efficient hyperparameter optimization at scale, *80*, 1437–1446.

Fernández-Delgado, M., Cernadas, E., Barro, S., & Amorim, D. (2014). Do we need hundreds of classifiers to solve real world classification problems? *The Journal of Machine Learning Research*, *15*(1), 3133–3181.

Feurer, M., & Hutter, F. (2019). Hyperparameter optimization. In F. Hutter, L. Kotthoff, & J. Vanschoren (Eds.), *Automl: Methods, systems, challenges* (pp. 3–33). Springer.

Feurer, M., Klein, A., Eggensperger, K., Springenberg, J. T., Blum, M., & Hutter, F. (2015a). Efficient and robust automated machine learning. In *Proceedings of the 28th international conference on neural information processing systems: Volume 2* (pp. 2755–2763). Curan Associates, Inc.

Feurer, M., Springenberg, J. T., & Hutter, F. (2015b). Initializing Bayesian hyperparameter optimization via meta-learning. In B. Bonet & S. Koenig (Eds.), *Proceedings of the twentyninth AAAI conference on artificial intelligence, January 25–30, 2015, Austin, Texas, USA (pp. 1128–1135)* (Vol. 29). AAAI Press.

Finn, C., Abbeel, P., & Levine, S. (2017). Model-agnostic meta-learning for fast adaptation of deep networks. In D. Precup & Y. Teh (Eds.), *Proceedings of the 34th international conference on machine learning, ICML 2017* (Vol. 70, pp. 1126–1135). JMLR Inc.

Franceschi, L., Donini, M., Frasconi, P., & Pontil, M. (2017). Forward and reverse gradient-based Hyperparameter optimization. In D. Precup & Y. Teh (Eds.), *Proceedings of the 34th international conference on machine learning, ICML 2017* (Vol. 70, pp. 1165–1173). JMLR Inc.

Franceschi, L., Frasconi, P., Salzo, S., Grazzi, R., & Pontil, M. (2018). Bilevel programming for hyperparameter optimization and meta-learning. In J. Dy & A. Krause (Eds.), *Proceedings of the 35th international conference on machine learning, ICML 2018* (Vol. 80, pp. 1568–1577). JMLR Inc.

Gardner, J. R., Pleiss, G., Bindel, D., Weinberger, K. Q., & Wilson, A. G. (2018). Gpytorch: Blackbox matrix-matrix gaussian process inference with gpu acceleration. Advances in Neural Information Processing Systems.

Garrido-Merchán, E. C., & Hernández-Lobato, D. (2020). Dealing with categorical and integervalued variables in Bayesian optimization with Gaussian processes. *Neurocomputing*, *380*, 20–35.

Gelbart, M., Snoek, J., & Adams, R. (2014). Bayesian optimization with unknown constraints, 250–258.

Gijsbers, P., Pfisterer, F., van Rijn, J., Bischl, B., & Vanschoren, J. (2021). *Meta-learning for symbolic hyperparameter defaults. Proceedings of the Genetic and Evolutionary Computation Conference Companion,* pp. 151–152.

Ginsbourger, D., Le Riche, R., & Carraro, L. (2010). Kriging is well-suited to parallelize optimization. In Y. Tenne & C.-K. Goh (Eds.), *Computational intelligence in expensive optimization problems* (pp. 131–162). Springer.

Guyon, I., Saffari, A., Dror, G., & Cawley, G. (2010). Model selection: Beyond the Bayesian/frequentist divide. *The Journal of Machine Learning Research*, *11*(3), 61–87.

Hancock, J. T., & Khoshgoftaar, T. M. (2020). Survey on categorical data for neural networks. *Journal of Big Data*, *7*(1), 28.

Hansen, N., Auger, A., Ros, R., Mersmann, O., Tušar, T., & Brockhoff, D. (2021). COCO: A platform for comparing continuous optimizers in a black-box setting. *Optimization Methods and Software*, *36*(1), 114–144.

Hastie, T., Tibshirani, R., & Friedman, J. (2009). *The elements of statistical learning: Data mining, inference, and prediction*. Springer Science & Business Media.

He, X., Zhao, K., & Chu, X. (2021). Automl: A survey of the state-of-the-art. *Knowledge-Based Systems*, *212*, 106622.

Hennig, P., & Schuler, C. J. (2012). Entropy search for information-efficient global optimization. *Journal of Machine Learning Research*, *13*(6), 1809–1837.

Hernández-Lobato, D., Hernandez-Lobato, J., Shah, A., & Adams, R. (2016). Predictive entropy search for multi-objective Bayesian optimizationIn M. Balcan, & K. Weinberger (Eds.),oof Collator inserted content on: 11 Jan, 02:49 AM" class="new" id="43a711be-0288-4975-b47c-aee092509355" updatedon="11 Jan, 02:49 AM"> In M. Balcan, & K. Weinberger (Eds.), *Proceedings of the 33rd international conference on machine learning, ICML 2016* (pp. 1492–1501). JMLR Inc.

Horn, D., & Bischl, B. (2016). Multi-objective parameter configuration of machine learning algorithms using model-based optimization. *IEEE symposium series on computational intelligence (SSCI)*, *2016*, 1–8.

Hornung, R., Bernau, C., Truntzer, C., Wilson, R., Stadler, T., & Boulesteix, A.-L. (2015). A measure of the impact of CV incompleteness on prediction error estimation with application to PCA and normalization. *BMC Medical Research Methodology*, *15*, 95.

Hutter, F., Hoos, H., & Leyton-Brown, K. (2011). Sequential model-based optimization for general algorithm configuration, *6683*, 507–523.

Hutter, F., Kotthoff, L., & Vanschoren, J. (Eds.). (2019). *Automated machine learning: Methods, systems, challenges*. Springer.

Hutter, F., Hamadi, Y., Hoos, H. H., & Leyton-Brown, K. (2006). Performance prediction and automated tuning of randomized and parametric algorithms. International Conference on Principles and Practice of Constraint Programming, pp. 213–228.

Hutter, F., Hoos, H. H., & Leyton-Brown, K. (2012). Parallel algorithm configuration. In Y. Hamadi & M. Schoenauer (Eds.), *Learning and intelligent optimization* (pp. 55–70). Springer.

Hutter, F., Hoos, H. H., Leyton-Brown, K., & Stützle, T. (2009). ParamILS: An automatic algorithm configuration framework. *Journal of Artificial Intelligence Research*, *36*, 267–306.

Hvarfner, C., Stoll, D., Souza, A., Lindauer, M., Hutter, F., & Nardi, L. (2022). piBO: Augmenting acquisition functions with user beliefs for Bayesian optimization. In *10th International Conference on Learning Representations, ICLR'22* (pp. 1–30). OpenReview.

Jalali, H., van Nieuwenhuyse, I., & Picheny, V. (2017). Comparison of kriging-based algorithms for simulation optimization with heterogeneous noise. *European Journal of Operational Research*, *261*(1), 279–301.

Jamieson, K., & Talwalkar, A. (2016). Non-stochastic best arm identification and hyperparameter optimization. Proceedings of the 19th International Con-ference on Artificial Intelligence and Statistics (AISTATS), pp. 240–248.

Japkowicz, N., & Shah, M. (2011). *Evaluating learning algorithms: A classification perspective*. Cambridge University Press.

Jasrasaria, D., & Pyzer-Knapp, E. O. (2018). Dynamic control of explore/exploit trade-off in Bayesian optimization. *Advances in Intelligent Systems and Computing*, *858*, 1–15.

Jin, H., Song, Q., & Hu, X. (2019). Auto-keras: An efficient neural architecture search system. Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining, pp. 1946–1956.

Jones, D. R. (2001). A taxonomy of global optimization methods based on response surfaces. *Journal of Global Optimization*, *21*(4), 345–383.

Jones, D. R. (2009). Direct global optimization algorithm. *Encyclopedia of Optimization*, *1*(1), 431–440.

Jones, D. R., Schonlau, M., & Welch, W. J. (1998). Efficient global optimization of expensive black-box functions. *Journal of Global Optimization*, *13*(4), 455–492.

Kandasamy, K., Schneider, J., & Póczos, B. (2015). High dimensional Bayesian optimisation and bandits via additive models, *37*, 295–304.

Kandasamy, K., Vysyaraju, K. R., Neiswanger, W., Paria, B., Collins, C. R., Schneider, J., Poczos, B., & Xing, E. P. (2020). Tuning hyperparameters without grad students: Scalable and robust Bayesian optimisation with dragonfly. *Journal of Machine Learning Research*, *21*(81), 1–27.

Khalid, R., & Javaid, N. (2020). A survey on hyperparameters optimization algorithms of forecasting models in smart grid. *Sustainable Cities and Society*, *61*, 102275.

Kingma, D. P., & Ba, J. (2015). *Adam: A method for stochastic optimization*. OpenReview.

Klein, A., Dai, Z., Hutter, F., Lawrence, N., & Gonzalez, J. (2019). Meta-surrogate benchmarking for hyperparameter optimization. *Advances in Neural Information Processing Systems*, *32*, 6270–6280.

Klein, A., Falkner, S., Bartels, S., Hennig, P., & Hutter, F. (2017). Fast Bayesian optimization of machine learning hyperparameters on large datasets. In A. Singh & J. Zhu (Eds.), *Proceedings of the 20th international conference on artificial intelligence and statistics* (pp. 528–536) PMLR.

Kohavi, R. (1995). A study of cross-validation and bootstrap for accuracy estimation and model selection. *International Joint Conference on Artificial Intelligence*, 2, 1137–1143.

Kuhn, M. (2008). Building predictive models in R using the caret package. *Journal of Statistical Software*, *28*(5), 1–26.

Lang, M., Kotthaus, H., Marwedel, P., Weihs, C., Rahnenführer, J., & Bischl, B. (2015). Automatic model selection for high-dimensional survival analysis. *Journal of Statistical Computation and Simulation*, *85*(1), 62–76.

Lang, M., Binder, M., Richter, J., Schratz, P., Pfisterer, F., Coors, S., Au, Q., Casalicchio, G., Kotthoff, L., & Bischl, B. (2019). mlr3: A modern object-oriented machine learning framework in R. *Journal of Open Source Software*, *4*(44), 1903.

Larrañaga, P., & Lozano, J. A. (2001). *Estimation of distribution algorithms: A new tool for evolutionary computation* (Vol. 2). Springer Science & Business Media.

LeDell, E., & Poirier, S. (2020). H2O AutoML: Scalable automatic machine learning. 7th ICML Workshop on Automated Machine Learning (AutoML).

Li, A., Spyra, O., Perel, S., Dalibard, V., Jaderberg, M., Gu, C., Budden, D., Harley, T., & Gupta, P. (2019). A generalized framework for population based training. Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining, KDD 2019, Anchorage, AK, USA, August 4–8, 2019, pp. 1791–1799.

Li, K., & Malik, J. (2017). Learning to optimize. 5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24–26, 2017, Conference Track Proceedings.

Li, L., Jamieson, K., DeSalvo, G., Rostamizadeh, A., & Talwalkar, A. (2018). Hyperband: A novel bandit-based approach to hyperparameter optimization. *Journal of Machine Learning Research*, *18*(185), 1–52.

Li, R., Emmerich, M. T., Eggermont, J., Bäck, T., Schütz, M., Dijkstra, J., & Reiber, J. H. (2013). Mixed integer evolution strategies for parameter optimization. *Evolutionary Computation*, *21*(1), 29–64.

Li, Y., Shen, Y., Zhang, W., Chen, Y., Jiang, H., Liu, M., Jiang, J., Gao, J., Wu, W., Yang, Z., Zhang, C., & Cui, B. (2021). Openbox: A generalized black-box optimization service. *Proceedings of the 27th ACM SIGKDD Conference on Knowledge Discovery & Data Mining*, pp. 3209–3219. ACM.

Lindauer, M., Eggensperger, K., Feurer, M., Biedenkapp, A., Deng, D., Benjamins, C., Ruhkopf, T., Sass, R., & Hutter, F. (2022). SMAC3: A versatile Bayesian optimization package for hyperparameter optimization. *Journal of Machine Learning Research*, *23*(54), 1–9.

Lindauer, M., & Hutter, F. (2018). Warmstarting of model-based algorithm configuration. *Proceedings of the AAAI Conference on Artificial Intelligence*, *32*(1), 1355–1362.

Lindauer, M., & Hutter, F. (2020). Best practices for scientific research on neural architecture search. *Journal of Machine Learning Research*, *21*(243), 1–18.

Liu, H., Simonyan, K., & Yang, Y. (2019). DARTS: differentiable architecture search. 7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6–9, 2019.

López-Ibáñez, M., Dubois-Lacoste, J., Pérez Cáceres, L., Stützle, T., & Birattari, M. (2016). The irace package: Iterated racing for automatic algorithm configuration. *Operations Research Perspectives*, *3*, 43–58.

Lorraine, J., Vicol, P., & Duvenaud, D. (2020). Optimizing millions of hyperparameters by implicit differentiation. *International Conference on Artificial Intelligence and Statistics*, *108*, 1540–1552.

Loshchilov, I., & Hutter, F. (2017). SGDR: stochastic gradient descent with warm restarts. 5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24–26, 2017, Conference Track Proceedings.

Maclaurin, D., Duvenaud, D., & Adams, R. (2015). Gradient-based hyperparameter optimization through reversible learning. In F. Bach & D. Blei (Eds.), *Proceedings of the 32nd international conference on machine learning,* ICML 2015 (Vol. 37, pp. 2113–2122). JMLR, Inc.

Maron, O., & Moore, A. W. (1994). Hoeffding races: Accelerating model selection search for classification and function approximation. *Advances in Neural Information Processing Systems*, *6*, 59–66.

McIntire, M., Ratner, D., & Ermon, S. (2016). Sparse gaussian processes for Bayesian optimization. In A. Ihler, & D. Janzing (Eds.), *Proceedings of the Thirty-Second Conference on Uncertainty in Artificial Intelligence*, 517–526. AUAI Press.

McKay, M., Beckman, R., & Conover, W. (1979). Comparison of three methods for selecting values of input variables in the analysis of output from a computer code. *Technometrics*, *21*(2), 239–245.

Mebane, W. R., Jr., & Sekhon, J. S. (2011). Genetic optimization using derivatives: The rgenoud package for R. *Journal of Statistical Software*, *42*(11), 1–26.

Mohr, F., Wever, M., & Hüllermeier, E. (2018). ML-plan: Automated machine learning via hierarchical planning. *Machine Learning*, *107*(8–10), 1495–1515.

Molinaro, A. M., Simon, R., & Pfeiffer, R. M. (2005). Prediction error estimation: A comparison of resampling methods. *Bioinformatics*, *21*(15), 3301–3307.

Molnar, C., Casalicchio, G., & Bischl, B. (2019). Quantifying model complexity via functional decomposition for better post-hoc interpretability. *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*, 193–204.

Moosbauer, J., Binder, M., Schneider, L., Pfisterer, F., Becker, M., Lang, M., Kotthoff, L., & Bischl, B. (2022). Automated Benchmark‐Driven Design and Explanation of Hyperparameter Optimizers. *IEEE Transactions on Evolutionary Computation*, *26*(6), 1336–1350.

Mullen, K., Ardia, D., Gil, D., Windover, D., & Cline, J. (2011). DEoptim: An R package for global optimization by differential evolution. *Journal of Statistical Software*, *40*(6), 1–26.

Nardi, L., Koeplinger, D., & Olukotun, K. (2019). Practical design space exploration. 2019 IEEE 27th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS), pp. 347–358.

Nayebi, A., Munteanu, A., & Poloczek, M. (2019). A framework for Bayesian optimization in embedded subspaces. In K. Chaudhuri & R. Salakhutdinov (Eds.), *Proceedings of the 36th international conference on machine learning (pp. 4752–4761)* PMLR. JMLR Inc.

Ng, A. Y. (1997). Preventing "overfitting" of cross-validation data. *ICML*, *97*, 245–253.

Oh, C., Gavves, E., & Welling, M. (2018). BOCK: Bayesian optimization with cylindrical kernels. In J. Dy & A. Krause (Eds.), *Proceedings of the 35th international conference on machine learning*, ICML 2018 (Vol. 80, pp. 3865–3874). JMLR Inc.

Olson, R., Bartley, N., Urbanowicz, R., & Moore, J. (2016). Evaluation of a tree-based pipeline optimization tool for automating data science (T. Friedrich, Ed.), pp. 485–492.

Parker-Holder, J., Nguyen, V., & Roberts, S. J. (2020). Provably efficient online hyperparameter optimization with population-based bandits. Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems.

Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., Desmaison, A., Kopf, A., Yang, E., DeVito, Z., Raison, M., Tejani, A., Chilamkurthy, S., Steiner, B., Fang, L., ... Chintala, S. (2019). PyTorch: An imperative style, high-performance deep learning library. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, & R. Garnett (Eds.), *Proceedings of the 32nd International Conference on Advances in Neural Information Processing Systems* (pp. 8024–8035).

Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., Vanderplas, J., Passos, A., Cournapeau, D., Brucher, M., Perrot, M., & Duchesnay, E. (2011). Scikit-learn: Machine learning in python. *Journal of Machine Learning Research*, *12*, 2825–2830.

Perrone, V., Jenatton, R., Seeger, M., & Archambeau, C. (2018). Scalable hyperparameter transfer learning. In S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, & R. Garnett (Eds.), *Proceedings of the 31st International Conference on Advances in Neural Information Processing Systems* (pp. 12751–12761). Curan Associates, Inc.

Perrone, V., Shen, H., Seeger, M. W., Archambeau, C., & Jenatton, R. (2019). Learning search spaces for Bayesian optimization: Another view of hyperparameter transfer learning. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, & R. Garnett (Eds.), *Advances in neural information processing systems*. Curran Associates, Inc.

Pfisterer, F., Schneider, L., Moosbauer, J., Binder, M., & Bischl, B. (2022). YAHPO gym: An efficient multi-objective multi-fidelity benchmark for hyperparameter optimization. First conference on automated machine learning (Main Track).

Pfisterer, F., van Rijn, J. N., Probst, P., Müller, A. C., & Bischl, B. (2021). Learning multiple defaults for machine learning algorithms. In F. Chicano (Ed.), *Proceedings of the Genetic and Evolutionary Computation Conference Companion* (pp. 241–242). ACM.

Picheny, V., Wagner, T., & Ginsbourger, D. (2013). A benchmark of kriging-based infill criteria for noisy optimization. *Structural and Multidisciplinary Optimization*, *48*(3), 607–626.

Pineda Arango, S., Jomaa, H. S., Wistuba, M., & Grabocka, J. (2021). HPO-B: A large-scale reproducible benchmark for black-box HPO based on OpenML. In J. Vanschoren & S. Yeung (Eds.), *Proceedings of the neural information processing systems track on datasets and benchmarks*. Curan Associates, Inc.

Probst, P., Boulesteix, A.-L., & Bischl, B. (2019). Tunability: Importance of hyperparameters of machine learning algorithms. *Journal of Machine Learning Research*, *20*(53), 1–32.

Quinlan, J., & Cameron-Jones, R. (1995). Oversearching and layered search in empirical learning. *Breast Cancer*, *286*, 2–7.

Rasmussen, C. E., & Williams, C. K. I. (2006). *Gaussian processes for machine learning*. MIT Press.

Real, E., Aggarwal, A., Huang, Y., & Le, Q. V. (2019). Regularized evolution for image classifier architecture search. *Proceedings of the AAAI Conference on Artificial Intelligence*, *33*, 4780–4789.

Rice, J. R. (1976). The algorithm selection problem. *Advances in Computers*, *15*, 65–118.

Rocklin, M. (2015). Dask: Parallel computation with blocked algorithms and task scheduling. Proceedings of the 14th python in science conference, pp. 130–136.

Rodgers, D. P. (1985). Improvements in multiprocessor system design. *ACM SIGARCH Computer Architecture News*, *13*(3), 225–231.

Roustant, O., Ginsbourger, D., & Deville, Y. (2012). DiceKriging, DiceOptim: Two R packages for the analysis of computer experiments by kriging-based metamodeling and optimization. *Journal of Statistical Software*, *51*, 1–15.

Salinas, D., Seeger, M., Klein, A., Perrone, V., Wistuba, M., & Archambeau, C. (2022). Syne tune: A library for large scale hyperparameter tuning and reproducible research. First Conference on Automated Machine Learning (Main Track). OpenReview.

Sasena, M. J., Papalambros, P., & Goovaerts, P. (2002). Exploration of metamodeling sampling criteria for constrained global optimization. *Engineering Optimization*, *34*(3), 263–278.

Scott, W., Frazier, P., & Powell, W. (2011). The correlated knowledge gradient for simulation optimization of continuous parameters using gaussian process regression. *SIAM Journal on Optimization*, *21*(3), 996–1026.

Šehić, K., Gramfort, A., Salmon, J., & Nardi, L. (2022). LassoBench: A high-dimensional hyperparameter optimization benchmark suite for lasso. First conference on automated machine learning (Main Track).

Sekhon, J. S., & Mebane, W. R. (1998). Genetic optimization using derivatives. *Political Analysis*, *7*, 187–210.

Sexton, J., & Laake, P. (2009). Standard errors for bagged and random forest estimators. *Computational Statistics & Data Analysis*, *53*(3), 801–811.

Sheng, V. S., & Ling, C. X. (2006). Thresholding for making classifiers cost-sensitive. AAAI'06: Proceedings of the 21st national conference on artificial intelligence, vol. 6, pp. 476–481.

Simon, R. (2007). Resampling strategies for model assessment and selection. In W. Dubitzky, M. Granzow, & D. Berrar (Eds.), *Fundamentals of data mining in genomics and proteomics* (pp. 173–186). Springer.

Snoek, J., Larochelle, H., & Adams, R. (2012). Practical Bayesian optimization of machine learning algorithms. In P. Bartlett, F. Pereira, C. Burges, L. Bottou, & K. Weinberger (Eds.), *Advances in Neural Information Processing Systems 25 (NIPS 2012)* (pp. 2960–2968). Curran Associates, Inc.

Snoek, J., Swersky, K., Zemel, R., & Adams, R. (2014). Input warping for Bayesian optimization of non-stationary functions. In E. Xing & T. Jebara (Eds.), *Proceedings of the 31st International Conference on Machine Learning (ICML '14)* (pp. 1674–1682). Omnipress.

Snoek, J., Rippel, O., Swersky, K., Kiros, R., Satish, N., Sundaram, N., Patwary, M. M. A., Prabhat, M., & Adams, R. P. (2015). Scalable Bayesian optimization using deep neural networks. In F. Bach & D. Blei (Eds.), *Proceedings of the 32nd international conference on machine learning, ICML 2015* (Vol. 37, pp. 2171–2180). JMLR, Inc.

Soydaner, D. (2020). A comparison of optimization algorithms for deep learning. *International Journal of Pattern Recognition and Artificial Intelligence*, *34*(13), 2052013.

Speck, D., Biedenkapp, A., Hutter, F., Mattmüller, R., & Lindauer, M. (2021). Learning heuristic selection with dynamic algorithm configuration. Proceedings of the 31st International Conference on Automated Planning and Scheduling (ICAPS'21).

Srinivas, N., Krause, A., Kakade, S., & Seeger, M. (2010). Gaussian process optimization in the bandit setting: No regret and experimental design. Proceedings of the 27th International Conference on Machine Learning, pp. 1015–1022.

Stander, N., & Craig, K. (2002). On the robustness of a simple domain reduction scheme for simulation-based optimization. *Engineering Computations*, *19*, 431–450.

Swersky, K., Snoek, J., & Adams, R. (2013). Multi-task Bayesian optimization. In C. Burges, L. Bottou, M. Welling, Z. Ghahramani, & K. Weinberger (Eds.), *Advances in Neural Information Processing Systems 26 (NIPS 2013)* (pp. 2004–2012). Curran Associates, Inc.

Swersky, K., Duvenaud, D., Snoek, J., Hutter, F., & Osborne, M. A. (2014). Raiders of the lost architecture: Kernels for Bayesian optimization in conditional parameter spaces. NeurIPS workshop on Bayesian Optimization in Theory and Practice.

Thornton, C., Hutter, F., Hoos, H., & Leyton-Brown, K. (2013). Auto-WEKA:Combined selection and hyperparameter optimization of classification algorithms. In I. Dhillon, Y. Koren, R. Ghani, T. Senator, P. Bradley, R. Parekh, J. He, R. Grossman, & R. Uthurusamy (Eds.), *Proceedings of the 19th ACM SIGKDD international conference on Knowledge discovery and data mining* (pp. 847–855). ACM.

Turner, R., Eriksson, D., McCourt, M., Kiili, J., Laaksonen, E., Xu, Z., & Guyon, I. (2021). Bayesian optimization is superior to random search for machine learning hyperparameter tuning: Analysis of the black-box optimization challenge 2020. In H. Larochelle, M. Ranzato, R. Hadsell, M. Balcan, & H. Lin (Eds.), *Proceedings of the NeurIPS 2020 Competition and Demonstration Track* (pp. 3–26). JMLR, Inc.

van der Laan, M. J., Polley, E. C., & Hubbard, A. E. (2007). Super learner. *Statistical Applications in Genetics and Molecular Biology*, *6*(1), 25.

Van Rijn, J. N., Bischl, B., Torgo, L., Gao, B., Umaashankar, V., Fischer, S., Winter, P., Wiswedel, B., Berthold, M. R., & Vanschoren, J. (2013). OpenML: A collaborative science platform. In Blockeel, H., Kersting, K., Nijssen, S., & F. Železný (Eds.), *Machine Learning and Knowledge Discovery in Databases (ECML/PKDD 2013)* (Vol. 8190, pp. 645–649).

van Rijn, J. N., & Hutter, F. (2018). Hyperparameter importance across datasets. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining* (pp. 2367–2376). Association for Computing Machinery (ACM).

Vanschoren, J., van Rijn, J., Bischl, B., & Torgo, L. (2014). OpenML: Networked science in machine learning. *SIGKDD Explorations*, *15*(2), 49–60.

Wang, Z., Hutter, F., Zoghi, M., Matheson, D., & de Feitas, N. (2016). Bayesian optimization in a billion dimensions via random embeddings. *Journal of Artificial Intelligence Research*, *55*, 361–387.

White, C., Neiswanger, W., & Savani, Y. (2021). Bananas: Bayesian optimization with neural architectures for neural architecture search. Proceedings of the AAAI Conference on Artificial Intelligence.

Wistuba, M., Schilling, N., & Schmidt-Thieme, L. (2015a). Machine Learning and Knowledge Discovery in Databases (ECML/PKDD 2015). In A. Appice, P. Rodrigues, V. Costa, J. Gama, A. Jorge, & C. Soares (Eds.), *Hyperparameter search space pruning: A new component for sequential model-based hyperparameter optimization* (Vol. 9285, pp. 104–119). Cham: Springer.

Wistuba, M., Schilling, N., & Schmidt-Thieme, L. (2015b). *Learning hyperparameter optimization initializations.* 2015 IEEE International Conference on Data Science and Advanced Analytics (DSAA) (pp. 1–10). IEEE.

Wistuba, M., & Grabocka, J. (2021). Few-shot Bayesian optimization with deep kernel surrogates. 9th *International Conference on Learning Representations, ICLR'21*. OpenReview.

Wolpert, D. H. (1992). Stacked generalization. *Neural networks*, *5*(2), 241–259.

Xu, L., Hutter, F., Hoos, H. H., & Leyton-Brown, K. (2008). Satzilla: Portfolio-based algorithm selection for sat. *Journal of Artificial Intelligence Research*, *32*, 565–606.

Yang, L., & Shami, A. (2020). On hyperparameter optimization of machine learning algorithms: Theory and practice. *Neurocomputing*, *415*, 295–316.

Zela, A., Klein, A., Falkner, S., & Hutter, F. (2018). Towards automated deep learning: Efficient joint neural architecture and hyperparameter search. ICML 2018 AutoML Workshop.

Zhang, Z., Wang, X., & Zhu, W. (2021). Automated machine learning on graphs: A survey. In Z. Zhou (Ed.), *Proceedings of the thirtieth international joint conference on artificial intelligence, IJCAI 2021, virtual event/Montreal, Canada, 19–27 August 2021* (pp. 4704–4712).

Zheng, J., Li, Z., Gao, L., & Jiang, G. (2016). A parameterized lower confidence bounding scheme for adaptive metamodel-based design optimization. *Engineering Computations*, *33*, 2165–2184.

Zimmer, L., Lindauer, M., & Hutter, F. (2021). Auto-pytorch: Multi-fidelity metalearning for efficient and robust autodl. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, *43*(9), 3079–3090.

How to cite this article: Bischl, B., Binder, M., Lang, M., Pielok, T., Richter, J., Coors, S., Thomas, J., Ullmann, T., Becker, M., Boulesteix, A.-L., Deng, D., & Lindauer, M. (2023). Hyperparameter optimization: Foundations, algorithms, best practices, and open challenges. *WIREs Data Mining and Knowledge Discovery*, *13*(2), e1484. https://doi.org/10.1002/widm.1484