Efficient tuning of automated machine learning pipelines

Nguyen, D.A.

CHAPTER 3

An In-Depth Review of AutoML Optimization Approaches

This chapter introduces commonly used optimization approaches for AutoML optimization problems. We note that our discussion in this chapter will rely heavily on the problem definition and accompanying notations discussed in Chapter 1 (Section 1.1). These notations are crucial for the ongoing analysis, and we discussed them in detail in their original context in Chapter 1 to ensure a better understanding.

All of the approaches presented in this study follow the same principle: finding the best machine learning pipeline configuration $p \in \mathcal{M}$ to maximize a measurement performance[1] for a given machine learning problem with the k-fold cross-validation technique. They can be divided into two groups:

1. The performance of a particular configuration will be evaluated on all k-folds.
2. They intend to save computational cost by evaluating it on a subset of data, e.g., on fewer folds, to infer performance on the entire data. Hence, we shall use the term *function call* to indicate one-time access to a configuration on one fold.

The term *function evaluation* indicates the average performance over k folds, i.e., a *function call* is k times cheaper than a *function evaluation* in terms of evaluating data input. The difference between the term *function call* and *function evaluation* in this thesis is shown in Figure 2.5.

3.1 Black-box optimization approaches

In general, both hyperparameter and AutoML optimization problems are typically treated as black-box optimization problems for various reasons. For instance, we

[1] See Section 2.1.2.2.

cannot access a gradient of the objective function concerning the hyperparameters, or it is not possible to directly optimize the generalization performance as the training datasets are of limited size [8]. Generally, every black-box optimization approach can solve these problems. This section introduces three common optimization approaches, grid search (Section 3.1.1), random search (Section 3.1.2), and Bayesian optimization (Section 3.1.3). The working principles of these three approaches are shown in Figure 3.1. As shown, grid search sequentially evaluates points individually on a user-defined grid. On the other hand, a random search evaluates points at random, as the name implies. Bayesian optimization (BO) is a more complex technique based on advanced probabilistic models that makes it intelligent for automatically finding suitable configurations in the search space. In this example, we can see that BO can find more configurations with stronger results than other approaches. That is, more samples in the purple-filled contours.

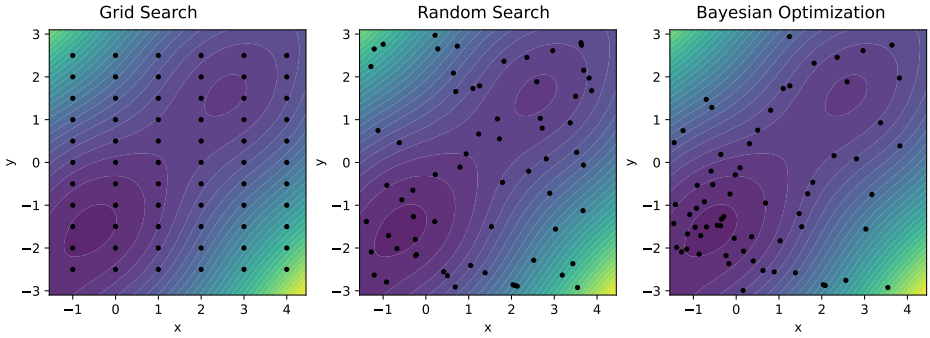

Figure 3.1: An illustration between grid search (left), random search (middle), and Bayesian optimization (right) for hyperparameter optimization on the McCormick's function $f(x, y) = -1.5x + 2.5y + (x - y)^2 + \sin(x + y) + 1$ with two continuous parameters: $x \in [-1.5, 4]$ and $y \in [-3, 3]$. All three approaches used a total of 66 functions evaluations. Purple-filled contours indicate regions with strong results, whereas yellow ones show regions with poor results.

3.1.1 Grid Search

Grid search is the most basic optimization algorithm. Given a set of hyperparameters, each of which has a (finite) set of values, for instance, continuous hyperparameter, e.g., $[0, 1] \in \mathbb{R}$, ordinal hyperparameter, e.g., $[1, 10] \in \mathbb{Z}$, boolean hyperparameter, e.g., [True, False], nominal hyperparameter, e.g.,[Linear, RBF, Poly, Sigmoid].

We enumerate all combinations of these sets and create a list of all candidates. Grid search evaluates each of these candidates and chooses the best configuration among them – the number of function evaluations is precisely the number of configurations. However, practitioners are usually restricted by a limited computational budget, i.e., the number of function evaluations, for hyperparameter optimization and AutoML optimization problems. Such a limited budget is typically much smaller than the number of possible evaluation configurations. Thus, a limited budget restricts the applicability of grid search.

3.1.2 Random Search

Unlike grid search, which assesses all configurations (for continuous hyperparameters based on a sufficiently coarse-grained discretization), random search [30] evaluates only a subset of available candidate configurations at random until the given budget runs out and returns the best of the sampled configurations. The random search for AutoML optimization is summarized in Algorithm 1, it consists of the following two steps:

- Generate a set of random configurations (line 3): here we adapted random sampling in unstructured HPO problem to AutoML optimization problem based on the search space (i.e., operators and hyperparameters space) and the number of needed configurations. The sampling algorithm is presented in Algorithm 2.
- Evaluating and selecting configuration: Each setting $p_i \in \{p_1, \ldots, p_B\}$ from the previous step will be evaluated on the objective function f (line 5). Next, the current best setting is updated (lines 6-9).

Lastly, when the optimization process is done, the best setting p^* is reported.

Random Sampling used in Algorithm 1 is presented in Algorithm 2:

- Random selection of a sequence of operators: All operators $\mathcal{O}_{1,\ldots,z} \in \mathbb{O}$ are randomly sampled to have a sequence of algorithms (line 5).
- Sampling hyperparameters: The corresponding hyperparameters are sampled randomly (lines 6-7), taking into consideration the selected algorithms in the previous step. The result is returned as a complete ML pipeline setting p, i.e., a sequence of ML algorithms and their hyperparameter settings.
- Lastly, the set of sampled configurations is returned.

Algorithm 1: Random Search

Input: $\mathbb{O}$: sequence of operators, Λ: hyperparameter spaces, f : objective function, B : number of iterations
Output: p^*: the best found configuration

$p^* \leftarrow \varnothing$
$\Delta^* \leftarrow 0$
$\Theta = \{p_1, \ldots, p_B\} \leftarrow$ RANDOM SAMPLING $(\mathbb{O}, \Lambda, B)$ `// see Algorithm 2`
foreach $p_i \in \{p_1, \ldots, p_B\}$ **do**
  $\Delta_i \leftarrow f(p_i)$ `// evaluate the configuration` p_i
  if $\Delta_i > \Delta^*$ **then**
    $p^* \leftarrow p_i$
    $\Delta^* \leftarrow \Delta_i$
  end
end
return p^*, Δ^* `// return the best found setting`

Recent studies [8], [13], [30], [47] have noted that random search can perform better than grid search, particularly when only a few hyperparameters impact the performance of the machine learning algorithm. Despite its simplicity, random search remains a crucial benchmark for evaluating the effectiveness of new optimization methods.

Algorithm 2: Random Sampling for AutoML optimization

Input: $\mathbb{O}$: sequence of operators, Λ: hyperparameter spaces, T: number of configuration
Output: $\Theta = \{p_1, \ldots, p_T\}$: set of T configurations

$t \leftarrow 1$
$\Theta = \varnothing$
while $t \leq T$ **do**
  $p \leftarrow \varnothing$
  foreach $\mathcal{O}_i \in \mathbb{O}$ **do**
    $\mathcal{A}_i^{n_i} \leftarrow \mathcal{U}(\mathcal{O}_i)$ `// randomly choose one algorithm for the` i^{th} `operator`
    $\lambda_i \leftarrow \mathcal{U}(\Lambda_i^{n_i})$ `// randomly select a hyperparameter setting for the selected algorithm` $\mathcal{A}_i^{n_i}$
    $p \leftarrow p \cup \{\mathcal{A}_i^{n_i}, \lambda_i\}$
  end
  $\Theta \leftarrow \Theta \cup p$ `// insert the new configuration` p `into` Θ
  $t \leftarrow t + 1$
end
return $\Theta = \{p_1, \ldots, p_T\}$ `// return a set of` T `configurations`

3.1.3 Bayesian Optimization

As the AutoML optimization task is typically time-consuming, it is preferable to devise/choose an optimizer that delivers a good ML pipeline setting with a relatively small computational budget. Building upon surrogate models and the expected improvement criterion, Bayesian Optimization (BO) [155] is designed for such a scenario. Generally, BO iteratively updates a surrogate model $\mathcal{P}(f|\mathcal{H})$which aims to learn the probability distribution of the response value conditioned on setting p, from the historical information, i.e., the so-far evaluated ML pipeline settings and the corresponding objective function $\mathcal{H} = \{(p_i, \Delta_i)_{i=1}^n\}$. The new candidate ML pipeline is chosen by optimizing the acquisition function [156], [157], which is defined over the surrogate model $\mathcal{P}$ and often balances the exploration and exploitation of the search. A detailed outline of the BO is presented in Algorithm 3 and Figure 3.2.

Many variants have been proposed for BO, including the Sequential Model-based Algorithm Configuration (SMAC) [25], Sequential Parameter Optimisation (SPO) [27], Mixed-Integer Parallel Efficient Global Optimization (MIPEGO) [38], and Tree-structured Parzen Estimator (Hyperopt) [24], [153], [158]. They differ mostly in the initial sampling method, the probabilistic model, and the acquisition function. Common choices for the probabilistic model are Random forests (RF) [72], Gaussian process regression (GP) [159], and TPE [24]. As for the acquisition function, the Expected Improvement (EI), the Probability of Improvement (PI) [157], and the Upper Confidence Bound (UCB) [160] are more frequently applied among many other alternatives.

3.1.3.1 Probabilistic Regression Models

The central idea of BO is to construct a surrogate model from the observed data points on real-valued objective function f. The surrogate model aims to predict the performance of untested ML pipeline configurations by modeling the relationship between the set of evaluated configurations Θ and their true response value Δ. In the following, we will briefly introduce three commonly used surrogate models: (1) Gaussian processes – the well-known traditional surrogate model, (2) Random Forest, and (3) Tree-structured Parzen Estimator – two popular surrogate models for AutoML optimization.

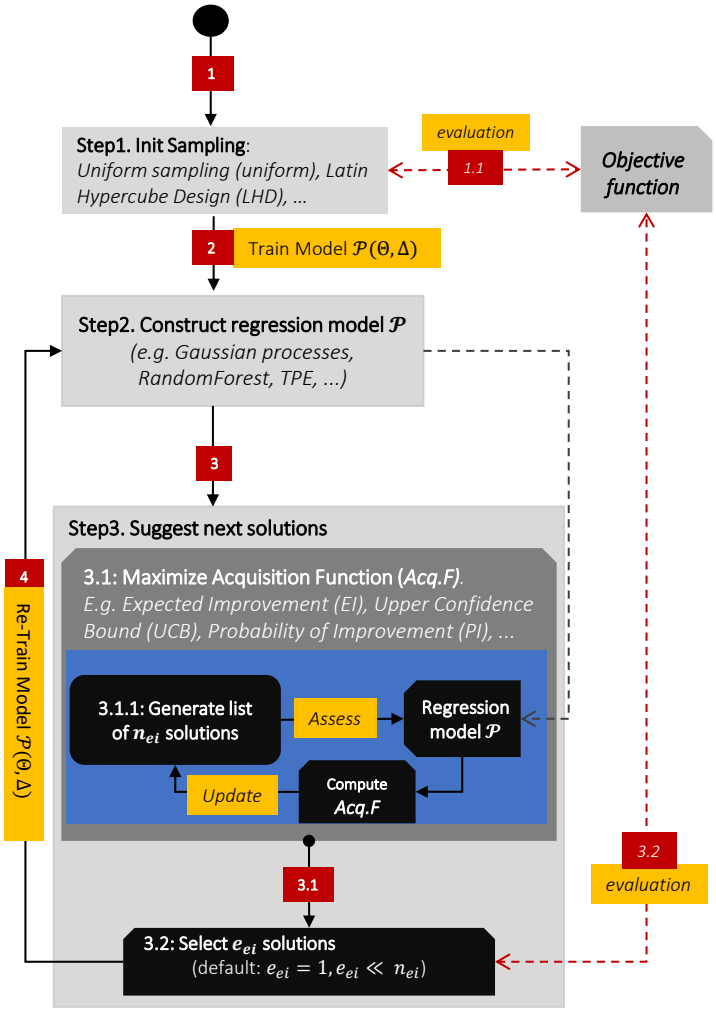

Figure 3.2: Bayesian Optimization

Gaussian processes

Gaussian processes (GP) [159] is a traditional surrogate model for Bayesian optimization. Generally speaking, GP is a generalization of Multivariate Gaussian distribution [161], where the mean vector $\boldsymbol{\mu}$ and covariance matrix $\boldsymbol{\Sigma}$ are redefined by a mean function $\mu(p)$ and a covariance kernel function $K(p, p')$ of any two observations[2]. The objective function f (expensive) is modeled as a GP, and can

[2]Note that a function is an infinite vector.

Algorithm 3: Bayesian Optimization for AutoML optimization

Input: $\mathbb{O}$: sequence of operators, Λ: hyperparameter spaces f : objective function, n_{init}: initial sample size,n_{ei}: number of samples to be sampled in the maximize expected improvement (EI) step, e_{ei}: number of samples to be suggested at each BO' iteration,

Output: p^*: the best ML pipeline configuration

$\Theta \leftarrow$ INIT SAMPLING$(\mathbb{O}, \Lambda, n_{\text{init}})$ `// typically done via Random sampling in Algorithm 2`

`// Evaluate all configurations on the given objective function` f`, their performances are saved to` Δ

$\mathcal{P} \leftarrow$ TRAIN $\mathcal{P}(\Theta, \Delta)$

while *not terminate* **do**

  $\Theta_{\text{new}} \leftarrow \mathcal{P}$.MAXIMIZE EI$(n_{\text{ei}}, e_{\text{ei}})$ `// EI is Expected improvement`

  $\Delta_{\text{new}} \leftarrow \varnothing$

  foreach $p_i \in \Theta_{new}$ **do**

    $\Delta_i \leftarrow f(p_i)$

    $\Delta_{\text{new}} \leftarrow \Delta_{\text{new}} \cup \Delta_i$

  end

  $\Theta \leftarrow \Theta_{\text{new}}$

  $\Delta \leftarrow \Delta_{\text{new}}$

  $\mathcal{P} \leftarrow$ TRAIN $\mathcal{P}(\Theta, \Delta)$ `// Re-train model`

end

return $p^* \in \Theta$ `// return the best found configuration`

be defined as $f(p) \sim \mathcal{GP}(\mu(p), K(p, p'))$. In other words, for each sample p, a Gaussian process defines a mean μ and standard deviation σ within a Gaussian distribution. The squared exponential function is a commonly used kernel function: $K(p, p') = \sigma_f^2 \exp[\frac{-(p-p')^2}{2l^2}]$ where l denotes the length scale[3] and σ_f^2 denotes the output variance. Subsequently, the predictive distribution for an unseen configuration $\mathcal{P}(f_{\text{new}}|\Theta, \Delta, p_{\text{new}})$ follows a Gaussian distribution. Hence, the mean and variance can be computed as follows:

$$\mu(p_{\text{new}}) = \mathbf{k}_* \mathbf{K}^{-1} \Delta; \sigma^2(p_{\text{new}}) = k_{**} \mathbf{k}_* \mathbf{K}^{-1} \mathbf{k}_*^{\text{T}} \tag{3.1}$$

where the covariance matrices are calculated as $\mathbf{k}_* = [k(p_1, p^*), \ldots, k(p_t, p^*)]$, $\mathbf{K} = [k(p_i, p_j)]_{\forall i,j \in \{1,\ldots,t\}}$ and $k_{**} = k(p_{\text{new}}, p_{\text{new}})$.

[3]The length scale establishes a point's *area of influence.* inside the parameter space, where the effect of an observation diminishes as one moves away from it.

Random forests

Random forests (RF) [72], is an algorithm used in Machine Learning for regression and classification. It is employed in SMAC [25] as the mere surrogate model[4]. Fundamentally, RF can be considered as a collection of regression (decision) trees. The historical data (Θ, Δ) were randomly divided into multi decision trees with few features. The trees in the forest individually score the unseen configuration candidate p_{new} ($p_{\text{new}} \in \mathcal{M}$), and the final result is based on the majority of the votes. A major limitation of RF is that it does not provide an estimate of the variance in its predictions. When adopting it in the BO scenario, SMAC [25] uses the empirical variance in the predictions of trees in the ensemble. Hence, mean μ and variance σ^2 for the new candidate p_{new} are computed as the empirical mean and variance of each tree r in the forest of B trees:

$$\mu(p_{\text{new}}) = \frac{1}{|B|} \sum_{r \in B} r(p_{\text{new}}); \sigma^2(p_{\text{new}}) = \frac{1}{|B|-1} \sum_{r \in B} \left(r(p_{\text{new}}) - \mu(p_{\text{new}})\right)^2 \quad (3.2)$$

where B denotes a set of trees, r denotes a tree in the forest, i.e., $r \in B$. $r(p_{\text{new}})$ is the predicted value of the new (unseen) configuration p_{new} by a tree r.

Tree-structured Parzen Estimator

Tree-structured Parzen Estimator (TPE) [24] is another alternative to a GP, which is a tree-based model by using the Parzen-window density estimators [162], [163]. Instead of modeling the distribution of the true objective function f, TPE models the likelihood $\mathcal{P}(\mathcal{H}|f)$ by using the parzen window kernel density estimator. In this setting, the evaluated configurations are split into two density distributions of a well $l(p)$ and a badly $g(p)$ performing set depending on whether its performance is below or above a predefined threshold[5] α. We note that $l(p)$ and $g(p)$ probability models are usually represented by Gaussian Mixture Models (GMMs) or Kernel Density Estimation (KDE) independently. Hence, we have two means, that is, $\mu_{l(p)}$ for the mean of $l(p)$ and $\mu_{g(p)}$ for the mean of $g(p)$, and two variances, that is, $\sigma^2_{l(p)}$ for the variance of $l(p)$ and $\sigma^2_{g(p)}$ for the variance of $g(p)$, the computation of these values depend on the models used. For the detailed discussion and relevant formulas, we refer the interested reader to [24] for further discussion on the Hyperopt framework, a well-known implementation of TPE, [164] and [165] for further discussion on KDE and GMMs.

[4]Note that the property of regression trees is supported conditional variables domains, while GP family currently do not.

[5]By default, $\alpha = 25\%$.

3.1.3.2 Acquisition Function

Bayesian optimization is designed to find a global optimum for an optimization problem that may have many local optima. However, as noted in the previous section, the surrogate model only approximates the true objective function, and its estimations may be imperfect. This leads to the following question: Should we exploit the most known search area or explore other areas that are less known? The so-called Acquisition Function (AF, *or infill-criterion*) [155] is designed to answer this question, aiming to achieve a trade-off between exploration and exploitation. Generally, AF computes the expected improvement value from the mean and covariance (uncertainty) estimated by a regression model. Therefore, we can choose a new configuration candidate for evaluation by maximizing the expected improvement values.

Although many acquisition functions have been proposed [38], [157], [160], [166]–[172], the *Expected improvement* (EI) [156] is the most popular acquisition function for BO and remains the default AF in BO packages, such as SMAC [25], Spearmint [28], SPO [27] and TPE [24], [158]. EI balances the trade-off between exploration and exploitation via the expectation of the improvement function over the best-found value $\Delta^*_{(t)}$ at time step t as $I_t(p) = \max\{0, \hat{f}(p) - \Delta^*_{(t-1)}\}$, where $\Delta^*_{(t-1)} = \max(\Delta^0, \ldots, \Delta^{t-1})$ and $\hat{f}(p)$ denotes the predicted performance of the setting p via surrogate model $\mathcal{P}$. The EI is thus defined as:

$$\mathbb{E}[I_t(p)] = \int_0^\infty I_t(p)\, d\mathcal{P} \tag{3.3}$$

Let us denote by $z = z_{(t-1)}(p) = \frac{\mu_{(t-1)}(p) - \Delta^*_{(t-1)}}{\sigma_{(t-1)}(p)}$, we obtain the closed-form AF by taking the expectation via the improvement function $I_t(p)$ as:

$$\mathbb{E}[I_t(p)] = \sigma_{(t-1)}(p)\phi(z) + [\mu_{(t-1)}(p) - \Delta^*_{(t-1)}]\Phi(z) \tag{3.4}$$

where $\mu(p)$ and $\sigma(p)$ denoted the mean and standard deviations; $\phi(.)$ and $\Phi(.)$ are the probability density function (p.d.f) and the cumulative distribution function (c.d.f) of the standard normal distribution. Hence, the next setting is selected by maximizing the EI:

$$p_{\text{new}} = \underset{p \in \mathcal{M}}{\operatorname{argmax}}\, \mathbb{E}[I_t(p)] \tag{3.5}$$

where $\mathcal{M}$ denotes the AutoML search space (see Section 1.1).

3.2 Multi-fidelity approaches

As mentioned in Section 1.1, the k-fold cross-validation is typically used when evaluating a pipeline configuration to avoid the over-fitting problem (see Section 2.1.3). However, if the performance of a particular configuration is poor when evaluated on the first folds, it is likely to not perform well on the rest of the cross-validation fold [173]. Hence, we should not invest further computational resources in this configuration or redistribute the saved resources to the most promising configurations. A class of optimization methods named *multi-fidelity approaches* intends to save computational resources and speed up the optimizing process by evaluating configurations on a subset of the input data [174], [175], limiting iterations [13], or using a subset of features [8]. In this study, we use the term k-fold cross-validation (see Figure 2.5 for reference), then the multifidelity limits the use of a few folds of the cross-validation folds, that is, using i folds ($i \leq k$). The configuration candidates in this class of methods tend to be evaluated faster on fewer folds than the approaches in Section 3.1. Hence, we use the term *function call* to indicate a one-time access to a configuration on one fold.

This thesis reviews two commonly used classes of methods aiming at reducing computational effort, namely: (1) Racing procedure approaches and (2) Bandit-based approaches.

3.2.1 Racing procedure

Hoeffding Races [32], [173] were the first version of the racing procedure. It was initially designed to find the best machine learning model for a set of problem instances (here, we use the term k-folds instead) in the supervised machine learning domain. To reduce the computational cost of poor configurations, a (pairwise) statistical test (e.g., t-test, Friedman-test [176]) is used to determine poor configurations to be terminated as soon as enough statistical evidence arises against them, that is, the ones that are significantly worse than the best.

Although a number of racing procedure variants have been developed, such as F-Race [177], [178], Sampling F-Race [179], and Iterated racing (irace) [33], [180], [181] ; irace is the most the latest of this class. It is particularly well-suited for Hyperparameter Optimization [13] and AutoML optimization [182]. Hence, we present the irace algorithm in greater detail in the following section. For details on other methods, we refer the reader to Birattari (2009) and the book chapter by Hoos (2012).

3.2.1.1 Iterated racing (irace)

In contrast to Hoeffding Races, the later variants of the racing procedure add a rank-based Friedman test (i.e., Friedman two-way analysis of variance by ranks) to determine if there is any significant difference between configurations. If any differences were found, pairwise comparisons were performed with the best candidate. Irace also followed this procedure. The detailed outline of irace is shown in Algorithm 4. Irace first initializes parameters (lines 1-2). The first round uses the random sampling method in Algorithm 2, generates a set of $N_j (N_j = \lfloor \frac{B_j}{T^{\text{first}} + T^{\text{each}}} \rfloor)$ configurations (line 5). A racing procedure (line 6) is used to discard poorly performing configurations, based on their evaluated performance on T^{first} folds. This race relies on the Friedman test [176] and Conover post-hoc test [183] with a significance level α[6].

After the first race, a new race is initialized by taking the remaining budget $(B - B^{\text{used}})$ and the number of remaining races $(N^{\text{iter}} - j + 1)$ (lines 11-12). Next, EliteBasedSampling generates a set of N_j configurations by sampling the set of surviving configurations Θ^*, from the previous race. For every new sample, the sampling procedure repeats as follows:

1. One sequence of operators will be chosen as the parent sequence $(\mathcal{A}_1, \ldots, \mathcal{A}_z)^{\text{parent}}$ (see Figure 1.2 for the used notation) for this new race, with a probability ρ^{parent} that is based on its configuration p^{parent}, $(p^{\text{parent}} = (\mathcal{A}_{1,\lambda}, \ldots, \mathcal{A}_{z,\lambda})^{\text{parent}} \in \Theta^*)$ and its rank r^{parent} over the surviving set Θ^*. The probability ρ^{parent} is computed as follows:

$$\rho^{\text{parent}} = \frac{2 \cdot |\Theta^*| - r^{\text{parent}} + 1}{|\Theta^*| \cdot (|\Theta^*| + 1)} \tag{3.6}$$

2. The corresponding hyperparameters λ to $(\mathcal{A}_1, \ldots, \mathcal{A}_z)^{\text{parent}}$ are sampled by either a truncated normal distribution for numerical hyperparameters, or a discrete distribution for categorical hyperparameters[7].

3.2.2 Bandit-based approaches

The class of bandit-based approaches is similar to the racing procedure in that they terminate the worst pipeline configurations early. However, compared with the racing procedure, they differ in two ways:

[6] By default $\alpha = 0.05$.

[7] Ordinal hyperparameters are considered as numerical.

Algorithm 4: Iterated racing algorithm

Input: $\mathbb{O}$: sequence of operators, Λ: hyperparameter spaces, f : objective function, $\mathbf{I}$: set of problem instances (or *set of k folds*), T^{first}: the number of instances (*folds*) needed to do the first test, T^{each}: the number of instances (*folds*) to test on the later round (by default $T^{\text{each}} = 1$), B: total budget (*maximum number of function calls*)

Output: Θ^*: set of best configurations

$N^{\text{param}} \leftarrow |\mathbb{O}|+|\Lambda|$ // number of parameter spaces equal to the total number of operators and the total number of algorithms' hyperparameters

$N^{\text{iter}} \leftarrow \lfloor 2 + log_n N^{\text{param}} \rfloor$ // number of races to be executed

// THE FIRST RACE

$j \leftarrow 1$ // $j = 1, \ldots, N^{\text{iter}}$

$B_j \leftarrow \frac{B}{N^{\text{iter}}}$ // compute budget for the first round

$N_j \leftarrow \lfloor \frac{B_j}{T^{\text{first}}+T^{\text{each}}} \rfloor$ // number of configuration to be sampled at the first race

$\Theta^j \leftarrow$ RANDOM SAMPLING$(\mathbb{O}, \Lambda, N_j)$ // see Algorithm 2

$\Theta^* \leftarrow$ RACE$(\Theta^j, B_j, T^{\text{first}}, \mathbf{I})$ // determine the set of good configurations

$B^{\text{used}} \leftarrow B_j$ // used budgets

// LATER RACES

while *not terminate* **do**

  $j \leftarrow j + 1$

  $B_j \leftarrow \frac{B-B^{\text{used}}}{N^{\text{iter}}-j+1}$ // compute budgets for the current race

  $N_j \leftarrow \lfloor \frac{B_j}{T^{\text{first}}+T^{\text{each}} \times \min\{5,j\}} \rfloor - |\Theta^*|$ // number of configurations to be sampled for the current race

  $\Theta^j \leftarrow$ ELITEBASEDSAMPLING$(\mathbb{O}, \Lambda, N_j, \Theta^*)$

  $\Theta^j \leftarrow \Theta^j \cup \Theta^*$

  $\Theta^* \leftarrow$ RACE$(\Theta^j, B_j, T^{\text{each}}, \mathbf{I})$ // determine the set of good configurations

  $B^{\text{used}} \leftarrow B^{\text{used}} + B_j$ // update used budgets

end

return Θ^*

1. All configurations were compared directly based on their evaluated performance instead of using a statistical procedure.

2. The number of rounds and budget can be estimated based on the input budgets, that is, the budgets for each round are equally assigned, but later rounds have fewer candidates than the previous rounds.

In the following, we present *Successive Halving* [34] in more detail and outline its limitations. Next, we discuss two variants of Successive Halving to overcome these limitations [35], [36].

3.2.2.1 Successive Halving

Jamieson and Talwalkar (2016) introduced *Successive Halving* as a simple yet efficient algorithm for multi-fidelity optimization. The outline of *Successive Halving* is summarized in Algorithm 5. Here, we slightly adapt the algorithm for AutoML optimization. That is, we use the term AutoML search space instead of hyperparameter space, and use it for the k-fold cross-validation scenario, that is, at least one candidate will be assigned a sufficient budget to evaluate all k folds, and no configuration can have more than that budget. It requires a budget (finite value) B, i.e., the maximum number of function calls, the number of configurations n, the maximum number of folds that can be used for a single configuration R. The procedure pre-computes the number of rounds t to be executed (line 1). The value of t is then recomputed by using line $2-7$ to find an appropriate value of t for the provided budget B and n when using the discard ratio η. Next, a set of n configurations is generated randomly and saved to Θ_r (line 3). For each round, the budget for a configuration is computed in line 13, i.e., either a subset of data or $B_r \in \{1, \ldots, k\}$ folds ($B_r << k$). Herein, we slightly modify to adapt to the above mentioned scenario to ensure no configuration has more than R folds and less than 1 folds. Next, all configurations $p \in \Theta_r$ are assessed on B_r folds. At the end of the round, we only keep the top $\frac{1}{\eta}$ configurations[8] based on their performances to go to the next round (line 16). The successive procedure is then repeated until the last round is done. Lastly, the best-found configuration is returned (line 19).

3.2.2.2 Hyperband

Successive Halving requires the number of configurations n and budgets B, e.g., the number of function calls, as input parameters. Assume that we have a fixed budget B, e.g., total number of function calls, the proportion of $\frac{B}{n}$ leads to a consideration of whether we should consider (1) more configurations (large n) in the race with small average folds or (2) a small number of configurations (small n) with higher average folds. [35] pointed out that in practice, the problem itself might have some noise, i.e., the accuracy rate on folds might be significantly different. If the noise is

[8]We note that the default proportion discard of *half* was changed to *one third* with the recent studies [35]–[37].

Algorithm 5: Successive Halving algorithm

Input: $\mathbb{O}$: sequence of operators, Λ: hyperparameter spaces, f : objective function, $\mathbf{I}$: set of problem instances (or set of k folds), B: total budget (*maximum number of function calls*), n: number of configurations, η: Proportion discard ratio ($\eta = 2$ by default), R: maximum number of instances (folds) that can be allocated to a configuration ($R = |\mathbf{I}|$, by default).

Output: Θ_t

$t \leftarrow \log_\eta(\min\{R, n\})$ `// pre-compute the number of rounds based on` R, n `and` η
foreach $t \leq log_\eta(min\{R, n\})$ **do**
  if $nR(t+1)\eta^{-t} \leq B$ **then**
    Evaluate whether the given budget B are sufficient to accommodate the number of rounds t.
    return t `// number of rounds to be executed`
  else
    $t \leftarrow t - 1$
  end
end
$\Theta_r \leftarrow$ RANDOM SAMPLING$(\mathbb{O}, \Lambda, n)$ `// randomly create` n `configurations using Algorithm 2`
$R_{\text{remain}} \leftarrow R$ `// number of instances/folds remain unevaluated.`
$r \leftarrow 0$
while $r \leq t$ **do**
  $B_r \leftarrow \min(\max(\lfloor (R\eta^{r-t} \rfloor, 1), R_{\text{remain}})$ `// Budget for a surviving configuration in the current round.`
  foreach $p \in \Theta_r$ **do**
    Assess the configuration p on B_r folds, which have not been evaluated so far.
  end
  $\Theta_{r+1} \leftarrow$ SELECT TOP $\lfloor \frac{|\Theta_r|}{\eta} \rfloor$ in Θ_r `// keep` $1/\eta$ `good configurations in terms of their corresponding observed performances`
  $R_{\text{remain}}, r \leftarrow R_{\text{remain}} - B_r, r + 1$
end
return $p \in \Theta_r$ `// return the best found configuration`

low, we can quickly determine the quality of the configurations on fewer folds. We can select a large number of configurations to maximize the possibility of finding the optimal solution. Otherwise, we should consider fewer configurations, but we will evaluate them in more detail.

Exploiting this finding, [35] proposed to have an outer loop of *Successive Halving*,

which will consider a different proportion of $\frac{\text{total budgets}}{\text{number of configurations}}$, where the number of configurations on each outer loop is reduced. The complete algorithm is outlined in Algorithm 6. The idea behind Hyperband is that it divides resources into brackets, e.g., $\{N^{\text{iter}}, N^{\text{iter}}-1, \ldots, 0\}$, with different configurations and executes *Successive Halving* as a sub-program, that is, the first loop executes *Successive Halving* with many configurations, but most of them will validate fewer folds. In contrast, the last loop handles fewer configurations but will be validated on most folds. This outer loop is in lines $4-13$.

Algorithm 6: Hyperband algorithm

Input: $\mathbb{O}$: sequence of operators, Λ: hyperparameter spaces, f : objective function, $\mathbf{I}$: set of problem instances (or set of k folds), B: total budget (*maximum number of function calls*), η: Proportion discard ratio ($\eta = 3$ by default), R: maximum number of instances that can be allocated to a configuration ($R = |\mathbf{I}|$, by default).

1 $N^{\text{iter}} \leftarrow \lfloor log_{\eta}(R) \rfloor$
2 $\Theta \leftarrow \varnothing$ `// set of configurations`
3 $B_{\text{remain}} \leftarrow B$
4 **foreach** $r \in \{N^{iter}, N^{iter}-1, \ldots, 0\}$ **do**
5 $\quad n_r \leftarrow \lceil \frac{B}{R} \times \frac{\eta^r}{r+1} \rceil$ `// number configurations to be sampled`
6 $\quad$ **if** $r > 0$ **then**
7 $\quad\quad B_r \leftarrow \frac{B}{N^{\text{iter}}}$ `// total budget for the current round`
8 $\quad$ **else**
9 $\quad\quad B_r \leftarrow B_{\text{remain}}$ `// total budget for the last round`
10 $\quad$ **end**
11 $\quad \Theta \leftarrow \Theta \cup \text{SuccessiveHalving}(\mathbb{O}, \Lambda, f, \mathbf{I}, B_r, n_r, \eta, R)$
$\quad$ `// SuccessiveHalving (Algorithm 5) is used as a subroutine.`
12 $\quad B_{\text{remain}} \leftarrow B_{\text{remain}} - B_r$ `// remaining budget`
13 **end**
14 **return** $p^* \in \Theta$ `// return the best found configuration`

Finally, both *Hyperband* and *Successive Halving* are considered fast random search methods owing to the use of random sampling for generating configurations. Therefore, they also inherited the major limitation of random search for proposing new configurations but were not improved to take information accumulated over the search history into account, such as Evolutionary Strategies [108], Bayesian Optimization [159], [184], which can propose configurations based on the assessed points so far. Instead of randomly proposing new configurations, BOHB [36] and DACOpt [37] proposed to use combine a Bandit approach and Bayesian

Optimization to maximize expected improvement. BOHB replaces the random sampling step in Successive Halving by TPE at the step after the first round. In contrast, DACOpt uses Successive Halving as an outer loop that can suggest good search areas for Bayesian optimization.