FLO: Fast and Lightweight Hyperparameter Optimization for AutoML

Chi Wang* & Qingyun Wu†

November 13, 2019

Abstract

Integrating ML models in software is of growing interest. Building accurate models requires right choice of hyperparameters for training procedures (learners), when the training dataset is given. AutoML tools provide APIs to automate the choice, which usually involve many trials of different hyperparameters for a given training dataset. Since training and evaluation of complex models can be time and resource consuming, existing AutoML solutions require long time or large resource to produce accurate models for large scale training data. That prevents AutoML to be embedded in a software which needs to repeatedly tune hyperparameters and produce models to be consumed by other components, such as large-scale data systems. We present a fast and lightweight hyperparameter optimization method FLO and use it to build an efficient AutoML solution. Our method optimizes for minimal evaluation cost instead of number of iterations to find accurate models. Our main idea is to leverage a holistic consideration of the relations among model complexity, evaluation cost and accuracy. FLO has a strong anytime performance and significantly outperforms Bayesian Optimization and random search for hyperparameter tuning on a large open source AutoML Benchmark. Our AutoML solution also outperforms top-ranked AutoML libraries in a majority of the tasks on this benchmark.

1 Introduction

The software industry has grown an increasing interest of integrating data-driven decision making components fueled by machine learning techniques. A classification or regression model, for example, makes predictions by learning from labeled data. Ideally, a software should be able to automatically produce such models given labeled data, and let other components consume their predictions. There are many great options of generic supervised learners, such as random forest, gradient boosted trees and neural networks, available from existing libraries such as scikit-learn [PVG+11] and XGBoost [CG16]. Each learner has a set of hyperparameters to tune which have a large impact to the model accuracy after training data are given. Hyperparameter optimization is a core component for building Automated Machine Learning (AutoML) software.

Since 2013, a number of open source AutoML libraries have been developed, such as Auto-WEKA [THHLB13], auto-sklearn [FKE+15], TPOT [OUA+16] and H2O AutoML [H2O]. Major cloud service providers like Amazon, Microsoft and Google also offer services of similar kind. Most of them are used by data scientists outside the software consuming the produced models. One limiting factor for software integration is that existing AutoML solutions require long time or large resource to produce accurate models for large scale training datasets, despite promising results published based on small datasets. For example, using a recent large-scale AutoML benchmark [GLT+19], the state-of-the-art solutions underperform a tuned random forest baseline on 40-50% tasks given one CPU hour. Time and resource efficiency constrains their integration into latency-sensitive and resource-hungry software, which is the case for most commercial systems that process large scale data.

The major source of inefficiency stems from the challenge of hyperparameter optimization. Most AutoML solutions are based on random search [BB12], Bayesian optimization [BCdF10], or

*Microsoft Research, Redmond, WA; email: wang.chi@microsoft.com
†University of Virginia, Charlottesville, VA; email: qw2ky@virginia.edu; work done in Microsoft Research

genetic programming [OBUM16], which are all generally more efficient than grid search [PVG+11]. The dominating search strategy is regarded as Bayesian optimization, a 40-year old technique for blackbox function optimization [O'H78]. It has been applied to AutoML in a series of academic work since a decade ago [BCdF10, BBBK11, HHLB11, SLA12], and embraced by many libraries. The main idea is to use Bayesian models to guide the random research. The model uses observations of the accuracy of random hyperparameter choices to predict the accuracy corresponding to each choice. As more observations are collected, the predicted accuracy is weighed more in the randomized search, and the search is hopefully oriented towards more promising hyperparameters. The drawback is that the model requires a large number of random observations in the hyperparameter space to play a significant (not necessarily positive) role. When the dataset is large and evaluation (e.g., cross validation or holdout) is expensive, the search efficiency is close to uniformly random search under constrained time and resource. Some effort has been made to address this drawback by introducing non-uniform meta-learned prior based on a large number of experiments in history using many different datasets, learners and hyperparameters. However, this requires heavy weight preparation using even more resources, including huge computational burden of running and saving millions of experiments, and acquisition of datasets with sufficient coverage to train a meta model, which is not always feasible or accessible.

We present FLO, a fast and lightweight hyperparameter optimization method for AutoML. Given a labeled dataset and an accuracy metric, it continuously searches for hyperparameter choices and produces models optimizing the metric. It outputs more and more accurate models as the search time goes by. The goal is to use minimal amount of time and computational resource to find a model with desired accuracy. It can be used to tune the hyperparameters of a specific learner, or used as a building block for a more complex AutoML system. As a proof of the former, we enriched the AutoML Benchmark [GLT+19] with the a regression benchmark [OLCO+17], and evaluate FLO's performance in tuning XGBoost. FLO outperforms random search, Bayesian optimization, as well as existing AutoML libraries in most of the datasets. As a proof of the latter, we extend FLO to be a fully-functional AutoML library, *FLAML*, which demonstrates superior performance over existing AutoML libraries on the majority of the datasets in the enriched AutoML Benchmark.

Our main idea is to leverage a unique structure in the search space of hyperparameters for machine learners: There are a subset of hyperparameters related to model complexity which have significant blackbox impact to model accuracy, and significant whitebox impact to evaluation cost. For example, in gradient boosted trees, the number of trees and the depth per tree are such hyperparameters. More complex data distributions generally require more and deeper trees. Using wrong complexity hyperparameters can lead to overfitting or underfitting of the data, both leading to low accuracy on test data. But the exact best choice is unknown and varies with the specific dataset. On the other hand, the training and testing cost is linear to both hyperparameters. So evaluating hyperparameters corresponding to low complexity is cheaper than those corresponding to high complexity. Intuitively, we should prioritize the former when the search time is limited, in case small models suffice. Ignoring this priority will make the searcher prone to trying unnecessarily large models, which cost long time to evaluate. But we should not overprioritize it in case large models are required, to avoid spending too much time in trying small models. It is important to strike a right balance as the true best complexity is unknown a priori.

While the complexity-related model behavior is well known in the ML community [HTF01], perhaps surprisingly, it has not been exploited by existing AutoML solutions. One hypothetical reason is that most solutions are designed and evaluated using small scale datasets and the required model complexity does not have a large variation. Also, it is difficult for Bayesian optimization to effectively leverage this insight because it does not expose explicit control over the search order. This is in contrast to gradient-based optimization, which directs the search based on derivatives of the function to optimize. Unfortunately, hyperparameter optimization is a blackbox optimization problem for which the gradient of the objective function is not available to the optimizer. Therefore, we design a new *zeroth-order* optimization method. As opposed to Bayesian optimization, our method gradually traverses the search space and constrains the exploration of new hyperparameter choices to be within a distance from the previous choice. As opposed to gradient-based optimization method, our method only needs access to the value of the objective, which is the accuracy metric evaluated from each hyperparameter choice. Our method takes into consideration the evaluation cost difference between simple and complex models. Intuitively speaking, it starts from smallest models and then quickly moves to better models. To further improve latency for large datasets,

we inject a progressive data sampling strategy during the search. It leverages our insight about model complexity in a reverse direction: Smaller models require smaller sample size to evaluate than larger models. Therefore, we can also start from small sample size and use larger sample size when we search for larger models. To decide when to increase sample size, we invent a notion of *expected time for improvement* (abrv. ETI) based on the search trajectory. Using ETI, FLO adaptively changes sample size in runtime without hardcoded scheduling.

This work makes the following contributions.

- We propose a novel zeroth-order optimization method FLO to leverage the unique structure of search space for ML hyperparameters. It is the first method to successfully leverage the correlation between hyperparameter and evaluation cost to optimize search efficiency. We tailor our design to performing well without knowing the true best model complexity required for a dataset. In the mean time, our method is model-free, and has almost no computational overhead beyond the evaluations of each hyperparameter choice.
- We extend the proposed hyperparameter tuning method to build a AutoML solution FLAML. Our extension remains fast and lightweight. To support multiple learners, we add a learner selection component to choose one learner to perform one step of hyperparameter search at each iteration. FLAML adaptively switches among learners based on ETI, which considers the speed of improvement of each learner.
- We perform extensive evaluation using the latest AutoML benchmark which contains large-scale binary and multi-classification tasks. We also enrich it with datasets from a regression benchmark to test regression tasks. With varying time budget from one minute to one hour, FLO outperforms random search, six variations of Bayesian optimization, as well as three existing AutoML libraries in most of the tasks given the same budget, with significant margin. Moreover, even with one *minute*, FLO already found models with better or equal performance than the other methods in one *hour*, for 40-92% tasks. Likewise, FLAML wins over the three AutoML libraries in the majority of tasks with significant margin. FLAML's performance in one *minute* is better than or equal to auto-sklearn, H2O AutoML and TPOT's performance in one *hour* on over 62%, 58% and 75% datasets, respectively.

2 Related work

A overview of a selection of AutoML tools can be found in Table 1. It lists the four top-performing open-source AutoML libraries according to the AutoML Benchmark [GLT+19], two proprietary tools with publications recently, and our AutoML solution FLAML based on our optimization method FLO. Note that the scope of our study is limited to the tabular data. So we do not include the large literature about automated neural architectural search for highly specialized domain like images, which is branded as AutoML sometimes.

Table 1: A comparison of a selection of AutoML tools

Tool	Optimization	Learner Selection	Meta-learning	Ensemble
Auto-WEKA	BO (SMAC)	As Hyperparameter	-	-
Auto-sklearn	BO (SMAC)	As Hyperparameter	Warm-start	✓
TPOT	Genetic Programming	As Hyperparameter	-	✓
H2O AutoML	Random Search	Fixed Order Enum	-	✓
Azure AutoML	BO (GP)	As Hyperparameter	Collaborative Filtering	✓
Alpine Meadow	BO (SMAC)	Weighted Sampling	Random Forest	-
FLAML	FLO	Adaptive Order	-	-

Auto-WEKA [THHLB13] is built on the Java machine learning package WEKA. It is one of the first systems to consider joint algorithm selection and hyperparameter optimization in addition to preprocessing steps. Auto-sklearn [FKE+15] is declared the overall winner of the ChaLearn AutoML Challenge 1 in 2015-2016 and 2 in 2017-2018. It provides a scikit-learn-like interface in Python. It features automatic ensemble construction after Bayesian optimization. Meta-learning is used to warm-start the search procedure with a few pipelines based on the similarity between the new dataset with historical datasets. The similarity is measured using meta features of each dataset

such as the number of features and instances. TPOT [OUA+16] constructs machine learning pipelines of arbitrary length using scikit-learn learners and XGBoost. In its search, preprocessing and stacking are both considered. While technically pipelines can be of any length, TPOT performs multi-objective optimization: It aims to keep the number of components in the pipeline small while optimizing the main metric. H2O AutoML [H2O] is a Java-based open-source library. It performs random search for each learner in the H2O machine learning package, in addition to XGBoost. The learners are ordered manually, starting from XGBoost. After searching hyperparameters for all the learners, it uses stacking to ensemble the models. Azure AutoML [FSE18] uses collaborative filtering to improve Bayesian optimization. It collects hundreds of datasets and more than 40,000 ML pipelines, and run each pipeline on each dataset. That results in millions of experiments. The results are stored in a huge matrix and used to recommend pipelines to run for a new dataset. The recommended pipeline must be one of the seen 40K ML pipelines. So it must run a large number of pipelines for preparation, and once a new pipeline is added, it must be run on a large number of datasets. Due to the huge preparation cost, only small datasets are used for meta-learning, which makes generalizability to large datasets a question. Alpine Meadow [SZB+19] also focuses on small datasets as it features interactivity. It heavily relies on parallelism as well as meta-learning to achieve good performance. The hundreds of datasets used for meta-learning are proprietary from DARPA. FLAML does not use meta-learning or ensemble, though they can be added in future work.

From the overview, we can see that automated hyperparameter tuning and learner selection are the core of AutoML systems. The majority of libraries employ Bayesian optimization for hyperparameter tuning, with TPOT and H2O AutoML as two exceptions (using genetic programming and random search respectively). The main improvement over the years is in the use of meta learning, ensemble and more computational resources, while the fundamental algorithms behind these libraries are established before 2013. In the following, we briefly review those algorithms.

Prior to these AutoML tools, grid search was the most popular tuning strategy. It discretizes the search space of the concerned hyperparameters and tries all the values in the grid. It can naturally take advantage of parallelism. However, the cost of grid search increases exponentially with hyperparameter dimensions. A simple yet surprisingly effective alternative is to use random combinations of hyperparameter values, especially when the objective function has a low effective dimensionality, as shown in [BB12]. Bayesian optimization (BO) [BCdF10] fits a probabilistic model to capture the relationship between hyperparameter settings and their measured performance, uses this probabilistic model to make decisions about where in the space to next evaluate the function, while integrating out uncertainty. It trades off exploration of new parts of the space vs. exploitation in known good regions, evaluates that hyperparameter setting, updates the model with the result, and iterates. There are two major choices that must be made when performing Bayesian optimization. First, one must select a prior over functions that will express assumptions about the function being optimized. For example, Snoek et al. [SLA12] use Gaussian process (GP), Bergstra et al. [BBBK11] uses the tree Parzen estimator (TPE), and Hutter et al. [HHLB11] uses random forest (SMAC). Second, one must choose an acquisition function, which is used to construct a utility function from the model posterior, allowing us to determine the next point to evaluate. Two common acquisition functions are the *expected improvement* over the currently best observed value, and *upper confidence bound*. They both take the uncertainty of probabilistic model estimation into consideration, and automatically trade off between exploration and exploitation as a result of the Bayesian decision theoretic treatment.

We should note that BO is a generic approach for blackbox function optimization, and has shown superiority over random search in problems beyond AutoML [GSM+17]. However, the strong performance of H2O AutoML is an indicator that the complexity of using BO is not fully justified in the context of AutoML. Random search remains a simple yet strong baseline for AutoML, for several potential reasons. First, BO methods usually depends on the proper choice of prior and acquisition function, which may greatly affect the performance of BO methods and some choices of them can make BO methods computationally expensive. Second, BO essentially tries to optimize the number of function evaluations, which does not necessarily leads to minimized evaluation time. There exists some heuristic technique to take evaluation cost into consideration through *expected improvement per second* [SLA12]. But this technique does not solve the problem fundamentally and can be computationally very inefficient. The mismatch between the number of function evaluations and the total cost in the context of AutoML is a common issue for all the other studies of hyperparameter optimization methods (such as the ones listed in [KGG+18]).

That motivates our innovation in the search strategy.

3 FLO

This section describes our proposed hyperparameter optimization method. We formulate the problem in Section 3.1, and then present the main idea of our solution in Section 3.2. The algorithm is detailed in Section 3.3.

3.1 Hyperparameter Optimization for A Single Learner

This section focuses on the setting of a fixed learner L with d hyperparameters to tune. Hyperparameter tuning for L can be considered as an optimization problem where we want to minimize a loss function $f : \mathcal{S} \to \mathcal{R}$, in which $\mathcal{S} = [l_1, h_1] \times \cdots \times [l_d, h_d]$ is the valid configuration space in $\mathcal{R}^{+d}$. Here we assume all the hyperparameters are positive numbers, and l_i, h_i are the lower and upper bound for each hyperparameter. With suitable transformation, that assumption does not lose generality for numeric hyperparameters. We discuss categorical hyperparameters in the next section.

In the AutoML context, $f(\mathbf{c})$ is a given error metric of a model when trained using the hyperparameter $\mathbf{c}$. The metric is evaluated either using cross-validation or held-out validation data. The cost of evaluating $f(\mathbf{c})$ is thus the cost of cross-validation or training and testing using the hold-out method. We denote the cost as $g(\mathbf{c})$. The ideal goal is to minimize the total cost before finding a model with best accuracy. Since the evaluation cost is the dominating cost of the entire search procedure, we can optimize our search sequence $\{\mathbf{c}_i\}$ with the following goal:

$$\begin{aligned} &\min_{i^*, \{\mathbf{c}_i\}_{i=1}^{i^*}} \sum_{1 \le i \le i^*} g(\mathbf{c}_i) \\ &s.t. f(\mathbf{c}_{i^*}) = \min_{\mathbf{c} \in \mathcal{S}} f(\mathbf{c}) \end{aligned} \tag{1}$$

i^* corresponds to the number of iterations it takes to find the minimal loss. Let $\mathbf{c}^*$ be the configuration with the lowest evaluation cost among all configurations with best accuracy, i.e., $\mathbf{c}^* = \arg\min_{f(\mathbf{c}) = \min_{\mathbf{c}' \in \mathcal{S}} f(\mathbf{c}')} g(\mathbf{c})$. The optimal search sequence is then $g(\mathbf{c}^*)$ with $i^* = 1, \mathbf{c}_1 = \mathbf{c}^*$. That is, the first evaluated configuration in the search sequence happens to be the optimal configuration $\mathbf{c}^*$. This is almost impossible. So the above optimization problem is extremely difficult and no previous work has formulated it this way. However, we would still like to present this formulation to emphasize that we aim to minimize the total cost. In contrast, previous hyperparameter optimization method inherently tries to minimize the number of iterations i^* to find the minimal loss. It is reasonable when the evaluation cost does not vary much for different configurations, however it requires the search space to be constrained in a narrow space. For AutoML, It is not uncommon for the evaluation cost to range from subseconds to minutes of CPU time. As a general purpose tuning software that can scale to different problem size, this limitation needs to be addressed. With that said, optimizing for evaluation cost makes the blackbox optimization problem even more difficult, which explains the reason of no previous attempt.

In practice, since the ground truth of the best accuracy is never known, the effectiveness of a search sequence is evaluated in a more pragmatic way. We either measure the error within a given time, or measure the time taken to reach a given error threshold. The former is more common because the reachable error thresholds are often unknown as well. And the time budget is usually varied to obtain a comprehensive assessment of the search performance.

3.2 Main Idea

Many machine learners have a small subset of hyperparameters with known impact to model complexity (abrv. MCPs). These hyperparameters are critical to tune because different datasets require different model complexity. The model complexity needs to fit the difficulty of the data distribution to expect high accuracy after training. Too complex models or too simple models can lead to overfitting or underfitting [HTF01]. While the impact of MCPs on the error metric f is opaque, the impact on the evaluation cost g is more transparent. For example, in K-nearest-neighbor classifiers, the hyperparameter K for the number of nearest neighbors has a linear impact

to the evaluation cost. In random forest, the number of trees and the depth of each tree both have linear impact to the evaluation cost.

With this observation, we decompose the hyperparameter vector $\mathbf{c}$ into two parts: $\mathbf{c} = (\mathbf{x}, \mathbf{y})$, where $\mathbf{x} \in \mathcal{R}^{d_0}$ represents the complexity-related hyperparameters, and $\mathbf{y} \in \mathcal{R}^{d_1}$ represents the remaining hyperparameters. $d_0 + d_1 = d$. We also assume the cost $g(\mathbf{c})$ is linear with respect to $x_1, \ldots, x_{d_0}$, i.e., $g(\mathbf{c}) = \Theta(\prod_{i=1}^{d_0} x_i)$. Note that if some hyperparameter has a non-linear impact to the cost, we may be still able to do a transformation of the hyperparameter to make it satisfy the assumption. For example, if g is proportional to x^2, we can let $x' = x^2$, and search for x' instead of x. Therefore, the assumption is not overly restrictive. The assumption is only needed for developing theoretical argument and recommended practice, and not a hard requirement for FLO. Also note that we do not need a formal definition of model complexity such as AIC or BIC [Aka98, Sch78], which is used to perform model selection without using validation data and not used for computational efficiency reasons.

The model complexity related hyperparameters are of particular interest, because they make the cost function g more transparent. Otherwise, we would have two blackbox functions f and g to optimize. To illustrate how the information about g helps the search, we consider a extremely simple scenario, where all we have to tune is one MCP. That is, $d_0 = d = 1$, f is a blackbox function of a single variable x, and g is a linear function of x. To further simplify it, we assume $f(x)$ has its optimum for all $x \geq x^*$, so it suffices to find one x in that range. In this simplified scenario, Problem (1) has a solution with constant-ratio approximation guarantee of optimality: start the search from the smallest x in $\mathcal{S}$, x_0, and geometrically increase x with base b at each iteration. With this solution, i^* is the smallest integer such that $x_0^{i^*-1} \geq x^*$, and $\sum_{1 \leq i \leq i^*} g(\mathbf{c}_i) = \Theta(x_0(1 + b + \cdots + b^{i^*-1})) = \Theta(\frac{x_0(b^{i^*}-1)}{b-1}) = \frac{b^2}{b-1}\Theta(x_0 b^{i^*-2}) < \frac{b^2}{b-1} g(x^*)$. So the solution has a 4-approx guarantee when $b = 2$, i.e., the total cost is no larger than the cost of directly evaluating f for the optimal value x^*.

The optimality of geometric scheduling was first proved by Provost et al. [PJO99], and the optimal base value was presented by Huang et al. [HWDC19]. The parameter x considered in that work is the training data sample size, and the goal is to find a sample size which is sufficiently large. Although their method is limited to this special case (one dimension, flat range for optimum values at the end), it provides the basic intuition for solving Problem (1) in more challenging scenario: start from configurations with low evaluation cost, and quickly move to searching for configurations with higher cost.

In the special case, there is only direction of search (increase the value), and the search schedule can be fully specified offline. That is because there is only one parameter and we know there is a flat range for optimum values at the end a priori. All of these nice properties break in generic cases. First, since we do not know in which region the optimal values can be found in general, we have more than one direction to search even when there is only one dimension. We should search for more expensive models only if necessary. Second, for the same reason, prefixed search schedule and fixed geometric base cannot be used. Third, when tuning more than one hyperparameter, the number of directions becomes infinite because we can allocate different magnitude of changes for different dimensions. In typical gradient-based search, we can use gradient information to decide the direction. But since we are dealing with blackbox function f, we do not have such information. Fourth, the search may get stuck at local optimum point. Fifth, we need to handle both MCPs and other hyperparameters.

To solve these several challenges, we design a new zeroth-order optimization method. It performs search towards promising directions even when gradient information is unavailable. It only requires evaluations of f to make the decision. Our design is influenced by the recent theoretical work on zeroth-order optimization methods for online learning as alternatives of the corresponding first-order methods, as they estimate gradients using function value difference [DJWW15, Sha17]. We detail our solution in the next subsection.

One additional challenge for large datasets is that even simple models can take long time to train. A natural idea is to use sampled data to reduce the evaluation cost, but that also generally leads to worse accuracy. Some recent work has proposed to use progressive sampling to begin with small sample size while having chance to use larger sample size to refine the evaluation. For example, Hyperband [LJD+17] and BOHB [FKH18] use a heuristic successive halving strategy [JT16] to allocate sampling budget across configurations generated by random search and Bayesian optimization respectively. ABC [HWDC19] proposes a method to select an approximately best configuration among a given set of configurations with theoretical guarantee on the accuracy

loss and total cost. These methods are not directly applicable to our newly designed zeroth-order method. We incorporate a new progressive data sampling strategy using our insight about model complexity in a reverse direction. In the beginning of this section, we mention that model complexity must fit a particular dataset to expect highest accuracy after training. The implication of that is for a fixed model complexity, there exists a best sample size to fit the complexity. Using larger than best sample size to evaluate a small model is a waste of resource, because there will be a larger and better model to fit the larger sample anyway. And using smaller than best sample size to evaluate a large model may hide the true power of the large model and mislead the searcher to stop increasing model complexity. Intuitively, we should begin with small sample size as our search begins with low model complexity. At the time we find increasing model complexity does not improve the performance, we should increase sample size. The sample size should be increased geometrically for the same reason as the MCPs. Following that intuition, we estimate the expected time for improvement (ETI) for using the current sample size and using a larger sample size. If the ETI for using the current sample size is larger, we choose to increase the sample size.

To estimate the ETI for the current sample size s, we check the time since the last improvement, and the time taken for the last improvement. To formalize, let $t_0 < t_1$ be the two time points when the latest two best configurations are found, and t be the current time. The ETI for using the current sample size is then estimated as

$$E = \max(t - t_1, t_1 - t_0) \tag{2}$$

The ETI E' for using a larger sample size, say $2s$, is estimated as twice the evaluation time for the best configuration based on the assumption that the same configuration will have better accuracy when the sample size is doubled. Note that this assumption is not critical and both estimations do not have to be, and cannot be precisely correct. What we want to achieve is for the sample size to be sufficient for the model complexity under active search, and not too much more than necessary. Ultimately, we will use the full data to search for the best model, and the data sampling strategy is for preventing unnecessary resource spend on the low complexity models.

3.3 Algorithm

The FLO algorithm is detailed in Algorithm 1. It takes the search space $\mathcal{S}$ as input, including d_0 MCPs and $d_1 = d - d_0$ other hyperparameters. For convenience we write $\mathbf{c} = (\mathbf{c}^0, \mathbf{c}^1)$, where $\mathbf{c}^0$ and $\mathbf{c}^1$ correspond to the d_0 MCPs and d_1 non MCPs. s_0 is the initial sample size. For the non MCPs, we can take some default configurations as the initial points (optional). For example, all scikit-learn learners as well as XGBoost has default settings. If this default configuration is not provided, FLO will just use a random initial start point. For MCPs, we always begin with the smallest values which correspond to lowest evaluation cost. The geometric base B begins as 2, although this number is not critical as it will adaptively change during the search (lines 20-22). We maintain a counter n for the number of times we fail to find an improvement over the current configuration, with which we use to monitor the search.

The search in FLO is performed continuously and results are reported and they are found. It can be stopped at any time. Until stopped, it keeps trying new configurations (lines 4-27) iteratively, and reports a model if its performance is found to be better then previous results (line 9 and 12). At each iteration, it tries one or two configurations which are generated based on the current configuration $\mathbf{c} = (\mathbf{c}^0, \mathbf{c}^1)$. The two new configurations $\mathbf{c}^+$ and $\mathbf{c}^-$ are along two opposite directions (line 5), and the direction $\mathbf{u}$ is randomly generated (line 4). Note that we either vary the MCPs and fix the rest ($p = 0$) or vice versa ($p = 1$). For example, when $p = 0$, we generate two new MCP vectors $\mathbf{c}^{0,+} = (c_1^0 B^{u_1}, \ldots, c_{d_0}^0 B^{u_{d_0}})$ and $\mathbf{c}^{0,-} = (c_1^0 B^{-u_1}, \ldots, c_{d_0}^0 B^{-u_{d_0}})$. If $d_0 = 2$, for example, and $u_1 = 1, u_2 = 0$, then the geometric increase is fully allocated to c_1^0. If u_j corresponds to increase of c_j^0, then $-u_j$ corresponds to decrease. Line 6 projects the new configurations to the feasible space $\mathcal{S}$. That is, if $c_j^{\pm} < l_j$, then $c_j^{\pm} \leftarrow l_j$; if $c_j^{\pm} > h_j$, then $c_j^{\pm} \leftarrow h_j$. Line 8 evaluates $f(\mathbf{c}^+)$ and compare it with the current configuration's performance $f(\mathbf{c})$. If $\mathbf{c}^+$ improves $\mathbf{c}$, we report the trained model and move to this new configuration. Otherwise, line 11 evaluates $f(\mathbf{c}^-)$ and does the same comparison. If neither new configuration is an improvement, line 14 alternates the search between MCPs and non-MCPs, unless one of the two subsets is empty (line 15). If we fail to find an improvement for either set, we increase the counter n, and compare to a threshold U which is determined based on the number of dimensions. When n is equal to U, line 20 first tries to decrease the geometric base because $\mathbf{c}$ may be close to a local optimum. When B is so

Algorithm 1 FLO

Inputs: $\mathcal{S} = [l_1, h_1] \times \cdots \times [l_d, h_d], d_0, s_0$, default configuration for non-MCP $\mathbf{y}_0$ (optional)

Initialization: $\mathbf{c}^0 = \mathbf{x}_0 = (l_1, \ldots, l_{d_0}), \mathbf{c}^1 \leftarrow \mathbf{y}_0$ if $\mathbf{y}_0$ is provided else random point in $[l_{d_0+1}, h_{d_0+1}] \times \cdots \times [l_d, h_d]$, $B = B_0 = 2, p = p_0 \leftarrow 0$ if $d_0 > 0$ else 1, $n = 0$, $U = 2^{d_p - 1}$, $s = s_0$, $E = E' = 0$

for $i = 1, 2, \ldots$ **do**
  if $E \leq E'$ **then**
    Sample a d_p-dimensional unit vector $\mathbf{u}$ uniformly at random
    Propose two candidate directions: $\mathbf{c}^{p,+} \leftarrow \mathbf{c}^p \cdot B^{\mathbf{u}}$, and $\mathbf{c}^{p,-} \leftarrow \mathbf{c}^p \cdot B^{-\mathbf{u}}$
    Project $\mathbf{c}^{p,\pm}$ to $\mathcal{S}$
    Set $\mathbf{c}^{1-p,\pm} = \mathbf{c}^{1-p}, \mathbf{c} \leftarrow (\mathbf{c}^0, \mathbf{c}^1)$, $\mathbf{c}^{\pm} \leftarrow (\mathbf{c}^{0,\pm}, \mathbf{c}^{1,\pm})$
    if $f(\mathbf{c}^+) < f(\mathbf{c})$ **then**
      Set $\mathbf{c}^p \leftarrow \mathbf{c}^{p,+}$ and report the model trained with $\mathbf{c}^+$
    else
      if $f(\mathbf{c}^-) < f(\mathbf{c})$ **then**
        Set $\mathbf{c}^p \leftarrow \mathbf{c}^{p,-}$ and report the model trained with $\mathbf{c}^-$
      else
        Set $p \leftarrow 1 - p$
        if $d_p = 0$ **then** $p \leftarrow 1 - p$
        if $p = 1$ or $d_1 = 0$ **then**
          $n \leftarrow n + 1$
          if $n = U$ **then**
            $n \leftarrow 0$
            Set $B \leftarrow \sqrt{B}$
            if $B \leq \min_{k=0,1} \min_{j \in [d_k]} (1 + \frac{1}{c_j^k})^{\sqrt{d_k}}$ **then**
              Set $B = B_0 = \min\{B_0^2, \max_{k=0,1} \max_{j \in [d_k]} (\frac{h_j}{l_j})^{\sqrt{d_k}}\}$
              if $d_0 > 0$ **then**
                Reset $\mathbf{c}^0 \leftarrow \mathbf{x}_0$
              else
                $\mathbf{c}^1 \leftarrow \mathbf{y}_0$ if $\mathbf{y}_0$ is provided else random point
              Set $p \leftarrow p_0, s \leftarrow s_0$
  else
    $s \leftarrow 2s$
    Reevaluate $f(\mathbf{c})$ on sample size s
  update ETI E and E'

small that the expected change in all dimensions in one iteration is smaller than 1 (line 21), we consider the search converged to a local optimum. Here we assume the hyperparameters have been transformed such that the change of value by 1 is considered insignificant. Then, lines 23-27 reset the search. When search is reset, we would like to move more quickly to unexplored space, so we increase the geometric base B, unless it is so large that the expected move for all dimensions will move out of boundary (line 22).

The main computational cost of the FLO algorithm is in line 9 and 12, when evaluating the function f at $\mathbf{c}^{\pm}$. The other computations in the algorithm are negligible: sampling the unit vector randomly, elementwise power and multiplication for the vector, and simple comparison. The algorithm is extremely lightweight. The search can return its first result in a short time because configurations with low cost will be searched in the beginning. Gradually FLO will move to configurations with higher cost if that direction leads to lower error (via line 8 to 12). The reachable configurations at each iteration have between $1/B$ and B times as large values at each dimension from the current configuration. On expectation each dimension changes by $B^{\pm\sqrt{d}}$ times. If increasing complexity in that range does not improve the error, FLO will refrain from trying even higher complexity until convergence (line 22) in this low complexity region. Note that since we use multiplicative expansion (line 5), the region is not overly restrictive. If increasing complexity does lead to improved accuracy, the allowed search region will expand quickly (geometrically in expectation). Therefore, if high complexity is required for the task, FLO is still expected to reach that region in a time comparable with the evaluation cost of the configurations in that region, thanks to the power of geometric series.

Compared to the fixed geometric scheduling that works for the special simple case presented in Section 3.2, FLO has several carefully designed mechanisms to make it efficient and robust for generic cases. First, it uses randomized search directions with geometric steps. This means that each dimension will change geometrically, but the effective geometric base for each dimension is randomly allocated based on the random vector $\mathbf{u}$ at each iteration. The design retains the power of geometric series and introduces the power of randomization. Second, we consider two opposite directions at each iteration. The intuition is that if one direction leads to increased error, the opposite direction is likely to decrease the error. In line 10, given that we already observe two configurations $\mathbf{c}$ and $\mathbf{c}^+$ and find the direction from $\mathbf{c}$ to $\mathbf{c}^+$ leads to error increase (unfortunately), we would rather use that information and try the opposite direction $\mathbf{c}^-$ than wasting this observation (with high acquisition cost) before trying another random direction. This design tries to make best use of each evaluation given that these evaluations are expensive and are the only sources for guiding our search. Third, the geometric base B changes adaptively during the search. This is necessary because as we get close to a local optimum, we need to reduce the magnitude of changes from the current configuration to make the optimum point reachable. Every time we decrease the geometric base, we need to perform twice as many iterations as before to reach the same distance from the current configuration. So as we get closer and closer to the local optimum, the base will keep decreasing. Once the base is decreased so much that each dimension by expectation would not change by over 1, we conclude that we have reached a local optimum and restart the search. We call this a new *round* of search. For the new round, we begin with a higher base than the initial base of the last round (line 22) to increase the chance of exploring a different region of the search space. Last, we separate MCPs and non MCPs because they have different impact to the error and the evaluation cost. MCPs must be tuned to fit a specific task because the appropriate model complexity largely varies from task to task. They also have most significant impact to the evaluation cost. That is the reason we always initialize MCPs from the smallest values when search starts and restarts (line 24). Non-MCPs often have default values that work for a large variety of tasks, and mature learners often already pre-tuned these values. Hence we take the hint from the learners and initialize the non MCPs using the default configuration. In the case of unavailable defaults, randomly initialization is chosen since that has no significant impact to the evaluation time.

It is possible to further improve FLO by choosing the random direction with preference to unexplored area when search restarts. That requires some model-based design like Bayesian optimization. For this work, we keep the lightweight model-free design, and leave that possible extension for future work.

4 FLAML

This section presents our AutoML system FLAML built based on FLO. In Section 4.1, we first describe how to extend FLO to handle multiple learners. Then Section 4.2 specifies the detailed design. Finally, Section 4.3 discusses the extensibility of FLAML.

4.1 Learner Selection

Consider k learners, each with its own set of hyperparameters to optimize. That means we have k blackbox loss functions $f_1(\mathbf{c}_1), \ldots, f_k(\mathbf{c}_k)$, and cost functions $g_1(\mathbf{c}_1), \ldots, g_k(\mathbf{c}_k)$. Our goal for efficient joint learner selection and hyperparameter optimization is to design a search sequence $\{(m_i, \mathbf{c}_i)\}$:

$$\begin{aligned} &\min_{i^*, \{(m_i, \mathbf{c}_i)\}_{i=1}^{i^*}} \sum_{i=1}^{i^*} g_{m_i}(\mathbf{c}_i) \\ &s.t. f_{m_{i^*}}(\mathbf{c}_{i^*}) = \min_{j \in [k]} \min_{\mathbf{c} \in \mathcal{S}_j} f_m(\mathbf{c}) \end{aligned} \tag{3}$$

This problem generalizes Problem (1). If there is one learner $k = 1$, the problem is equivalent to Problem (1). A natural idea to solve Problem (3) is to use our solver FLO for every learner $1, \ldots, k$ and return the best among them. This idea is difficult to be applied because it is unclear when to stop FLO for each learner. The next natural idea is to apply one search iteration of FLO for every learner in turn. The issue of this simple strategy is that it assigns all the learners the same priority, including the low-performing ones. Eventually we only need to find the best learners and its best configuration. That means all the resources spent on the worse learners are the costs we would like to avoid. If we know which learner is the best a priori, we should apply FLO on it only and ignore all the other learners. While we must explore different learners because we do not know which learner fits the task best in advance, prioritizing well-performing and fast learners is an intuitively better strategy than round-robin search. In the mean time, we need to be aware that during the search, the performance of each learner changes dynamically, so we must prioritize adaptively.

While these ideas resemble to the exploration-exploitation trade-off in best arm identification, we must point out that the optimization goal and the required condition to apply theoretically provable identification algorithms are not satisfied by our problem. Thus we design a new solution based on a heuristic using the ETI notion described in Section 3.2. In FLO, ETI is used for deciding when to increase sample size. Here we generalize it for learner selection. For each learner $m \in [k]$, the ETI is the expected time for improvement over the current best loss among all the learners. ETI for a learner m at a particular time can be estimated depending on whether it has the best loss among all the learners:

(a) Learner m currently has the best loss among all the learners. In this case, we follow the same estimation formula Eq. (2), because learner m just needs to find an improvement over its own best configuration.

(b) Learner m does not have the best loss among all the learners currently. In this case, assume the current learner with the best loss is m^*. For m to find an improvement over the current best loss f_{m^*}, it needs to improve its own loss f_m by at least $\Delta_m = f_m - f_{m^*}$. Δ_m represents the accuracy gap between learner m and m^*. To estimate the time to fill that gap, we also need to estimate the *speed of improvement* S_m for learner m. That is, how much loss reduction does learner m expect to achieve in its own search sequence. We calculate S_m as:

$$S_m = \frac{I_m}{\tau_m} \tag{4}$$

where I_m is the loss reduction between the latest two best configurations for learner m, and τ_m is the time taken in search between finding these two configurations. In the special case where $I_m = 0$, i.e., the first configuration searched for learner m is the best configuration for m so far, we set $I_m = f_m$. Finally, we calculate ETI for learner m as $\frac{\Delta_m}{S_m}$.

In the beginning, FLAML performs one search iteration using FLO for each learner. Afterwards, it chooses one learner with the lowest ETI to perform one FLO iteration. After every iteration, the ETIs for the learners are updated. Using ETI, we dynamically adjust the priority according to each learner's search dynamics. The choice of learner does not only depend on the current loss and evaluation time, but also the time needed to find improvement and how large the improvement is.

Therefore, even if a learner is fast and finds low loss, if it is stuck for too long without finding a better configuration, it will yield priority to other learners. This ensures that with long enough time, no learner will be left out, and the global optimum loss among all learners can be reached.

FLAML remains fast as it optimizes for the overall evaluation cost instead of number of iterations. FLAML remains lightweight too. For each learner, we only need to keep track of the current configuration, the time and loss of the latest two configurations, and the sample size. Therefore, switching from learner to learner has no overhead. We remark that learner selection is a categorical choice. And our solution can be generalized to optimize categorical and numeric hyperparameters jointly.

4.2 System Specification

FLAML supports classification and regression tasks. For classification, it supports XGBoost, LightGBM, CatBoost, sklearn.GradientBoosting, sklearn.RandomForest, sklearn.ExtraTrees, sklearn.AdaBoost, sklearn.KNeighbors, sklearn.LinearSVC, sklearn.DecisionTree, sklearn.BernoulliNB, H2O.GLM and fully connected DNN. For regression, it supports the same learner except sklearn.LinearSVC and sklearn.BernoulliNB, plus sklearn.GaussianNB. New learners can be easily added into the system, which we describe in the next subsection. By default, it enables XGBoost, LightGBM and sklearn.RandomForest.

FLAML supports all the standard accuracy metrics. For metrics that are desired to be maximized, we use the distance from upper bound as the loss function, e.g., 1-accuracy.

FLAML can support both cross validation and holdout as the evaluation method of each configuration. For cross validation, the reported model is trained on the full training data. For holdout, the model during loss function evaluation is trained on 66% of the training data, and validated on the remaining. When FLO restarts the search for one learner, and the learner has the best loss, we train a model using that learner's best configuration with full training data and report it. By default, FLAML uses 5-fold cross-validation for datasets with # instances $\times$ # features below 200K, and holdout for datasets larger than that.

FLAML can be run on a single core or multiple cores. FLAML is designed to work with low resource consumption, so it evaluates one configuration at a time and allocates all the resources to the learner. The extra computation in FLAML is negligible. Since FLO starts search from low complexity models for every learner, this design minimizes the latency between two iterations so that the algorithm can get feedback as early as possible. We discuss how to extend it for high resource use case in the next subsection.

For progressive sampling, FLAML shuffles the data randomly in the beginning and to get a sample with size s, it takes the first s tuples of the shuffled data. By default, the initial sample size is 10K tuples.

4.3 Extensibility

Adding a learner. To add a learner, we need to specify which hyperparameters are the MCPs for FLO, and the range of each MCP. The lower bound of a MCP can be set corresponding to the simplest meaningful model, and the upper bound can be set corresponding to the most complex model that is affordable to train. Given that FLO carefully controls the search order, the range can be set loosely to make sure the best hyperparameters are included. No extra preparation work is needed for adding a learner, in contrast to the libraries relying on meta learning [FSE18, SZB+19].

Adding feature preprocessor. To support feature preprocessor search, we can regard each combination of feature preprocessor and learner as a categorical choice, and apply the same ETI-based selection. An alternative is to decompose the choice as two steps: choose a feature preprocessor and choose a learner. We can maintain separate ETIs for feature preprocessors and learners. The advantage of decomposition is that we can have fewer choices at each step to conquer the combinatorial decision space, and can be generalized to longer sequences of pipelines. In the case of feature preprocessor being expensive, we should cache the processed training data.

Adding ensemble postprocessor. Stacked ensemble can be added as a post-processing step like existing libraries [H2O]. It requires remembering all the models' predictions on cross-validation folds that have been evaluated, or the best models' predictions for each learner. And extra time

needs to be spent on building the ensemble and retraining each model. FLAML does not do it to keep the overhead low, but it is a standard add-on when storage and extra computation time are not concerns.

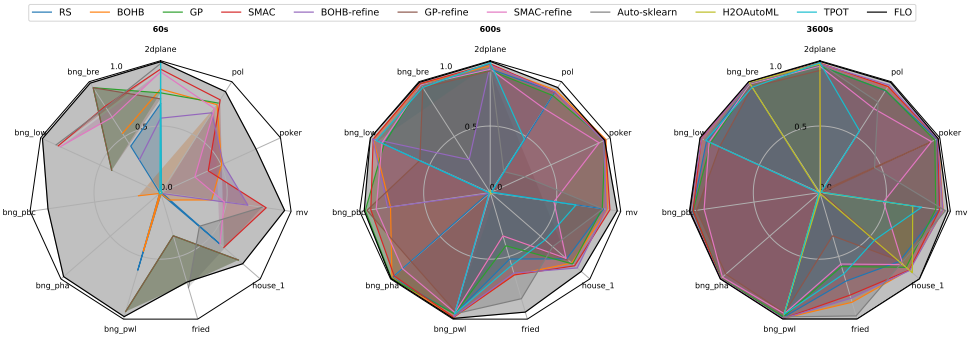
Meta learning. There are several ways FLAML can benefit from meta learning. First, FLAML gives each learner one FLO iteration in the beginning, because it does not have ETIs to start with. If we use meta learning to train a estimator of the evaluation cost for each learner's initial configuration, we can use that estimator as the initial ETI, such that a super expensive learner will not be tried in the beginning. Note that even though this requires running the learner for some datasets to build the cost model, the task of estimating the cost is significantly easier than estimating the accuracy. Also, since it is only used for ETI, the precision requirement for the cost model is low. So we expect the amount of preparation for adding a new learner is still significantly lower than the other libraries. Second, FLAML uses a few constants such as the initial sample size, and the threshold of data size for using cross-validation vs. holdout. These constants can be tuned from meta learning. Third, the initial configuration of each learner is set as the simplest model. While this ensures the search time is optimal if the simplest model is the best model, there can be a better start point if the minimal required model complexity is predictable. One can consider using meta learning to build such a predictor. While the predictor for minimal required model complexity does not have to be precise, we would prefer underestimate over overestimate.

Parallelization. When abundant cores are available, we can extend FLAML to perform multiple searches in parallel. After choosing one learner with lowest ETI to perform one FLO iteration, if there are available cores, we can choose the next learner with the second lowest ETI to perform one FLO iteration, and so on. When one FLO iteration for a learner finishes, the resource is released and we can select a learner again using updated ETIs. If all the learners have been running FLO search and there are still available resources, we can choose the learner with lowest ETI to create a new search thread. That thread performs a new round of FLO search from the restart point (line 19 in Algorithm 1). Recall that a round of FLO search ends at the restart point of the next round. Since each search round in FLO is independent, this new search thread does not interfere with the old thread. The only communication needed across search threads is the global best configuration's loss. As long as more resource is available, we can keep creating new FLO threads in this way. Each FLO thread has its own ETI, so the learner selection component naturally extends to FLO thread selection.

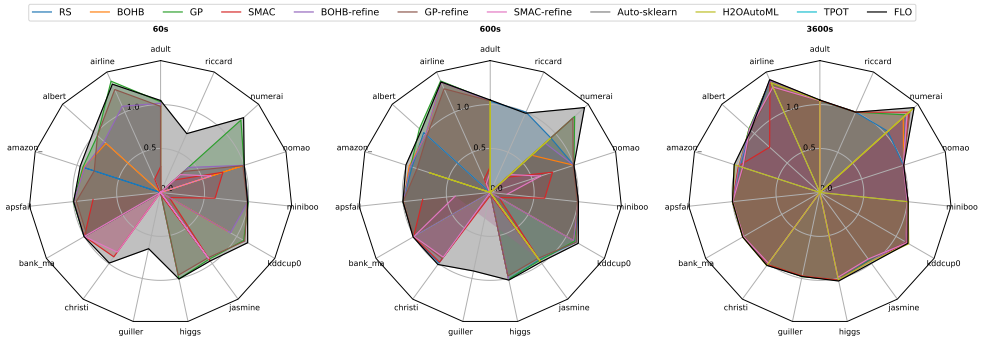
5 Experiments

We now describe an extensive experimental study of FLO and FLAML's performance using a latest open source AutoML benchmark [GLT+19]. This AutoML benchmark includes 39 classification tasks (with 22 binary classification tasks and 17 mulit-class classification tasks). Each task consists of a dataset, a metric to optimize, and specific resources to use. The 39 classification datasets were selected by this benchmark from previous AutoML papers, competitions and benchmarks, according to a predefined list of criteria. These datasets vary in the number of samples and features by orders of magnitude, and vary in the occurrence of numeric features, categorical features and missing values. Datasets that can be easily solved with AutoML or did not represent typical AutoML scenarios were excluded. The datasets in this benchmark are about 15 times larger on average than prior effort [BA18], in spite of fewer datasets. The list of datasets is available on OpenML. Since this AutoML benchmark does not include regression tasks, we enriched it with regression tasks from a machine learning evaluation and comparison benchmark called PMLB [OLCO+17]. Following the same spirit of the classification benchmark, among the 120 OpenML regression datasets in PMLB, we selected the ones whose numbers of instances are larger than 10,000. That results in 14 regression tasks. We categorize the resulting 53 datasets into small datasets and large datasets according to whether #instance × #feature is larger than 200K. In this enriched benchmark, roc-auc is used as the optimization metric for binary tasks, log-loss for multi-class tasks, and r2 score for regression tasks. All experiments are executed under the same computational conditions using 1 core. As each dataset has 10 cross-validation folds, all the results reported in this paper are averaged over the 10 folds.

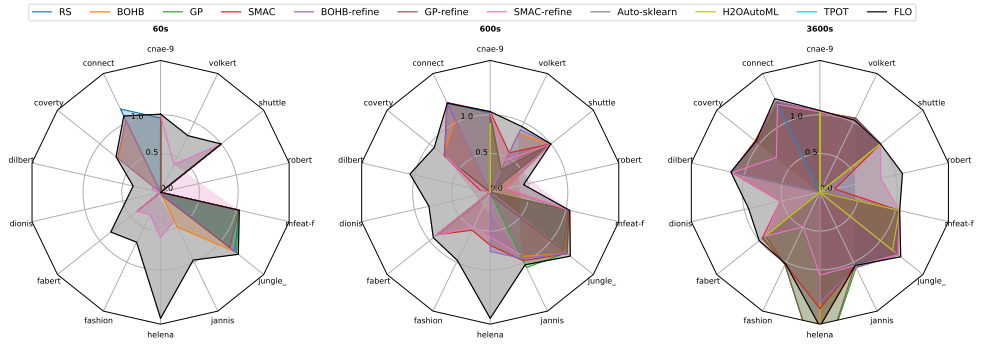
In the following two subsections, we first evaluate FLO for hyperparameter optimization. Then we evaluate FLAML to compare its performance with existing AutoML libraries.



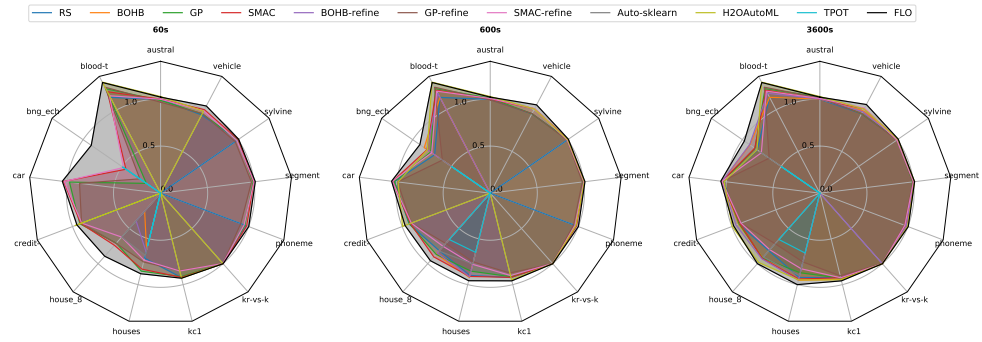
(a) Scaled scores on large regression datasets



(b) Scaled scores on large binary classification datasets



(c) Scaled scores on large multi-class classification datasets



(d) Scaled scores on small datasets (including all small regression, binary, multi-class datasets)

Figure 1: Performance comparison of XGBoost hyperparameter tuning on 53 datasets from the enriched AutoML benchmark, with varying time budget of 60s, 600s, 3600s

Table 2: Hyperparameters tuned in XGBoost

hyperparameter name	type	range	complexity related
tree num	int	[4, # instance]	✓
leaf num	int	[4, # instance]	✓
inverse of min child weight	float	[1/20, 1/0.01]	✓
learning rate	float	[4, 1024]	✗
subsample	float	[0.01, 1.0]	✗
reg alpha	float	[1e-10, 1.0]	✗
reg lambda	float	[1e-10, 1.0]	✗
colsample by level	float	[0.6, 1.0]	✗
colsample by tree	float	[0.7, 1.0]	✗

5.1 Evaluation of FLO

5.1.1 Baselines and evaluation setup

With the purpose of evaluating hyperparameter optimization efficacy, we compare FLO to 10 methods and categorized them into 4 types.

- Random search (RS).
- Bayesian optimization (BO): We included SMAC, GP, and BOHB, where BOHB is a combination of Bayesian optimization (TPE) with Hyperband and is shown to be faster than the vanilla Bayesian optimization on some tasks because of the dynamic data subsampling in Hyperband.
- Bayesian optimization with altered objective function: In addition, we included a refined version of each of the three BO baselines, which uses a simple heuristic to account for the different evaluation time of MCPs: It discounts the original BO's minimization objective, i.e. the loss, by adding a model complexity related factor $\frac{g(\mathbf{c}_i)-\min_{\mathbf{c}\in\mathcal{S}} g(\mathbf{c})}{\max_{\mathbf{c}\in\mathcal{S}} g(\mathbf{c})-\min_{\mathbf{c}\in\mathcal{S}} g(\mathbf{c})}$, where $\mathcal{S}$ is the hyperparameter search space in our experiments. By doing so, these refined BO methods take model complexity into consideration when searching for hyperparameters. The reason for including them is to verify whether some simple revisions of BO methods can properly take advantage of the existence of MCPs to reduce the total evaluation time for hyperparameter optimization. Note that this refined version of BO is in spirit very similar to the heuristic technique *expected improvement per second* proposed in [SLA12] but more computationally efficient because it does not need to build another Bayesian model to estimate the evaluation time on the fly. In our following discussion, We differentiate these refined versions of BO by adding '-refine' suffix to the original BO methods.
- AutoML for a single learner: auto-sklearn, H2O AutoML, and TPOT can tune a single learner exclusively if specified. The reason for including these AutoML libraries is that they are the top three performers reported by the benchmark [GLT+19], and we want to verify whether FLO can have competitive performance compared with methods that work out of the box, even though they have different design choices, such as hyperparameter setting, meta-learning, and model ensemble.

We evaluate FLO and all the hyperparameter optimization methods' performance in tuning XGBoost and select XGBoot as the learner to use in the AutoML libraries. XGBoost is a good learner to test for several reasons: (1). XGBoost is known to have superior performance on many machine learning tasks and it is one of the most commonly used learner in many machine learning competitions and applications. (2). XGBoost is the only commonly supported learner by auto-sklearn (regression only), H2O AutoML and TPOT. (3). It has both model-complexity-related hyperparameters and model-complexity-unrelated hyperparameters. In Table 2 we list the parameters tuned in XGBoost in our experiments. Inverse is taken on min child weight because min child weight is negatively correlated with model complexity. For FLO, we set the upper bound of tree and leaf numbers as the number of instances. That is a loose bound and we let FLO figure out what complexity is needed itself. For BO, we tried two different settings. The first setting is the same as FLO. The second setting is based on the choices of other AutoML libraries, such

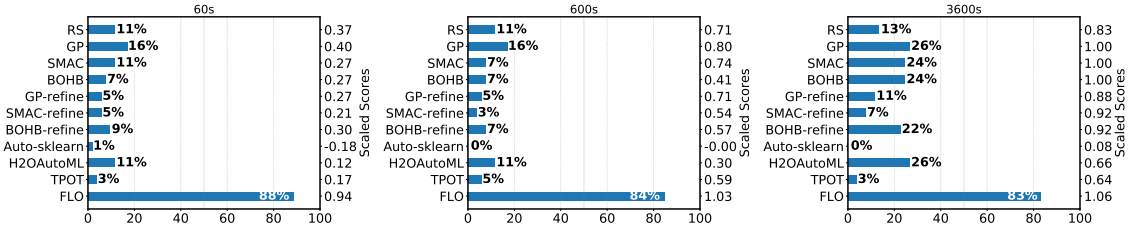


Figure 2: The success rate, i.e., the percentage of datasets each hyperparameter optimization method reaches best score (within 0.5% score difference) for each time budget

as H2O AutoML and auto-sklearn. The upper bounds for tree and leaf numbers in the second setting are 500 and 1024 respectively. While the second setting has a more narrow range than the first setting, BO performs better in the second setting in the given time budget, because the loose range makes BO evaluate very time-consuming hyperparameters sometimes. So to be fair to the BO methods, we report BO's performance using the second setting.

In accordance with the optimization metrics used in previous work, we use r2 score (for regression tasks), roc-auc (for binary classification tasks), and negative log-loss (for mulit-class classification tasks) as the scores to evaluate the performance of different methods (the higher the better). To better visualize the performance across different datasets, we scaled the original scores using the same way as in the referred AutoML benchmark. For classification tasks, the score is scaled between a constant class prior predictor (=0) and a tuned random forest (=1), the original scores of which are already provided by this benchmark. It is worth mentioning that the tuned random forest can take long to finish, ranging from 13 minutes to 4 hours for the classification tasks. On some datasets, none of the existing AutoML solutions outperforms it. So it is a strong baseline and achieving a score above 1 is not easy. Regression tasks are not originally included in the referred AutoML benchmark, and the same tuned random forest baseline could not finish on some of the regression datasets due to memory error. So we scale the scores for regression tasks between the worst performing method (=0) within 1m, and the best performing method (=1) within 1h among all the evaluated methods in our experiments.

5.1.2 Performance comparison

Performance of all the methods The scaled scores of FLO and all the baselines given different desired time budgets (1m, 10m and 1h) on all the datasets are shown in Figure 1. In Figure 1, each of the radar chat shows the scaled scores of different methods on a set of datasets (spokes on the radar chat) given a desired time budget. Each row of radar chat, i.e, each sub-figure in Figure 1 shows the performance comparison on the same set of datasets given different desired time budgets. Results of all 53 datasets are summarized into 4 sub-figures according to their dataset sizes and task types.

Figure 1 demonstrates the superior performance of FLO from the following perspectives: *Advantage 1: Good anytime performance given the different time budgets.* Given all the magnitudes of time budgets tried (1m, 10m, and 1h), FLO achieves the best score on most of the datasets (with three obvious exceptions on large multi-class datasets, including connect in 1m, Jannis in 10m and Helena in 1h). *Advantage 2: Significant lead on large datasets.* FLO is in a significant lead of performance on large datasets, especially when time budget is small, which can be seen from the magnitude of gaps between FLO and the other baselines in Figure 1 (a), (b) and (c) with time budget 60s, and 600s. *Advantage 3: Fast convergence to the best performance.* By comparing the methods across the three radar charts in each row of Figure 1, we can see how each method performs under different time budgets. By reading the charts in this way, we can find that on many datasets, FLO's results in 1m are already competitive to the best results achieved by all the methods in 1h, which indicates FLO's fast convergence to optimal or near optimal hyperparameters.

Overall comparison Figure 2 depicts how often each method is a winner or close enough to the winner at different time budgets. We refer to the fraction of datasets on which a method reaches the best score as the *success rate* of that method. We use a tolerance ratio of 0.5% to exclude the marginal differences on the scores, i.e. when the difference between two scores is within the

Table 3: Distribution of score difference between FLO and baselines in 1m

	FLO win	median	quartile	max	FLO lose	median	quartile	max
RS	90%	+0.05	+0.22	+1.52	2%	-0.10	-0.10	-0.10
GP	84%	+0.05	+0.20	+7.05	4%	-0.02	-0.03	-0.04
SMAC	90%	+0.22	+0.79	+7.93	0%	*	*	*
BOHB	94%	+0.27	+0.56	+1.64	0%	*	*	*
GP-refine	94%	+0.09	0.40	+7.05	0%	*	*	*
SMAC-refine	96%	+0.27	+0.53	+6.11	0%	*	*	*
BOHB-refine	94%	+0.52	+1.00	+5.26	0%	*	*	*
Auto-sklearn	93%	+0.44	+1.23	+5.23	7%	-0.04	-0.04	-0.04
H2OAutoML	90%	+0.06	+0.06	+0.06	4%	-0.02	-0.02	-0.03
TPOT	98%	+0.02	+0.07	+0.40	0%	*	*	*

Table 4: Distribution of score difference between FLO and baselines in 10m

	FLO win	median	quartile	max	FLO lose	median	quartile	max
RS	87%	+0.04	+0.11	+0.52	2%	-0.01	-0.01	-0.01
GP	81%	+0.05	+0.10	+0.62	6%	-0.03	-0.03	-0.04
SMAC	90%	+0.06	+0.41	+1.21	2%	-0.01	-0.01	-0.01
BOHB	89%	+0.06	+0.14	+10.7	2%	-0.01	-0.01	-0.01
GP-refine	94%	+0.08	+0.20	+1.18	0%	*	*	*
SMAC-refine	96%	+0.15	+0.66	+5.43	0%	*	*	*
BOHB-refine	100%	+0.07	+0.49	+5.27	0%	*	*	*
Auto-sklearn	100%	+0.16	+0.58	+3.05	0%	*	*	*
H2OAutoML	89%	+0.06	+0.14	+0.52	4%	-0.02	-0.02	-0.03
TPOT	92%	+0.06	+0.28	+0.66	0%	*	*	*

tolerance ratio, they are considered as close enough.

We can see that FLO has a dominating advantage over all the others in terms of both the success rate and the average scaled score. No other method has a nearly close performance overall. The three BO methods outperform RS, except that BOHB underperforms in 1m and 10m. One hypothetical reason for BOHB to underperform is that when using small training sample to evaluate the hyperparameters, the accuracy of models with large complexity is underestimated, and the search is then misguided. GP has a slight overall lead over the other two BO methods, but adding model complexity to the objective makes it worse. The refined versions of Bayesian optimization methods only helped BOHB in 1m and 10m. BOHB is improved because the preference to smaller models reduces its waste of time evaluating large models with small sample of training data, and leaves more time for evaluating on larger training samples. The other two methods, as well as BOHB in 1h are not improved because after adding model complexity to the objective, the optimization target of these methods compromise accuracy for evaluation time. That makes them unable to optimize towards the best hyperparameters over time. H2O AutoML based on RS has better performance than vanilla RS. That is because H2O AutoML performs random search following three default configurations of different model size (medium depth, deep and shallow).

Drilldown comparison To give the readers a more precise idea about how significant the advantages are, we show a drill-down performance comparison between FLO and the other methods in Table 3-6.

Table 3-5 present the distribution of score difference between FLO and each individual baseline, for time budget 1m, 10m and 1h respectively. They show the fraction of datasets where FLO wins against or loses to each baseline, as well as the median, upper quartile and max margins between the scaled scores. For example, with 10m time budget, FLO outperforms random search on 87% datasets, and loses to random research on 2% datasets. For the rest 11% datasets, they perform similarly. The median, upper quartile and max margins for FLO over RS in the winning cases are 0.04, 0.11 and 0.52 respectively, while the max margin in the losing case is 0.01. We can see that FLO wins in 72%-100% against each baseline and the margin is significant. Each baseline has a few cases with very bad performance, hence the max margin is large. In the few cases where FLO loses, the margin is quite small. These demonstrate the *Advantage 2* of FLO, as well as the

Table 5: Distribution of score difference between FLO and baselines in 1h

	FLO win	median	quartile	max	FLO lose	median	quartile	max
RS	81%	+0.03	+0.08	+5.32	4%	-0.02	-0.02	-0.02
GP	74%	+0.03	+0.08	+0.42	9%	-0.03	-0.03	-0.61
SMAC	72%	+0.04	+0.07	+0.09	6%	-0.01	-0.02	-0.03
BOHB	68%	+0.03	+0.06	+0.18	9%	-0.01	-0.02	-0.02
GP-refine	85%	+0.06	+0.10	+0.99	8%	-0.02	-0.18	-0.61
SMAC-refine	89%	+0.06	+0.17	+0.66	2%	-0.02	-0.02	-0.02
BOHB-refine	91%	+0.06	+0.14	+0.21	0%	*	*	*
Auto-sklearn	100%	+0.10	+0.20	+1.00	0%	*	*	*
H2OAutoML	74%	+0.03	+0.06	+0.27	6%	*	*	*
TPOT	92%	+0.05	+0.21	+0.59	0%	*	*	*

Table 6: Percentage of datasets where FLO has better or equal loss vs. the baselines (with tolerance ratio 0.5%), when using equal (left half) or smaller time budget (right half)

Baseline	1m vs 1m	10m vs 10m	1h vs 1h	10m vs 1h	1m vs 10m	1m vs 1h
RS	98%	98%	96%	83%	88%	62%
GP	96%	94%	91%	79%	81%	56%
SMAC	100%	98%	94%	75%	81%	44%
BOHB	100%	98%	91%	72%	81%	40%
GP-refine	100%	100%	92%	89%	96%	75%
SMAC-refine	100%	100%	98%	91%	92%	73%
BOHB-refine	100%	100%	100%	94%	88%	77%
Auto-sklearn	93%	100%	100%	93%	93%	71%
H2OAutoML	96%	96%	94%	87%	92%	62%
TPOT	100%	100%	100%	100%	94%	92%

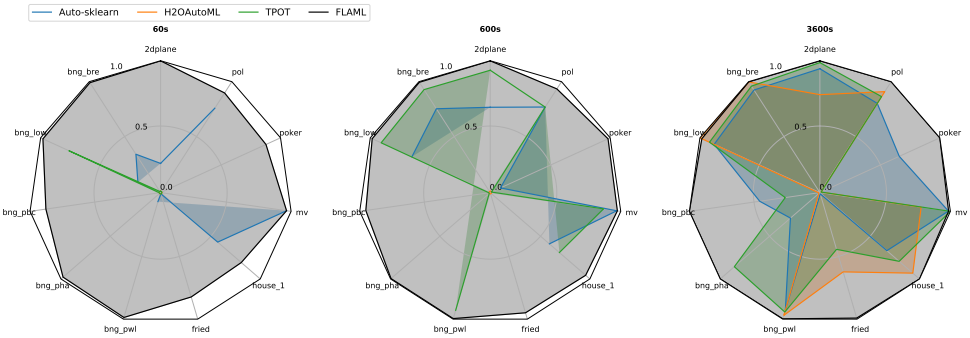
robustness of it.

In Table 6, each row shows the percentage of datasets where FLO is better than or equal to a particular baseline with different time budgets, which is reflected from the table header of the corresponding column as 'time budget for FLO vs. time budget for the baseline'. For example '1m vs 10m' in the table header means FLO's time budget is 1m and the concerned baseline's time budget is 10m. The left half in Table 6 summarizes Table 3-5 and shows *Advantage 1* of FLO. In the right half of Table 6 we compare FLO with the baselines under a more harsh condition to FLO by giving FLO less time than the baselines. The results show that FLO's 1m result is already better than or equal to baselines' 1h result on 40%-92% datasets, and 71%-100% when 10m is given to FLO. These results again verified the fast convergence of FLO as mentioned in *Advantage 3*.

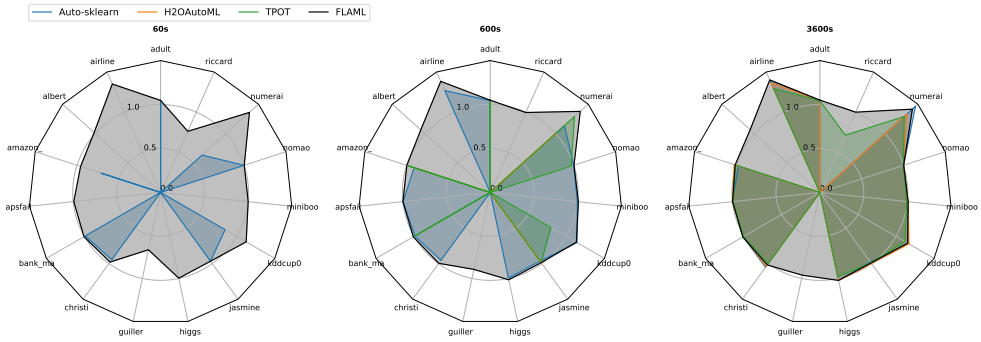
5.2 Evaluation of FLAML

In this section, we compare FLAML, auto-sklearn, H2O AutoML and TPOT on the enriched AutoML Benchmark. The purpose is to evaluate the out-of-box performance of each AutoML library. No restriction is made to the learners, preprocessing, meta-learning, or ensemble. All the libraries have enough memory to use on a server with 512GB RAM. FLAML's memory usage is the lowest, ranging from a few hundreds of megabytes to around 10GB. H2O AutoML consumes 100-200GB because it uses stacked ensembles. TPOT, and sometimes H2O AutoML, does not conform to the time budget strictly. We still allow them to finish, which gives them an advantage in search time.

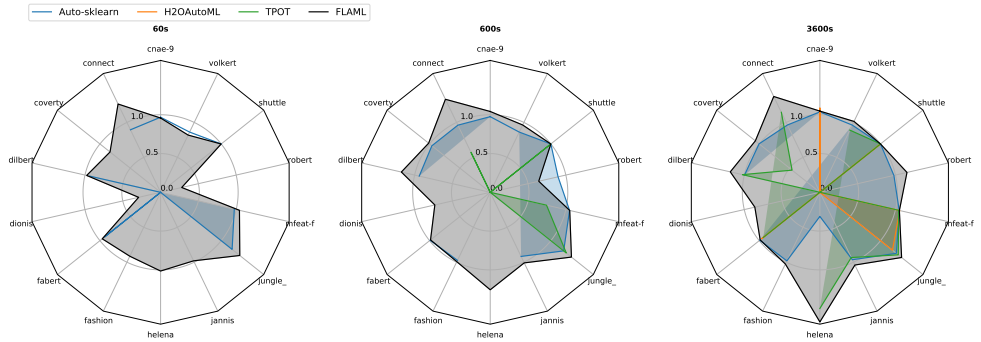
Figure 3 plots the scaled scores of all the methods given different desired time budgets (1m, 10m and 1h) on all the datasets. The observation is quite similar to Figure 3. FLAML demonstrates *Advantage 1:* good anytime performance given the different time budgets, *Advantage 2:* significant lead on large datasets, and *Advantage 3:* fast convergence to the best performance. There are only one two obvious losing cases for FLAML vs. a baseline. One is on Robert with time budget 10m, where FLAML loses to auto-sklearn. A posterior analysis reveals that auto-sklearn finds the best model with the first trial. That success is partly attributed to meta learning and partly to luck. We can see that when the time budget is 1h, FLAML outperforms auto-sklearn significantly on the



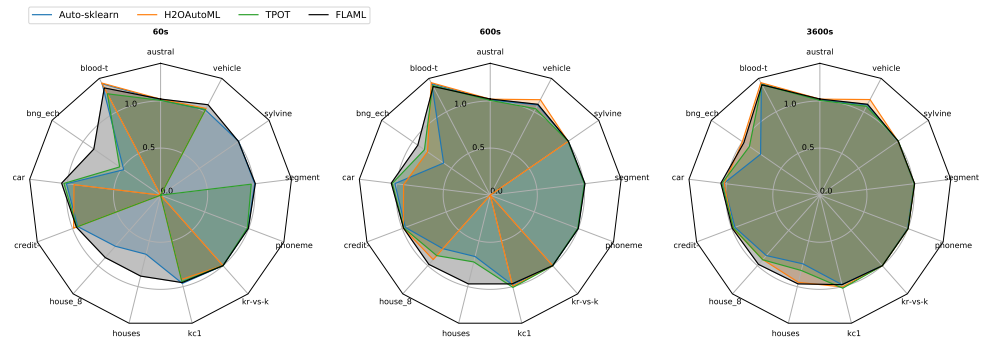
(a) Scaled scores on large regression datasets



(b) Scaled scores on large binary classification datasets



(c) Scaled scores on large multi-class classification datasets



(d) Scaled scores on small datasets (including all small regression, binary, multi-class datasets)

Figure 3: Performance comparison of AutoML libraries on 53 datasets from the enriched AutoML benchmark, with varying time budget of 60s, 600s, 3600s

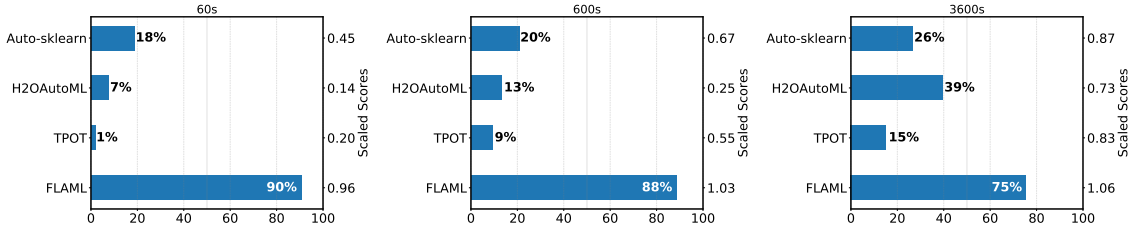


Figure 4: The percentage of datasets each AutoML solution reaches best score (within 0.5% score difference) for each time budget

Table 7: Distribution of score difference between FLAML and other libraries

1m	FLAML win	median	quartile	max	FLAML lose	median	quartile	max
Auto-sklearn	79%	+0.28	+0.83	+4.26	8%	-0.03	-0.04	-0.05
H2OAutoML	92%	+0.04	+0.09	+0.13	4%	-0.05	-0.05	-0.06
TPOT	98%	+0.03	+0.07	+0.33	0%	*	*	*
10m	FLAML win	median	quartile	max	FLAML lose	median	quartile	max
Auto-sklearn	75%	+0.16	+0.37	+3.52	9%	-0.01	-0.03	-0.26
H2OAutoML	85%	+0.13	+0.13	+0.36	6%	-0.05	-0.05	-0.06
TPOT	87%	+0.08	+ 0.17	+1.51	4%	-0.03	-0.03	-0.04
1h	FLAML win	median	quartile	max	FLAML lose	median	quartile	max
Auto-sklearn	64%	+0.10	+0.34	+1.36	9%	-0.01	-0.03	-0.05
H2OAutoML	57%	+0.07	+0.12	+0.37	19%	-0.02	-0.03	-0.05
TPOT	79%	+0.07	+0.17	+1.61	4%	-0.03	-0.03	-0.04

same dataset. Another case is on a small dataset vehicle, where FLAML loses to H2O AutoML in 10m and 1h. A posterior analysis reveals that the high performance of H2O AutoML in that dataset is attributed to a generalized linear model which was not included in FLAML's default search space, as well as stacked ensemble, which was not implemented in FLAML.

Figure 4 depicts the success rate, i.e., how often each method is a winner or close enough to the winner at different time budgets. It also annotates the average scaled score of each method over all datasets. We can see that FLAML has a significant advantage over all the others in terms of both the success rate and the average scaled score. Among the baselines, auto-sklearn has a lead in the average scaled score. It also has a higher success rate with time budget 1m and 10m. Given one hour time budget, H2O AutoML has more successes (39% vs. 26%) than auto-sklearn. Each of them has their own winning case, as shown from Figure 3. None of them gets close to FLAML's position: 75%-90% success rate and 0.96-1.06 average scaled score.

The drill-down performance comparison between FLAML and the other methods are presented in Table 7 and 8. Table 7 presents the distribution of score difference between FLAML and each individual baseline, for time budget 1m, 10m and 1h respectively. We can see that FLAML wins in a majority of datasets against each baseline and the margin is significant, as mentioned in *Advantage 2*. Except for a few outliers, the maximal margin of loss is smaller than the median winning margin against each method. For example, in 1m, while FLAML loses to auto-sklearn in 8% cases, the max loss is only 0.05, and the median and upper quartile winning margins are 0.28 and 0.83 over 79% cases.

In Table 8, each row shows the percentage of datasets where FLAML is better than or equal to a particular baseline with different time budgets. The left half in Table 8 summarizes Table 7 and

Table 8: Percentage of datasets where FLAML has better or equal loss vs. the baselines (with tolerance ratio 0.5%), when using equal (left half) or smaller time budget (right half)

Baseline	1m vs 1m	10m vs 10m	1h vs 1h	10m vs 1h	1m vs 10m	1m vs 1h
Auto-sklearn	92%	91%	91%	81%	79%	62%
H2OAutoML	96%	94%	81%	79%	87%	58%
TPOT	100%	96%	96%	91%	91%	75%

shows *Advantage 1* of FLAML. In the right half of Table 8 we compare FLAML with the baselines under a more harsh condition to FLAML by giving FLAML less time than the baselines. The results show that FLAML's 1m result is already better than or equal to baselines' 10m result on 79%-90% datasets, and the same for 10m vs 1h. These results again verified the fast convergence of FLAML as mentioned in *Advantage 3*.

6 Discussion and Future Work

Despite the competitive performance of FLO and FLAML, we do not intend to claim they can replace Bayesian optimization and existing AutoML libraries without limitations.

The first limitation of FLO is that it requires identification of model-complexity related hyperparameters. While FLO can be applied to the case where there is no such hyperparameter, we have not conducted a comparative evaluation against BO methods. The second limitation is that it does not work for categorical hyperparameters, without using the extension by FLAML. Even with FLAML, the search space we have evaluated so far only contains one categorical hyperparameter. With many years of research, there are BO methods which can deal with much more complex search space. That is a lead which certainly requires more research to catch up, before FLO can be used for more generic hyperparameter optimization problem.

As for FLAML, we have discussed many feature gaps from existing AutoML libraries in Section 4.3. Some of them are due to the lightweight design goal. We see two paths moving forward. First, we can keep the lightweight design, and find ways to add missing features without making it heavy weight, e.g., by introducing lightweight meta learning. Second, we can use the lightweight library as a building block, and find ways to integrate with other components of AutoML such as automated feature engineering. Both lead to interesting future work.

Even with these limitations, the strong empirical results in this work are encouraging. There are a few findings worth reflection. First, this work shows that introducing more control over search order based on model complexity has a big impact to saving search time and cost for AutoML. We believe this is not the only way to leverage the search space structure for more efficient search. Second, this work shows that zeroth-order search method has a great potential in hyperparameter optimization for AutoML. More research is worth investing to address the limitations of this very first attempt, including both algorithms and theoretical understanding. Third, this work suggests that it is possible to make AutoML fast and lightweight while achieving reasonable accuracy. That is a promising sign for AutoML to be integrated in resource-constrained and latency-sensitive software in future.

7 Conclusion

This work addresses the time and resource efficiency of AutoML towards software integration. It tackles the inefficiency of hyperparameter optimization methods designed for generic blackbox optimization, by leveraging a unique property of machine learning tasks. The proposed new method FLO is fast and lightweight, which makes it suitable for building into systems processing large scale training data. It outperforms Bayesian optimization and random search due to a stronger control over the search order which optimizes for saving overall cost. It is adaptive, extensible, and has strong anytime performance. A simple extension of FLO already makes it a highly competitive AutoML solution compared to the state of the art. Our AutoML solution FLAML remains fast and lightweight. It has not used meta learning, ensemble or feature preprocessors, which makes it promising to be used as a basic building block while exploring the multiple directions of improvement in future work.

References

[Aka98] Hirotogu Akaike. *Information Theory and an Extension of the Maximum Likelihood Principle*, pages 199–213. Springer New York, New York, NY, 1998.

[BA18] Adithya Balaji and Alexander Allen. Benchmarking automatic machine learning frameworks, 2018.

[BB12] James Bergstra and Yoshua Bengio. Random search for hyper-parameter optimization. *J. Mach. Learn. Res.*, 13:281–305, February 2012.

[BBBK11] James S Bergstra, Rémi Bardenet, Yoshua Bengio, and Balázs Kégl. Algorithms for hyper-parameter optimization. In *Advances in neural information processing systems*, pages 2546–2554, 2011.

[BCdF10] Eric Brochu, Vlad M. Cora, and Nando de Freitas. A tutorial on bayesian optimization of expensive cost functions, with application to active user modeling and hierarchical reinforcement learning. *CoRR*, abs/1012.2599, 2010.

[CG16] Tianqi Chen and Carlos Guestrin. Xgboost: A scalable tree boosting system. In *KDD*, 2016.

[DJWW15] John C Duchi, Michael I Jordan, Martin J Wainwright, and Andre Wibisono. Optimal rates for zero-order convex optimization: The power of two function evaluations. *IEEE Transactions on Information Theory*, 61(5):2788–2806, 2015.

[FKE+15] Matthias Feurer, Aaron Klein, Katharina Eggensperger, Jost Springenberg, Manuel Blum, and Frank Hutter. Efficient and robust automated machine learning. In *Advances in neural information processing systems*, pages 2962–2970, 2015.

[FKH18] Stefan Falkner, Aaron Klein, and Frank Hutter. BOHB: Robust and efficient hyperparameter optimization at scale. In *Proceedings of the 35th International Conference on Machine Learning (ICML 2018)*, 2018.

[FSE18] Nicolo Fusi, Rishit Sheth, and Melih Elibol. Probabilistic matrix factorization for automated machine learning. In *Advances in Neural Information Processing Systems*, pages 3348–3357, 2018.

[GLT+19] Pieter Gijsbers, Erin LeDell, Janek Thomas, Sébastien Poirier, Bernd Bischl, and Joaquin Vanschoren. An open source automl benchmark. *CoRR*, abs/1907.00909, 2019.

[GSM+17] Daniel Golovin, Benjamin Solnik, Subhodeep Moitra, Greg Kochanski, John Karro, and D Sculley. Google vizier: A service for black-box optimization. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2017.

[H2O] H2O.ai. H2o automl. `http://docs.h2o.ai/h2o/latest-stable/h2o-docs/automl.html`. 2019-10-29.

[HHLB11] Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown. Sequential model-based optimization for general algorithm configuration. In Carlos A. Coello Coello, editor, *Learning and Intelligent Optimization*, 2011.

[HTF01] Trevor Hastie, Robert Tibshirani, and Jerome Friedman. *The Elements of Statistical Learning*. Springer New York Inc., New York, NY, USA, 2001.

[HWDC19] Silu Huang, Chi Wang, Bolin Ding, and Surajit Chaudhuri. Efficient identification of approximate best configuration of training in large datasets. In *Proceedings of the 33rd AAAI Conference on Artificial Intelligence (AAAI)*, 2019.

[JT16] Kevin Jamieson and Ameet Talwalkar. Non-stochastic best arm identification and hyperparameter optimization. In *Artificial Intelligence and Statistics*, 2016.

[KGG+18] Patrick Koch, Oleg Golovidov, Steven Gardner, Brett Wujek, Joshua Griffin, and Yan Xu. Autotune: A derivative-free optimization framework for hyperparameter tuning. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, 2018.

[LJD+17] Lisha Li, Kevin Jamieson, Giulia DeSalvo, Afshin Rostamizadeh, and Ameet Talwalkar. Hyperband: A novel bandit-based approach to hyperparameter optimization. In *ICLR'17*, 2017.

[OBUM16] Randal S Olson, Nathan Bartley, Ryan J Urbanowicz, and Jason H Moore. Evaluation of a tree-based pipeline optimization tool for automating data science. In *Proceedings of the Genetic and Evolutionary Computation Conference 2016*, pages 485–492. ACM, 2016.

[O'H78] A. O'Hagan. Curve fitting and optimal design for prediction. *Journal of the Royal Statistical Society: Series B (Methodological)*, 40(1):1–24, 1978.

[OLCO+17] Randal S. Olson, William La Cava, Patryk Orzechowski, Ryan J. Urbanowicz, and Jason H. Moore. Pmlb: a large benchmark suite for machine learning evaluation and comparison. *BioData Mining*, 10(1):36, Dec 2017.

[OUA+16] Randal S. Olson, Ryan J. Urbanowicz, Peter C. Andrews, Nicole A. Lavender, La Creis Kidd, and Jason H. Moore. Automating biomedical data science through tree-based pipeline optimization. In Giovanni Squillero and Paolo Burelli, editors, *Applications of Evolutionary Computation*, pages 123–137. Springer International Publishing, 2016.

[PJO99] Foster Provost, David Jensen, and Tim Oates. Efficient progressive sampling. In *ACM SIGKDD international conference on Knowledge discovery and data mining*, 1999.

[PVG+11] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, Jake Vanderplas, Alexandre Passos, David Cournapeau, Matthieu Brucher, Matthieu Perrot, and Édouard Duchesnay. Scikit-learn: Machine learning in python. *JMLR*, 12:2825–2830, November 2011.

[Sch78] Gideon Schwarz. Estimating the dimension of a model. *Ann. Statist.*, 6(2):461–464, 03 1978.

[Sha17] Ohad Shamir. An optimal algorithm for bandit and zero-order convex optimization with two-point feedback. *Journal of Machine Learning Research*, 18(52):1–11, 2017.

[SLA12] Jasper Snoek, Hugo Larochelle, and Ryan P Adams. Practical bayesian optimization of machine learning algorithms. In *Advances in neural information processing systems*, pages 2951–2959, 2012.

[SZB+19] Zeyuan Shang, Emanuel Zgraggen, Benedetto Buratti, Ferdinand Kossmann, Philipp Eichmann, Yeounoh Chung, Carsten Binnig, Eli Upfal, and Tim Kraska. Democratizing data science through interactive curation of ml pipelines. In *Proceedings of the 2019 International Conference on Management of Data*, SIGMOD'19, 2019.

[THHLB13] Chris Thornton, Frank Hutter, Holger H Hoos, and Kevin Leyton-Brown. Auto-weka: Combined selection and hyperparameter optimization of classification algorithms. In *Proceedings of the 19th ACM SIGKDD international conference on Knowledge discovery and data mining*, 2013.