ECONOMICAL HYPERPARAMETER OPTIMIZATION WITH BLENDED SEARCH STRATEGY

Chi Wang*, **Qingyun Wu***, **Silu Huang, Amin Saied**
Microsoft Corporation, Redmond, WA 98052, USA
{wang.chi,qingyun.wu,silu.huang,amin.saied}@microsoft.com

ABSTRACT

We study the problem of using low cost to search for hyperparameter configurations in a large search space with heterogeneous evaluation cost and model quality. We propose a blended search strategy to combine the strengths of global and local search, and prioritize them on the fly with the goal of minimizing the total cost spent in finding good configurations. Our approach demonstrates robust performance for tuning both tree-based models and deep neural networks on a large AutoML benchmark, as well as superior performance in model quality, time, and resource consumption for a production transformer-based NLP model fine-tuning task.

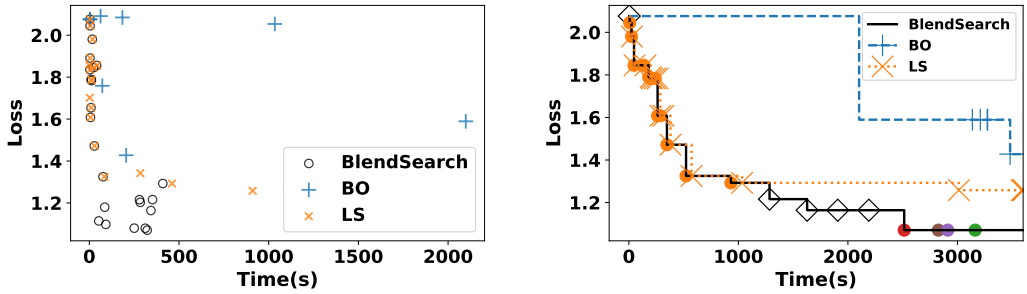
1 INTRODUCTION

Hyperparameter optimization (HPO) of modern machine learning models is a resource-consuming task, which is unaffordable to individuals or organizations with little resource (Yang & Shami, 2020). Operating HPO in a low-cost regime has numerous benefits, such as democratizing ML techniques, enabling new applications of ML, which requires frequent low-latency tuning, and reducing the carbon footprint. It is inherently challenging due to the nature of the task: trying a large number of configurations of *heterogeneous* cost and accuracy in a large search space. The expense can accumulate from multiple sources: either a large number of individually cheap trials or a small number of expensive trials can add up the required resources.

There have been multiple attempts to address the efficiency of HPO from different perspectives. Each of them has strengths and limitations. For example, Bayesian optimization (BO) (Brochu et al., 2010), which is a class of global optimization algorithms, is used to minimize the total number of iterations to reach global optima. However, when the cost of different hyperparameter configurations is heterogeneous, vanilla BO may select a configuration that incurs unnecessarily high cost. As opposed to BO, local search (LS) methods (Wu et al., 2021) are able to control total cost by preventing very expensive trials until necessary, but they may get trapped in local optima. Multi-fidelity methods (Jamieson & Talwalkar, 2016) aim to use cheap proxies to replace some of the expensive trials and approximate the accuracy assessment, but can only be used when such proxies exist. A single search strategy is difficult to meet the generic goal of economical HPO.

In this work, we propose a blended search strategy which combines global search and local search strategy such that we can enjoy benefits from both worlds: (1) global search can ensure the convergence to the global optima when the budget is sufficient; and (2) local search methods enable a better control on the cost incurred along the search trajectory. Given a particular global and local search method, our framework, which is named as `BlendSearch`, combines them according to the following design principles. (1) Instead of sticking with a particular method for configuration selection, we consider both of the candidate search methods and decide which one to use at each round of the configuration selection. (2) We use the global search method to help decide the starting points of local search threads. (3) We use the local search method to intervene the global search method's configuration selection to avoid configurations that may incur unnecessarily large evaluation cost. (4) We prioritize search instances of both methods according to their performance and efficiency of performance improvement on the fly. Extensive empirical evaluation on the AutoML

*Equal contribution



(a) Loss vs. evaluation time for configs tried by each method. One point represents one config. The lower is the loss the better is the quality of the config. Longer evaluation time corresponds to larger training cost

(b) Best loss vs. optimization time per method. Colored circles – configs proposed by LS threads in our method (color change indicates thread change); diamonds – configs proposed by BO in our method

Figure 1: A typical example of the different behaviors of BO, LS and our proposed `BlendSearch` in tuning a set of 11-dim hyperparameters for XGBoost. BO is prone to selecting expensive but not necessarily good configs. LS avoids expensive configs in the beginning but is prone to getting stuck in local regions. `BlendSearch` switches between one BO and multiple LS search threads and prioritizes the more promising ones, and turns out to try more low-cost, high-quality configs.

Benchmark (Gijsbers et al., 2019) validates the robust performance of our method on a wide variety of datasets. `BlendSearch` is now publicly available in an open-source AutoML Library[1].

2 Background and Related Work

We first briefly introduce the vanilla Bayesian optimization methods and local search methods, which are among the building blocks of our method. Bayesian optimization is a class of global optimization algorithms which is suitable for optimizing expensive black-box functions. It models the probabilistic distribution of the objective conditioned on the optimization variables. Typical models include Gaussian process (Snoek et al., 2012), random forest (Hutter et al., 2011), and tree Parzen estimator (TPE) (Bergstra et al., 2011). In BO methods, an acquisition function is used to determine the next point to evaluate. Two common acquisition functions are the expected improvement (EI) (Bull, 2011) over the currently best-observed objective and upper confidence bound (UCB) (Srinivas et al., 2009). Local search methods are prevalent in the general optimization literature (Spall et al., 1992; Nesterov & Spokoiny, 2017) but less studied in the HPO literature due to the possibility of getting trapped in local optima (György & Kocsis, 2011). Recent work (Wu et al., 2021) shows that a local search method `FLOW`^{2} can make HPO cost-effective when combined with low-cost initialization and random restart. At each iteration, it samples a pair of vectors (with opposite directions) uniformly at random from a unit sphere, the center of which is the best configuration found so far (a.k.a. incumbent) and the radius of which is the current stepsize. Expensive configurations are avoided in the beginning as each iteration proposes a configuration near the incumbent. Random restart of the local search is performed once the convergence condition is satisfied.

There are several attempts to address the limitations of vanilla BO or local search methods. BOwLS (BO with local search) (Gao et al., 2020) uses a BO model to select the starting point of a local search thread. Each local search thread is run until convergence and the BO model is updated with the start point and the converged loss. Trust region BO (Eriksson et al., 2019) fits a fixed number of local models and performs a principled global allocation of samples across these models via an implicit bandit approach. It is primarily designed for HPO problems with high-dimensional numerical hyperparamters. Unfortunately, all existing work that tries to combine global search with local search methods does not consider the heterogeneity of evaluation cost incurred along with the search. There are also a lot of attempts in making HPO efficient by speeding up configuration evaluation. Multi-fidelity optimizations (Klein et al., 2017; Li et al., 2017; Kandasamy et al., 2017; Falkner et al., 2018; Lu et al., 2019; Li et al., 2020) are proposed for this purpose. They usually require an additional degree of freedom in the problem called 'fidelity', to allow performance assessment on a configura-

[1] `https://github.com/microsoft/FLAML`

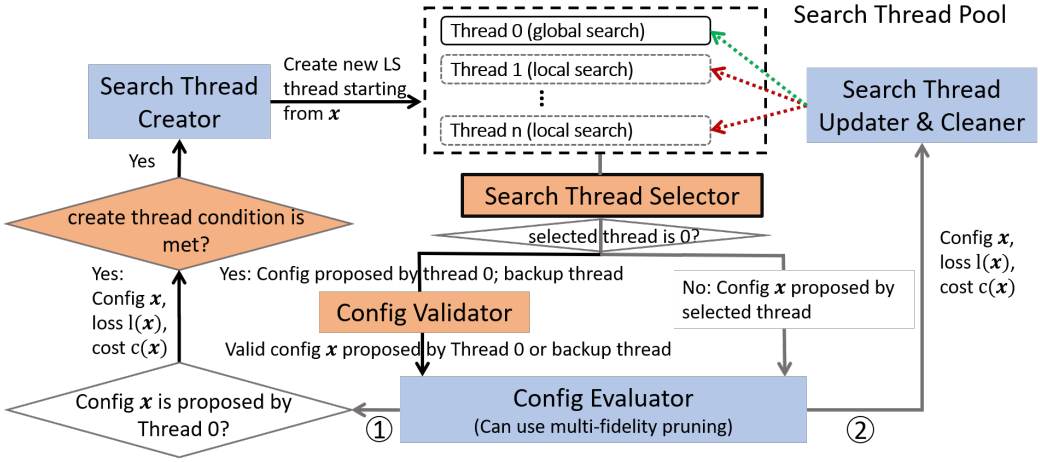


Figure 2: Framework. Both paths from 'Config Evaluator' are executed independently.

tion with different fidelities. There is surprisingly little prior work for generic cost-effective HPO. Gaussian process with expected improvement per second (GPEIPS) (Snoek et al., 2012) models the evaluation cost using another Gaussian process, and heuristically adds the estimated cost into the acquisition function. It does not always outperform GPEI as the acqusition function can overpenalize good but expensive configurations.

3 BlendSearch

Our framework needs the following information as inputs.

- B is the total budget of cost. In this work, we measure the cost by CPU/GPU time.
- P is the input HPO problem, which has the following attributes characterizing the problem.
  - $P.\mathcal{X}$ is the search space in which each element $\mathbf{x}$ is a d-dimensional hyperparameter configuration. For a non-categorical hyperparameter coordinate $i \in [d]$, if different values of $\mathbf{x}_i$ lead to heterogeneous cost and there is a known value $P.x_i^{LowCost}$ corresponding to a low cost, it is considered as a controlled dimension. We use $P.D$ to denote such a set of controlled dimensions.
  - $P.\texttt{LossFunc}(\cdot)$ is the loss function to be minimized in terms of the configurations $\mathbf{x} \in P.\mathcal{X}$.
  - $P.\texttt{CostFunc}(\cdot)$ is the cost function that outputs the cost incurred when evaluating $\mathbf{x}$.

  The goal of an HPO algorithm is to minimize the loss $P.\texttt{LossFunc}(\mathbf{x})$ with the constraint that the total cost incurred $G(\pi) := \sum_{\mathbf{x}\in\mathcal{I}(\pi)} P.\texttt{CostFunc}(\mathbf{x}) \leq B$, where $\mathcal{I}(\pi)$ is the search trajectory of algorithm π. Note that both $P.\texttt{LossFunc}(\mathbf{x})$ and $P.\texttt{CostFunc}(\mathbf{x})$ are black-box functions meaning that typically the analytic form is not available, and only function values can be observed. In order to distinguish the operation of querying from loss/cost function and the loss/cost observation, we use $P.\texttt{LossFunc}(\mathbf{x})$ and $P.\texttt{CostFunc}(\mathbf{x})$ to denote the former, and use $l(\mathbf{x})$ and $c(\mathbf{x})$ to denote the latter. l and c (omitting $\mathbf{x}$) are used when there is no ambiguity.
- $\mathcal{G}$ is the global search method to be used. $\mathcal{L}$ is the local search method to be used. $\mathcal{L}.\Delta$ is the largest stepsize used in local search method $\mathcal{L}$, i.e., the largest possible change on a hyperparameter value between two consecutive search steps.

The overall design of our framework is presented in Figure 2 and Algorithm 1. The key idea is to maintain a pool of search threads, one of which corresponds to global search and the others local search. The pool starts with one global search thread and gradually adds local search threads as the search goes on. Here a search thread is an instance of a global search or local search method, each with its own search trajectory. At each round, a *search thread selector* selects one of the search threads from the pool according to a priority metric that reflects the search threads' current performance and efficiency of performance improvement. The selected search thread will then be used to propose a configuration to evaluate in this round. When the selected search thread is the global search thread, a *config validator* first checks whether the proposed configuration is within

the 'admissible' region for evaluation. If not, it uses a backup local search thread instead. A local search thread will be created only when the global search thread proposes a valid config and a certain *create thread condition* is met, and will be deleted once it converges. Two local search threads will be merged into one if they are close enough. The priority of each search thread is updated after each evaluation round.

Algorithm 1 `BlendSearch`

Inputs: HPO problem P, a global search method $\mathcal{G}$, a local search method $\mathcal{L}$, total budget B.
1: **Initialization:** Initialize $\mathcal{F}.\mathbb{S} = [S_0]$ where S_0 is an instance of $\mathcal{G}$. We denote by $\mathcal{F}.\mathbf{x}_0$ the initial point of the search. By design, the values of the controlled dimensions of $\mathcal{F}.\mathbf{x}_0$ are set to be $P.\mathbf{x}^{LowCost}$ and those for the other dimensions are proposed by S_0.
2: **while** $\mathcal{F}.c < B$ **do**
3: $\tilde{S}, \tilde{S}_{\text{bak}} \leftarrow$ `SelectThread`$(\mathcal{F})$
4: $(\mathbf{x}, l, c) \leftarrow$ `SearchEvaluate1Step`$(\tilde{S}, \mathcal{F}, P)$
5: **if** $\mathbf{x}$ is `invalid` **then** $(\mathbf{x}, l, c) \leftarrow$ `SearchEvaluate1Step`$(\tilde{S}_{\text{bak}}, \mathcal{F}, P)$
6: **if** $\mathbf{x}$ is proposed by global search & `CreateNewLSCondition` is satisfied **then**
7: Initialize $S =$ `InitializeSearchThread`$(\mathcal{L}, P, (\mathbf{x}, l, c))$
8: Add the new LS thread S into the pool: $\mathcal{F}.\mathbb{S} \leftarrow \mathcal{F}.\mathbb{S} + S$
9: `DeleteAndMergeLS`$(\mathcal{F})$ ▷ Merge or delete existing LS threads when necessary
10: Update $\mathcal{F}$.`Priority`
11: **if** $\mathbf{x}$ is proposed by global search **then** `UpdateGSModel`$(S_0, (\mathbf{x}, l, c))$

For convenience, we use $\mathcal{F}$ to denote a collection of framework-level variables:

- $\mathcal{F}.\mathbb{S}$ is the list of search threads maintained in our framework. $\mathcal{F}.\mathbb{S}$ contains at least one search thread and among them there is one and only one global search thread, i.e., S_0, in $\mathcal{F}.\mathbb{S}$.
- $\mathcal{F}$.`Priority` is a priority dictionary, in which the keys are the search threads in $\mathcal{F}.\mathbb{S}$, and the values are the priority of the corresponding search threads.
- Bookkeeping information of $\mathcal{F}$: $\mathcal{F}.l^*$ is the best loss achieved among all the search threads and $\mathcal{F}.c$ is the total cost consumed in our framework.
- $\mathcal{F}.\mathcal{R}$ is the 'admissible' region on the controlled dimensions of the current search, which is a hyperrectangle and can be written in the form of $\mathcal{F}.\mathcal{R} := \{[\mathcal{F}.x_i^{\min}, \mathcal{F}.x_i^{\max}]\}_{i \in D}$ with $\mathcal{F}.x_i^{\min}$ and $\mathcal{F}.x_i^{\max}$ denoting the minimum and maximum value along the i-th dimension in $\mathcal{F}.\mathcal{R}$ respectively. They are initially set as $\mathcal{F}.x_i^{\min} = \mathcal{F}.x_i^{\max} = P.x_i^{LowCost}$ for all $i \in D$. The 'admissible' region gradually expands during the search: (1) it is expanded to cover all the points evaluated by all the search threads and all the points that are possible to be reached by the local search within one search step, as shown in line 7 and 8 of Algorithm 2; (2) it expands if a local search thread converges, as shown in line 3 of Algorithm 5 (included in Appendix A).

In the following, we explain the key steps in our algorithm.

Step 1: Search thread selector (line 3 of Alg 1). In addition to the primary search thread $\tilde{S}$, `SelectThread` also outputs a backup search thread $\tilde{S}_{\text{bak}}$ which is guaranteed to be a local search thread. It is set to be none when there is no local search thread yet in $\mathcal{F}.\mathbb{S}$. Specifically,

$$\tilde{S} = \arg\max_{S \in \mathcal{F}.\mathbb{S}} \mathcal{F}.\texttt{Priority}(S), \tilde{S}_{\text{bak}} = \begin{cases} \arg\max_{S \in (\mathcal{F}.\mathbb{S} \setminus S_0)} \mathcal{F}.\texttt{Priority}(S) & \mathcal{F}.\mathbb{S} \setminus S_0 \neq \emptyset \\ \text{None} & \mathcal{F}.\mathbb{S} \setminus S_0 = \emptyset, \end{cases} \tag{1}$$

The design of the priority metric follows the principle of optimism in the face of uncertainty from the multi-armed bandit problem to balance exploitation and exploration (Lattimore & Szepesvári, 2020). Specifically, linear extrapolation is performed adaptively and locally to calculate the improvement speed of each search thread. The estimated future reward based on such a linear extrapolation provides a first-order upper bound of the ground truth future reward assuming each search thread has a diminishing return, i.e., the speed of improvement decreases as more resource is spent. Formally, we introduce the following variables and functions for each search thread $S \in \mathcal{F}.\mathbb{S}$.

- Bookkeeping information: $S.l^{1\text{st}}$ and $S.l^{2\text{nd}}$ are the best loss so far and second best loss before the best loss is achieved. $S.c^{1\text{st}}$ and $S.c^{2\text{nd}}$ are the total cost taken when $S.l^{1\text{st}}$ and $S.l^{2\text{nd}}$ are achieved respectively. $S.c$ is the total cost spent in S. $S.\mathbf{x}^{1\text{st}}$ is the best configuration found so far.

- $S.s$ is the performance improvement speed of S. It is calculated as $S.s = \frac{S.l^{2\text{nd}} - S.l^{1\text{st}}}{S.c - S.c^{2\text{nd}}}$. This formula is only valid when there is at least one improvement. Otherwise, we do not have enough information to estimate the speed of improvement. We set the speed to the highest speed of all the search threads when $S.l^{2\text{nd}} = S.l^{1\text{st}}$. It is due to an implicit assumption of diminishing return.
- $S.x^{\min}$ and $S.x^{\max}$ are the minimum and maximum value of the i-th dimension of all hyperparameters configurations evaluated in S respectively.
- $S.\texttt{CostImp}(\cdot)$ is a function whose input is a target loss and output is the anticipated cost for S to generate a better loss than this target. We use the following formula to compute it, which is intuitively using the cost for improvement in the past to estimate that in the future.

$$S.\texttt{CostImp}(l) = \max\left\{S.c - S.c^{1\text{st}}, S.c^{1\text{st}} - S.c^{2\text{nd}}, 2\frac{S.l^{1\text{st}} - l}{S.s}\right\} \tag{2}$$

Our proposed priority metric is essentially the negative of the projected loss of S:

$$\mathcal{F}.\texttt{Priority}(S) = -(S.l^{1\text{st}} - S.s \times b) \tag{3}$$

in which $b = \min(\max_{S \in \mathcal{F}.\mathbb{S}} S.\texttt{CostImp}(\mathcal{F}.l^*), B - \mathcal{F}.c)$. $\max_{S \in \mathcal{F}.\mathbb{S}} S.\texttt{CostImp}(\mathcal{F}.l^*)$ can be considered as the resource needed for every S to have a better performance than the currently best performance $\mathcal{F}.l^*$. Our priority metric estimates the loss of each search thread if such an amount of resource (restricted by the budget left $B - \mathcal{F}.c$) is given. By considering both the search threads' current performance and potential improvement, it provides a fair trade-off between exploiting the currently-best and exploring the potentially-better choices.

Algorithm 2 `SearchEvaluate1Step`

Inputs: HPO problem P, search thread S, and $\mathcal{F}$

1: **if** S is None **then** Construct $\mathbf{x}$ as follows: generate the controlled dimensions of $\mathbf{x}$ by adding Guassian noises on the corresponding dimensions of $\mathcal{F}.\mathbf{x}_0$ and for the rest of the dimensions sample uniformly at random from the search space $P.\mathcal{X}$.

2: **else** $\mathbf{x} \leftarrow S.\texttt{ProposeConfig}()$

3: **if** $S = S_0$ (i.e., S is the global search thread) & $\mathbf{x} \notin \mathcal{F}.\mathcal{R}$ **then** $\mathbf{x} \leftarrow$`invalid`

4: **else** $l, c \leftarrow P.\texttt{LossFunc}(\mathbf{x}), P.\texttt{CostFunc}(\mathbf{x})$ ▷ Evaluate configuration $\mathbf{x}$

5: **if** $\mathbf{x} \neq$ `invalid` **then**

6: `BookKeeping`$(S, \mathbf{x}, l, c, \mathcal{F})$ and update speed $S.s$,

7: $\forall i \in P.D$, $S.x_i^{\min} \leftarrow \min\{\mathbf{x}_i, S.x_i^{\min}\}$, $\mathcal{F}.x_i^{\min} \leftarrow \min\{S.x_i^{\min} - \mathcal{L}.\Delta, \mathcal{F}.x_i^{\min}\}$,

8: $\forall i \in P.D$, $S.x_i^{\max} \leftarrow \max\{\mathbf{x}_i, S.x_i^{\max}\}$, $\mathcal{F}.x_i^{\max} \leftarrow \max\{S.x_i^{\max} + \mathcal{L}.\Delta, \mathcal{F}.x_i^{\max}\}$

9: **Outputs:** $\mathbf{x}, l, c$

Step 2: Config validator and evaluator (line 4-5 of Alg 1). After a search thread (and a backup search thread) is selected, the next step is to propose the next configuration to try with the chosen search thread(s). Intuitively speaking, we consider generating the next configuration to try primarily according to the selected search thread $\tilde{S}$ whose priority is ranked the highest. But we set a guard rail for the global search thread as it may propose an unnecessarily high-cost configuration. We thus introduce a *config validator* to validate the configurations proposed by global search according to whether they are within the current admissible region of our framework $\mathcal{F}.\mathcal{R}$ (line 3 of Alg 2). A configuration marked as '`invalid`' means that it is considered to be prone to incur unnecessarily high cost and will not be evaluated at this round. In this case, the selected backup search thread will be used to perform another round of `SearchEvaluate1Step` (line 5 of Alg 1) if it is a valid search thread (i.e., not none). In the case where the backup thread is none, we generate the new configuration according to line 1 of Alg 2. The config validator helps avoid potentially high-cost evaluation and thus avoid creating local search threads from high-cost points until necessary. It does not stick to local searches forever because the admissible region $\mathcal{F}.\mathcal{R}$ gets expanded. Note that according to the definition of $\mathcal{F}.\mathcal{R}$, only the controlled dimensions of the hyperparameter configurations are subject to validation check. If needed, a multi-fidelity pruning strategy can be used in this config evaluator component. Multi-fidelity pruning does not necessarily yield better performance. So the adoption of multi-fidelity pruning in `BlendSearch` is optional.

Step 3: Search thread creator, updater and cleaner (line 6-11 of Alg 1). If the newly proposed configuration is proposed by global search and it is not marked as 'invalid', we consider creating

a new local search thread using the proposed configuration as a starting point. To make sure the newly created local search thread is relatively good, we first check whether the proposed configuration's performance is better than at least half of the existing threads' performance (specified in the `CreateNewLSCondition`). If so, a new local search thread will be initialized and added to the active search thread pool $\mathbb{S}$. In `DeleteAndMergeLS`, we check whether a local search thread has converged according to the convergence condition of the specific local search method. If it is, the search thread will be removed from $\mathbb{S}$. In addition, we also go through all the local search threads to see whether the incumbent of a LS thread is reachable in one step by another LS thread with lower loss (ref. Appendix A). If so, the former LS thread will be deleted. After a configuration proposed by global search is evaluated, the observation tuple $(\mathbf{x}, l, c)$ is then used to update the model of the global search method through function `UpdateGSModel`. For example, when the global search method is a Bayesian optimization method, the model is the surrogate model used.

Due to page limit, detailed pseudocode for several of the straightforward functions mentioned in our framework are provided in Appendix A, including `CreateNewLSCondition`, `DeleteAndMergeLS`, `InitializeSearchThread` and `BookKeeping`.

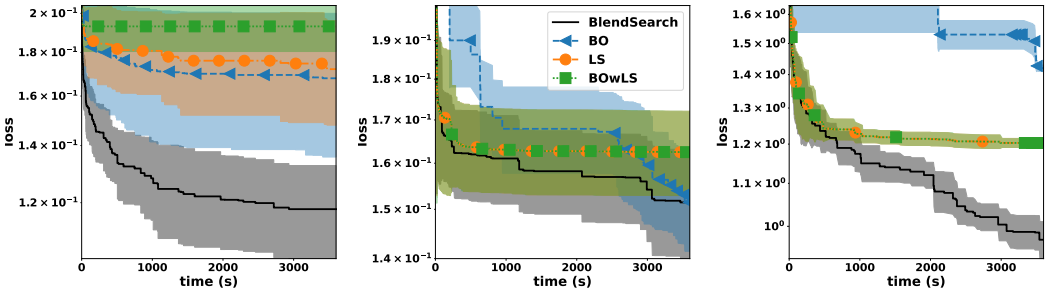
4 EXPERIMENTS

We evaluate `BlendSearch` in tuning tabular machine learning libraries with an AutoML benchmark (Gijsbers et al., 2019), and in fine-tuning NLP models for text data. The AutoML benchmark consists of 39 tabular datasets that represent real-world data science classification problems. It includes datasets of all sizes, of different problem domains and with various levels of difficulty. As each dataset has 10 cross-validation folds, all the results reported in this paper are averaged over the 10 folds. With this benchmark, we are able to evaluate multiple HPO methods on a large number of datasets within a manageable computational budget, for tuning three machine learning libraries: XGBoost (Chen & Guestrin, 2016), LightGBM (Ke et al., 2017) and DeepTables[2]. The first two are popular libraries based on gradient boosted trees, and the third is an easy-to-use deep learning toolkit which utilizes latest research findings for tabular data. We chose them because gradient boosted trees and deep neural networks are the most frequent winners in data science competitions. We run experiments for XGBoost and LightGBM on the AutoML benchmark and report the results in Section 4.1. As a real application, we report an NLP model fine-tuning task for a production use case in Section 4.2. Finally, in Section 4.3 we perform ablation study to investigate the effectiveness of several important components of our framework. Due to page limit, we include the results for tuning DeepTables in Appendix B. We include the following baselines in our experiments.

- BO (Akiba et al., 2019) – the Bayesian optimization baseline. We choose a modern HPO library Optuna and use the TPE sampler because of its flexibility in handling mixed continuous and discrete space and good peformance reported in existing work (Falkner et al., 2018).
- LS (Wu et al., 2021) – the recent baseline of using local search with random restart, based on `FLOW`^{2}. It is proved to be able to control cost effectively and outperform BO methods for numerical cost-related hyperparameter search.
- BOwLS (Gao et al., 2020) – the baseline of an existing approach of combining local search with BO, i.e., using BO to propose start points for local search.
- ASHA (Li et al., 2020), i.e., asynchronous successive halving – a state-of-the-art HPO method that uses multi-fidelity optimization and supports parallel tuning.

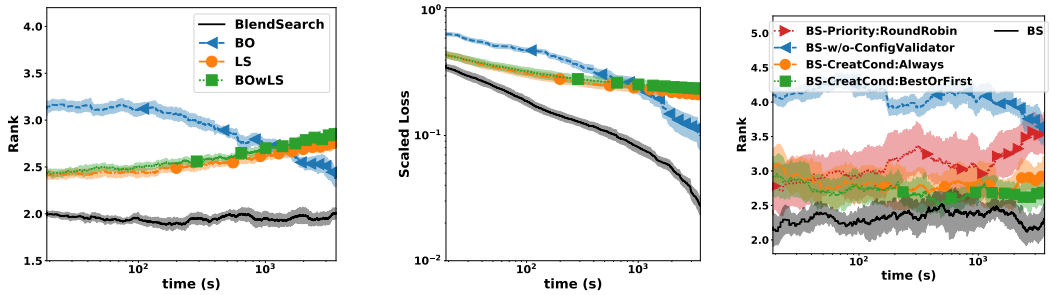
Initialization setting. For the local search method used in our framework, a low-cost initialization is needed to realize its unique advantages in controlling the cost. It is implemented via setting a low-cost initial value for each of the controlled dimensions. For example, among the hyperparameters tuned in LightGBM (shown in Table 2 of Appendix B), three hyperparameters, including 'tree num', 'leaf num' and 'min child weight' have initial values corresponding to the min or max values in their range, depending on whether they have a positive or negative correlation with the evaluation cost. It does not require the loss of the initial configuration to be low. Only one single low-cost initial value for each controlled dimension needs to be specified as input. To ensure a fair evaluation, we use the same low-cost initial point (if controlled dimensions exist) as the starting point of all the baselines.

[2] `https://github.com/DataCanvasIO/DeepTables`



(a) credit, loss = 1-auc (b) KDDCup09, loss = 1-auc (c) volkert, loss = log-loss

Figure 3: Optimization performance curve for XGBoost. Lines correspond to the mean loss over 10 folds, and shades correspond to 95% confidence intervals. 1-auc is used for binary classification and log-loss is used for multi-class classification.



(a) Rank (b) Scaled loss (c) Rank for ablation

Figure 4: Aggregated rank and scaled loss on LightGBM and XGBoost. The scaled loss is obtained by min-max scaling using the max and min loss over all time across all the methods. Sub-figure (a) & (b) share the same legend, and BS is short for BlendSearch in sub-figure (c). Both rank and scaled regret are aggregated first across datasets then across 10 folds of data in each dataset.

4.1 Tuning XGBoost and LightGBM

We tune a set of 9-dimensional hyperparameters (all numerical) in LightGBM and 11-dimensional hyperparameters (9 numerical and 2 categorical) in XGBoost. A detailed description of the search space can be found in Appendix B. In this section, we omit the multi-fidelity baseline method, i.e., ASHA, because we do not find a good 'fidelity' dimension that works well in tuning LightGBM and XGBoost (ref. additional results in Appendix B). We perform the evaluation on 37 out of the 39 datasets from the AutoML benchmark ('Robert' and 'Dionis' are excluded due to out-of-memory error and extremely long training time). The input budget B (in terms of CPU time) is set to be 4 hours for the 3 largest datasets among the 37 datasets, and 1 hour for the rest. From the performance curves shown in Figure 3, we observe that BO tends to perform well on small datasets, e.g., 'credit' in Figure 3(a), where the 1h budget is sufficient. Under the same budget, it may perform badly on large datasets, e.g., 'volkert' in Figure 3(c), as it may try configurations which consume a very large portion of the budget at the early stage of the search. On the medium size dataset, e.g., 'KDDCup09' in Figure 3(b), the local search method is more efficient than BO in the early stage but is outperformed in the later stage. `BlendSearch` performs similarly to the better one between LS and BO in the early stage, and surpasses both of them in the later stage. We also observe that BOwLS performs similarly with LS (sometimes worse). This is because BOwLS needs to wait until a local search converges before proposing a new one. The aggregated result over all the test cases in Figure 4(a) & (b) is consistent with these observations.

The interplay between local and global search in `BlendSearch`. We investigated the dynamics of local and global search thread selection in `BlendSearch`. As a case study, we show the results in tuning XGBoost on two datasets in Figure 1(b) and Figure 5(a). The result in Figure 1(b) shows

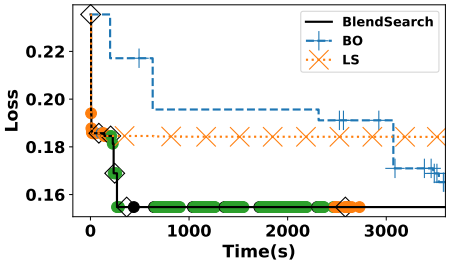


Figure 5: Search thread selection in `BlendSearch` in tuning XGBoost on dataset *KDDCup09*. In this figure (and Figure 1(b), which shows the result for dataset *Volkert*), each marker corresponds to one time of search thread selection in `BlendSearch`. The diamonds correspond to the global search thread, and the solid circles correspond to the local search threads (different colors for different threads).

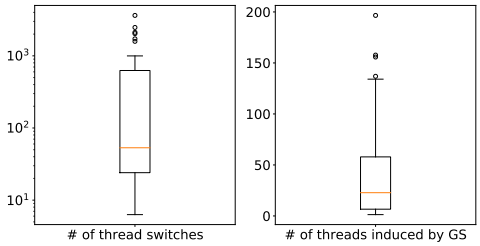


Figure 6: The distributions of the # of thread switches and the # of local search threads induced by the GS (short for global search) in `BlendSearch` for tuning XGBoost. The result is aggregated over all the datasets evaluated in tunning XGBoot. The outliers with very large # of thread switches and threads induced by GS are from very small datasets, where the number of evaluations is large.

that the global search indeed plays a role after 1000s which contributes to `BlendSearch`'s better performance comparing to LS. The result in Figure 5(a) shows that although the first point suggested by the global search (at around 200s) does not yield significantly better performance immediately, the induced new local search thread (the thread in green circle) lead to a significantly better performance soon. These two figures together indicate that global search is not only responsible for directly finding better configurations, but also for creating local search threads that can achieve yet better performance. We show two statistics about the overall interplay between global and local search in `BlendSearch` on all the datasets evaluated for tuning XGBoost in Figure 6.

We provide additional experiments for tuning XGBoost and LightGBM in Appendix B, including a comparison with ASHA under different settings of fidelity, and an empirical study about the effect of low-cost initialization.

Takeaway. (1) `BlendSearch` is able to overcome the limitations of BO and LS and at the same time, inherit their advantages. By blending BO and LS, `BlendSearch` is able to outperform both on this large collection of datasets over time. (2) The interplay between global and local search indeed contribute to `BlendSearch`'s good performance.

4.2 Transformer-based NLP model fine-tuning

This section presents an application of economical hyperparameter optimization to an NLP model fine tuning task used in a large software company. It starts from a large transformer-based model Turing-NLRv2 with 24 transformer layers, each with 16 attention heads and hidden-dimension 1024 totaling 340M parameters. It is pretrained on English Wikipedia (2,500M words) and BookCorpus (800M words), and uses byte-pair encoding[3]. This pre-trained model is then fine-tuned (Dai & Le, 2015) for use in multiple production scenarios, including sequence classification, named entity recognition and question answering. The fine-tuning procedure is performed for a dozen separate tasks, and is repeated on a regular cadence, typically every few weeks. We focus our experiment on a single sequence classification fine-tuning task where the objective is to label a document (consisting of one or more sentences) with one of five possible classes. For fine-tuning this model we introduce a classification layer with $1024 \times 5 = 5120$ additional weight parameters, randomly initialized. The dataset used for training consists of 52K labeled examples, which we split 80/20 for training/validation. The objective to maximize is the f1-score obtained on the validation set of 10.4K labeled documents. Selecting hyperparameters for fine-tuning this model has been a manual process that typically takes a data scientist a few days.

[3] `msturing.org`

Table 1: Results of language model fine-tuning.

method	wall-clock	GPU-hours	f1
Manual	(10-20 working hours, spanned to a few days)	40-80h	0.645
`BlendSearch`	**3h**	**12h**	**0.652**
ASHA-1	6h	24h	0.624
ASHA-16	6h	384h	0.640

In our experiment we use a 6-dimensional search space (4 numerical and 2 categorical). A detailed description of all the hyperparameters tuned, including their ranges, can be found in Table 5 in Appendix B. We compare to ASHA and let it use 16 VMs with 4 NVIDIA Tesla V100 GPUs on each VM. We run ASHA with 16 concurrent jobs for 6 hours in wallclock time. That amounts to $4 \times 16 \times 6 = 384$ GPU hours of hardware cost. We run `BlendSearch` on a *single* VM of the same configuration for 3 hours, which uses 12 GPU hours in total. For comparison, we include ASHA-1 using the same single VM. The results are summarized in Table 1.

Takeaway. Not only `BlendSearch` finds a more accurate model than both ASHA and manual effort, it does so faster and consumes only 3% of the resources of ASHA-16.

4.3 Ablation Study

To investigate the effectiveness of several important modules of our framework (colored in orange in Figure 2), we perform an ablation study in tuning XGBoost on a random subset of the datasets (one third of the datasets mentioned in Section 4.1). We show the aggregated rank of different variants of our method in Figure 4(c). Specifically, we study the following three modules of the framework. **(1) Priority metric.** 'BS-Priority:RoundRobin' uses the round-robin policy in the search thread selector. **(2) Config validator.** 'BS-w/o-ConfigValidator' skips the validity check of the proposed configuration. **(3) Create new thread condition.** In 'BS-CreateCond:Always' and 'BS-CreatCond:BestOrFirst', the following two conditions are used as the condition for creating new local search threads respectively: always create, and create a new thread only when the loss is better than all the existing search threads' loss or there is no local search thread yet.

From this ablation study, we have the following observations: (1) Making round-robin selection is worse than doing selection using our designed priority metric. Round-robin's relative performance becomes worse in the later stage of the search because it cannot avoid bad-performing search threads. (2) The config validator is also vital in our framework. (3) Overall, the conditions for creating new threads has a smaller impact on our method comparing to the other designs studied. 'Better than half' condition used by default tends to perform the best.

5 Extension and Future work

In the low-resource scenario which is targeted by this paper, each single trial is not resource-saturated if we spend all resources in it. So we do not recommend parallel trials in this low-resource scenario. In the case where more resources than the maximum resource each single trial can consume are available, our framework can be extended by running the trials from different search threads on multiple workers. For example, if there are additional workers available, we can keep invoking the search thread selector (but skip the local search threads that have $O(d)$ trials running). Our design of having multiple independent local search threads naturally allows efficient asynchronous parallel trials. The design of utilizing existing global optimization methods allows existing easy-to-parallelize global optimization (such as random search or batch versions of BO) to be plugged in. The prioritization of search threads is still useful as long as the maximal concurrent number of trials divided by the number of search threads is smaller than $O(d)$. Since our method can be used together with multi-fidelity pruning methods, it can naturally inherit the asynchronous resource scheduling when used in the parallel setting. Parallelization is now supported in the latest version of `BlendSearch`'s implementation.

In this work, we show the effectiveness of `BlendSearch` through extensive empirical study. As future work, it is worth studying the theoretical properties of `BlendSearch`, including theoretical guarantees about its convergence rate and total resource consumption.

REFERENCES

Takuya Akiba, Shotaro Sano, Toshihiko Yanase, Takeru Ohta, and Masanori Koyama. Optuna: A next-generation hyperparameter optimization framework. In *Proceedings of the 25rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2019.

James S Bergstra, Rémi Bardenet, Yoshua Bengio, and Balázs Kégl. Algorithms for hyper-parameter optimization. In *Advances in neural information processing systems*, 2011.

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.

Adam D. Bull. Convergence rates of efficient global optimization algorithms. *J. Mach. Learn. Res.*, 12:2879–2904, November 2011.

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

Kevin Clark, Minh-Thang Luong, Quoc V. Le, and Christopher D. Manning. ELECTRA: Pre-training text encoders as discriminators rather than generators. In *ICLR*, 2020.

Andrew M Dai and Quoc V Le. Semi-supervised sequence learning. *Advances in neural information processing systems*, 2015.

David Eriksson, Michael Pearce, Jacob Gardner, Ryan D Turner, and Matthias Poloczek. Scalable global optimization via local bayesian optimization. In *Advances in Neural Information Processing Systems*, 2019.

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.

Yuzhou Gao, Tengchao Yu, and Jinglai Li. Bayesian optimization with local search. In *International Conference on Machine Learning, Optimization, and Data Science*, 2020.

Pieter Gijsbers, Erin LeDell, Janek Thomas, Sébastien Poirier, Bernd Bischl, and Joaquin Vanschoren. An open source automl benchmark. In *AutoML Workshop at ICML 2019*, 2019.

András György and Levente Kocsis. Efficient multi-start strategies for local search algorithms. *Journal of Artificial Intelligence Research*, 41:407–444, 2011.

Pengcheng He, Xiaodong Liu, Jianfeng Gao, and Weizhu Chen. Deberta: Decoding-enhanced bert with disentangled attention. *arXiv preprint arXiv:2006.03654*, 2020.

Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown. Sequential model-based optimization for general algorithm configuration. In *Learning and Intelligent Optimization*, 2011.

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

Kirthevasan Kandasamy, Gautam Dasarathy, Jeff Schneider, and Barnabás Póczos. Multi-fidelity bayesian optimisation with continuous approximations. In *Proceedings of the 34th International Conference on Machine Learning*, 2017.

Guolin Ke, Qi Meng, Thomas Finley, Taifeng Wang, Wei Chen, Weidong Ma, Qiwei Ye, and Tie-Yan Liu. Lightgbm: A highly efficient gradient boosting decision tree. In *Advances in Neural Information Processing Systems*, 2017.

Aaron Klein, Stefan Falkner, Simon Bartels, Philipp Hennig, and Frank Hutter. Fast bayesian optimization of machine learning hyperparameters on large datasets. In *Artificial Intelligence and Statistics*, 2017.

Tor Lattimore and Csaba Szepesvári. *Bandit algorithms*. Cambridge University Press, 2020.

Liam Li, Kevin Jamieson, Afshin Rostamizadeh, Ekaterina Gonina, Jonathan Ben-tzur, Moritz Hardt, Benjamin Recht, and Ameet Talwalkar. A system for massively parallel hyperparameter tuning. In *Proceedings of Machine Learning and Systems*, 2020.

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

Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. Roberta: A robustly optimized bert pretraining approach. *arXiv preprint arXiv:1907.11692*, 2019.

Zhiyun Lu, Liyu Chen, Chao-Kai Chiang, and Fei Sha. Hyper-parameter tuning under a budget constraint. In *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence, IJCAI-19*, 2019.

Yurii Nesterov and Vladimir Spokoiny. Random gradient-free minimization of convex functions. *Foundations of Computational Mathematics*, 17(2):527–566, 2017.

Jasper Snoek, Hugo Larochelle, and Ryan P Adams. Practical bayesian optimization of machine learning algorithms. In *Advances in neural information processing systems*, 2012.

James C Spall et al. Multivariate stochastic approximation using a simultaneous perturbation gradient approximation. *IEEE transactions on automatic control*, 37(3):332–341, 1992.

Niranjan Srinivas, Andreas Krause, Sham M Kakade, and Matthias Seeger. Gaussian process optimization in the bandit setting: No regret and experimental design. *arXiv preprint arXiv:0912.3995*, 2009.

Qingyun Wu, Chi Wang, and Silu Huang. Frugal optimization for cost-related hyperparameters. In *AAAI'21*, 2021.

Li Yang and Abdallah Shami. On hyperparameter optimization of machine learning algorithms: Theory and practice. *Neurocomputing*, 415:295 – 316, 2020.