Python Wrapper for Simulating Multi-Fidelity Optimization on HPO Benchmarks without Any Wait

Shuhei Watanabe

Department of Computer Science, University of Freiburg, Germany
watanabs@cs.uni-freiburg.de

Abstract Hyperparameter (HP) optimization of deep learning (DL) is essential for high performance. As DL often requires several hours to days for its training, HP optimization (HPO) of DL is often prohibitively expensive. This boosted the emergence of tabular or surrogate benchmarks, which enable querying the (predictive) performance of DL with a specific HP configuration in a fraction. However, since the actual runtime of a DL training is significantly different from its query response time, simulators of an asynchronous HPO, e.g. multi-fidelity optimization, must wait for the actual runtime at each iteration in a naïve implementation; otherwise, the evaluation order during simulation does not match with the real experiment. To ease this issue, we developed a Python wrapper and describe its usage. This wrapper forces each worker to wait so that we yield exactly the same evaluation order as in the real experiment with only 10^{-2} seconds of waiting instead of waiting several hours. Our implementation is available at https://github.com/nabenabe0928/mfhpo-simulator/.

1 Introduction

Hyperparameter (HP) optimization of deep learning (DL) is crucial for strong performance (Chen et al. (2018); Henderson et al. (2018)) and it surged the research on HP optimization (HPO) of DL. However, due to the heavy computational nature of DL, HPO is often prohibitively expensive and both energy and time costs are not negligible. This is the driving force behind the emergence of tabular and surrogate benchmarks, which enable yielding the (predictive) performance of a specific HP configuration in less than a second (Eggensperger et al. (2015, 2021); Arango et al. (2021); Bansal et al. (2022)).

Although these benchmarks effectively reduce the energy usage and the runtime of experiments in many cases, experiments considering runtimes between parallel workers may not be easily benefited. For example, multi-fidelity optimization (MFO) (Kandasamy et al. (2017)) has been actively studied recently due to its computational efficiency (Jamieson and Talwalkar (2016); Li et al. (2017); Falkner et al. (2018); Awad et al. (2021)), but because of its asynchronous nature, the call order of each worker must be appropriately sorted out to not break the states that the actual experiments would go through. While this problem is naïvely addressed by making each worker wait for the runtime the actual DL training would take, each worker must wait for a substantial amount of time in this case, and hence it ends up wasting energy and time.

To address this problem, we developed a Python wrapper that automatically sorts out the right release order of evaluations for each worker in a fraction and forces each worker to wait internally in order to match the apparent evaluation order from the optimizer perspective. In this paper, we describe how our package solves the problem and how to use this wrapper. Our package can be easily used for existing parallel optimizers such as BOHB (Falkner et al. (2018)), DEHB (Awad et al. (2021)), and SMAC3 (Lindauer et al. (2022)) and we provide the example codes of how to pass our wrapper to these existing optimizers at https://github.com/nabenabe0928/mfhpo-simulator/.

2 Background

In this section, we describe the potential problems during MFO on tabular or surrogate benchmarks and explain why our wrapper would be useful. Throughout the paper, we assume minimization problems of an objective function $f(\boldsymbol{x}) : \mathcal{X} \to \mathbb{R}$ defined on the search space $\mathcal{X} \coloneqq \mathcal{X}_1 \times \mathcal{X}_2 \times \cdots \times \mathcal{X}_D$ where $\mathcal{X}_d \subseteq \mathbb{R}$ is the domain of the d-th HP. Furthermore, we define the (predictive) *actual* runtime function $\tau(\boldsymbol{x}) : \mathcal{X} \to \mathbb{R}_+$ of the objective function given an HP configuration $\boldsymbol{x}$. Note that although $f(\boldsymbol{x})$ and $\tau(\boldsymbol{x})$ could involve randomness, we only describe the deterministic version for the notational simplicity, and we use the notation $[i] \coloneqq \{1, 2, \ldots, i\}$ where i is a positive integer throughout this paper.

2.1 Asynchronous Optimization on Surrogate or Tabular Benchmarks

Assume we have the oracle, i.e. tabular or surrogate benchmark, of f and τ that could be queried in a negligible amount of time, the HP configuration $\boldsymbol{x}$ is sampled from a policy $\pi(\boldsymbol{x}|\mathcal{D}^{(N)})$ where $\mathcal{D}^{(N)} \coloneqq \{(\boldsymbol{x}_n, f(\boldsymbol{x}_n))\}_{n=1}^N$ is a set of observations, and we have a set of parallel workers $\{W_p\}_{p=1}^P$ where each worker $W_p : \mathcal{X} \to \mathbb{R}^2$ is a wrapper of $f(\boldsymbol{x})$ and $\tau(\boldsymbol{x})$. Let a mapping $i_n : \mathbb{Z}_+ \to [P]$ be an index specifier of which worker processed the n-th observation and $\mathcal{I}_p^{(N)} \coloneqq \{n \in [N] \mid i_n = p\}$ be a set of indices of observations the p-th worker processed. Then the (simulated) runtime of the p-th worker is computed as follows if we ignore the sampling time:

$$T_p^{(N)} \coloneqq \sum_{n \in \mathcal{I}_p^{(N)}} \tau(\boldsymbol{x}_n). \tag{1}$$

In turn, the $(N+1)$-th observation will be processed by the worker that will be free for the first time, and thus the index of the worker for the $(N+1)$-th observation is specified by $\operatorname{argmin}_{p \in [P]} T_p^{(N)}$.

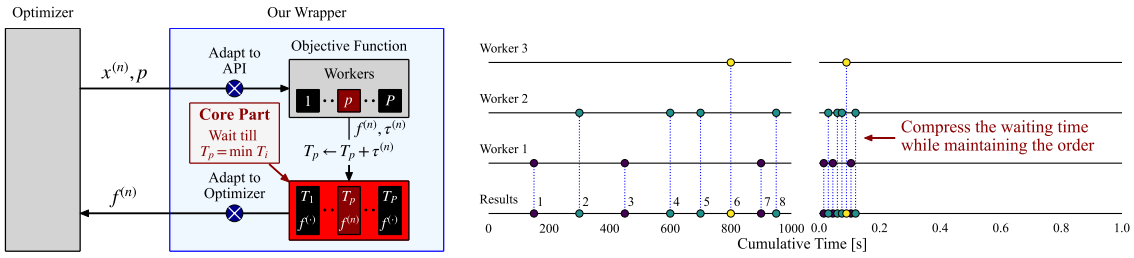
The problems of this setting are that (1) the policy is conditioned on $\mathcal{D}^{(N)}$ and the order of the observations must be preserved and (2) each worker must wait for the other workers to match the order to be realistic. While an obvious approach is to let each worker wait for the queried runtime $\tau(\boldsymbol{x})$ as seen in Figure 1b, it is a waste of energy and time. Therefore, we developed a Python wrapper to match the order while each worker waits for only a *negligible amount of time*.

2.2 Related Work

Although there have been many HPO benchmarks invented for MFO such as HPOBench (Eggensperger et al. (2021)), NASLib (Mehta et al. (2022)), and JAHS-Bench-201 (Bansal et al. (2022)), none of them provides a module to allow researchers to simulate runtime internally. Other than HPO benchmarks, many HPO frameworks handling MFO have also been developed so far such as Optuna (Akiba et al. (2019)), SMAC3 (Lindauer et al. (2022)), Dragonfly (Kandasamy et al. (2020)), and RayTune (Liaw et al. (2018)). However, no framework above considers the simulation of runtime. Although HyperTune (Li et al. (2022)) and SyneTune (Salinas et al. (2022)) are internally simulating the runtime, we cannot simulate optimizers of interest if the optimizers are not introduced in the packages. It implies that researchers cannot immediately simulate their own new methods unless they incorporate their methods in these packages. Furthermore, their simulation backend does not support multiprocessing and multithreading. Therefore, an easy-to-use Python wrapper for the simulation is needed.

3 Automatic Waiting Time Scheduling Wrapper

As an objective function may take a random seed and fidelity parameters in practice, we denote a set of the arguments for the n-th query as $\boldsymbol{a}^{(n)}$. Additionally, *job* means to allocate the n-th queried HP configuration $\boldsymbol{x}^{(n)}$ to a free worker and obtain its result $r^{(n)} \coloneqq (f(\boldsymbol{x}^{(n)}|\boldsymbol{a}^{(n)}), \tau(\boldsymbol{x}^{(n)}|\boldsymbol{a}^{(n)}))$. Besides that, we denote the n-th chronologically ordered result as r_n. Our wrapper is required to satisfy the following features:



(a) Wrapper Workflow

(b) Compression of Simulated Runtime

Figure 1: The conceptual visualizations of our wrapper. (a) The workflow of our wrapper. The gray parts are provided by users and our package is responsible for the light blue part. The blue circles with the white cross must be modified by users via inheritance to match the signature used in our wrapper as described in Listings 1 and 2. The p-th worker receives the n-th queried configuration $\boldsymbol{x}^{(n)}$ and stores its result $f^{(n)} := f(\boldsymbol{x}^{(n)}), \tau^{(n)} := \tau(\boldsymbol{x}^{(n)})$ in the file system. Our wrapper sorts out the right timing to return the n-th queried result $f^{(n)}$ to the optimizer based on the simulated runtime T_p. (b) The compression of simulated runtime. Each circle on each line represents the timing when each result was delivered from each worker. **Left**: an example when we naïvely wait for the (actual) runtime $\tau(\boldsymbol{x})$ of each query. Although we receive each result at the right timing, the simulation incurs a substantial amount of waiting time. **Right**: an example when we use our wrapper. We can significantly reduce the experiment time without losing the exact order.

- The i-th result r_i comes earlier than the j-th result r_j for all $i < j$,
- The wrapper recognizes each worker and allocates a job to the exact worker even when using multiprocessing (e.g. `joblib` and `dask`) and multithreading (e.g. `concurrent.futures`),
- The evaluation of each job can be restarted in MFO.

Note that an example of the restart of evaluation means that when we already evaluate DL with $\boldsymbol{x}$ for 20 epochs and would like to evaluate DL with the same HP configuration $\boldsymbol{x}$ for 100 epochs, we start the training of DL from the 21st epoch instead of from scratch using the intermediate state. To achieve these features, we chose to share the required information via the file system and create the following JSON files that map:

- from a thread or process ID of each worker to a worker index $p \in [P]$,
- from a worker index $p \in [P]$ to its (simulated) cumulative runtime $T_p^{(N)}$, and
- from the n-th configuration $\boldsymbol{x}_n$ to a list of intermediate states $s_n := (\tau(\boldsymbol{x}_n), T_{i_n}^{(n)}, \boldsymbol{a}_n)$.

Note that multiple intermediate states could exist for an HP configuration $\boldsymbol{x}_n$. Algorithm 1 presents the pseudocode. Although the algorithm itself is nothing special, we need to make sure that multiple workers will not edit the same file at the same time. Since usecases of our wrapper are not really limited to multiprocessing or multithreading that spawns child workers but could be file-based synchronization, we use `fcntl` to safely acquire file locks. Furthermore, the test coverage of our code on MacOS and Ubuntu is $95^{+}\%$ to validate the behavioral correctness of our package.

4 Tool Usage

In our package, `ObjectiveFuncWrapper` is the main module and it provides three different options: (1) function wrapper for multiprocessing or multithreading that spawns child workers (e.g. DEHB and SMAC3), (2) function wrapper for file-based or server-based synchronization (e.g. NePS [1]), or

[1] `https://github.com/automl/neps/`

Algorithm 1 Automatic Waiting Time Scheduling Wrapper (see Figure 1a as well)

function WORKER($\boldsymbol{x}^{(N)}, \boldsymbol{a}^{(N)}$)
  Get intermediate states from the file if intermediate states of $\boldsymbol{x}^{(N)}$ exist
  if evaluation can be restarted from the n-th observation **then**
    $\tau' \leftarrow \tau(\boldsymbol{x}_n | \boldsymbol{a}_n)$ ▷ Note that $s_n := (\tau_n, T^{(n)}_{i_n}, \boldsymbol{a}_n)$ s.t. $T^{(n)}_{i_n} \leq T^{(N)}_p + t^{(N)}$ cannot be used
  else ▷ Restart is not supported when there are multiple fidelities
    $\tau' \leftarrow 0$
  Query the result $r^{(N)} := (f^{(N)}, \tau^{(N)}) := (f(\boldsymbol{x}^{(N)} | \boldsymbol{a}^{(N)}), \tau(\boldsymbol{x}^{(N)} | \boldsymbol{a}^{(N)}))$
  $T_{\text{now}} \leftarrow \max(T_{\text{now}}, T^{(N)}_p) + t^{(N)}, T^{(N+1)}_p \leftarrow T_{\text{now}} + \tau^{(N)} - \tau', T^{(N+1)}_{p'} \leftarrow T^{(N)}_{p'}$ $(p' \neq p)$
  ▷ $k \in \mathbb{Z}_{\geq 0}$ is the number of results from the other workers appended during the wait
  Wait till $p \in \text{argmin}_{p \in [P]} T^{(N+k+1)}_p$ satisfies
  Record (or update the state s_n to) $s_{N+k+1} = (\tau^{(N)}, T^{(N+k+1)}_p, \boldsymbol{a}^{(N)})$ with a key $\boldsymbol{x}^{(N)}$
  return $f_{N+k+1} := f^{(N)}$
$\mathcal{D}_0 \leftarrow \emptyset, T^{(0)}_p \leftarrow 0, T_{\text{now}} \leftarrow 0, N \leftarrow 0$, and user needs to provide an optimizer policy π
while the budget is left **do** ▷ This codeblock is run by P different workers in parallel
  ▷ e.g. BOHB and DEHB define a fidelity (in $\boldsymbol{a}^{(N)}$) at each iteration by themselves
  Sample $\boldsymbol{x}^{(N)} \sim \pi(\cdot | \mathcal{D}_N)$ with $t^{(N)}$ seconds and get $\boldsymbol{a}^{(N)}$ from a user-defined algorithm
  $f^{(N)} \leftarrow \texttt{worker}(\boldsymbol{x}^{(N)}, \boldsymbol{a}^{(N)})$
  ▷ Recall that $\mathcal{D}_\cdot$ is modified from the other workers as well
  $\mathcal{D}_{N+k+1} \leftarrow \mathcal{D}_{N+k} \cup \{(\boldsymbol{x}^{(N)}, f^{(N)})\}, N \leftarrow N + k + 1$

MPI, and (3) function and optimizer wrapper for the ask-and-tell interface. In this paper, we discuss the usage of Options 1 and 2, and we kindly ask users to refer to Appendix A.2 for Option 3.

An instance of `ObjectiveFuncWrapper` serves as an objective function and we just need to pass it to an optimizer; see Appendix A.1 for the arguments of `ObjectiveFuncWrapper`. When optimizers take a different interface, we can easily modify the interface via inheritance:

Listing 1: An example of inheritance for a different interface.

```
class MyObjectiveFuncWrapper(ObjectiveFuncWrapper):
    def __call__(self, config, budget):
        # modify config into dict[str, Any]
        return super().__call__(
            eval_config=config,
            fidels={self.fidel_keys[0]: budget}
        )
```

In Listing 1, the optimizer requires the objective function to have arguments named `config` instead of `eval_config` and `budget` instead of `fidels`. Then we need to somehow modify `config` into the format of `eval_config` (`dict[str, Any]`) if `config` is not `dict[str, Any]`. We can also easily deactivate the MFO setting if `fidel_keys=None` is specified:

Listing 2: An example of inheritance for non MFO setting.

```
class MyObjectiveFuncWrapper(ObjectiveFuncWrapper):
    def __call__(self, eval_config):
        # fidel_keys must be None; otherwise, get an error
        return super().__call__(eval_config=eval_config)
```

Further examples for BOHB, DEHB, NePS, and SMAC3 on MLP in Table 6 of

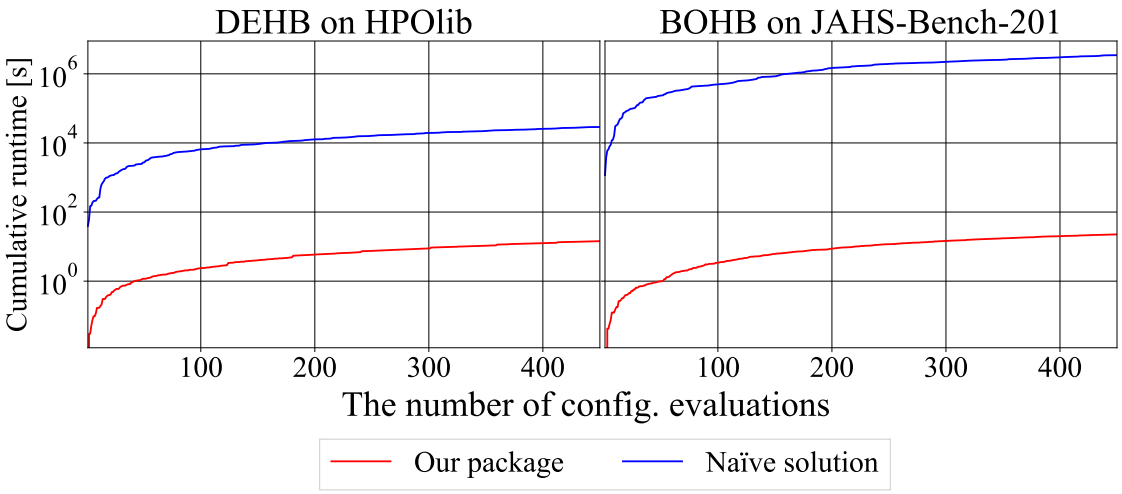


Figure 2: The benchmarks of runtime using our package (the red lines) and the naïve simulation (the blue lines), which waits for $\tau(\boldsymbol{x})$ at each iteration. Our benchmarks were performed on a machine with 12 cores of Intel core i7-1255U. **Left**: the benchmark using DEHB (Awad et al. (2021)) on Slice Localization of HPOlib (Klein and Hutter (2019)). **Right**: the benchmark using BOHB (Falkner et al. (2018)) on CIFAR10 of JAHS-Bench-201 (Bansal et al. (2022)). While BOHB took 23 seconds and DEHB took 15 seconds to finish each benchmark with our package, their naïve solutions took hours to days to finish benchmarks.

HPOBench (Eggensperger et al. (2021)), HPOlib (Klein and Hutter (2019)), JAHS-Bench-201 (Bansal et al. (2022)), LCBench (Zimmer et al. (2021)) in YAHPOBench (Pfisterer et al. (2022)), some synthetic functions. See Appendices B,C for more details. The examples are available at `https://github.com/nabenabe0928/mfhpo-simulator/`.

5 Broader Impact & Limitations

The primary motivation for this paper is to reduce the runtime of simulations for MFO. In Figure 2, we show an example of MFO using DEHB on Slice Localization of HPOlib (Klein and Hutter (2019)) and BOHB on CIFAR10 of JAHS-Bench-201 (Bansal et al. (2022)). Both examples use $\eta = 3$, which is a control parameter of HyperBand (Li et al. (2017)) and $P = 4$. As can be seen, while we could finish 450 evaluations in $15 \sim 25$ seconds with our wrapper, the naïve implementation required hours to days. As a computer consumes about 30Wh even just for waiting and 736 g of CO_2 is produced per 1kWh, the experiment of BOHB on JAHS-Bench-201 would have produced about 6 kg of CO_2. Therefore, researchers can reduce the corresponding amount of CO_2 for each experiment. The main limitation of our current wrapper is the assumption that none of the workers will not die and any additional workers will not be added after the initialization. Besides that, our package cannot be used on Windows OS because `fcntl` is not supported on Windows.

6 Conclusions

In this paper, we presented a Python wrapper to maintain the exact order of the observations without waiting for actual runtimes. Although some existing packages internally support the similar mechanism, they are not applicable to multiprocessing or multithread setups and they cannot be immediately used for newly developed methods. On the other hand, our package supports such distributed computing setups and researchers can simply wrap their objective functions by our wrapper and they can directly give to their optimizers. We describe the basic usage of our package and demonstrated that our package significantly reduce the CO_2 production that experiments using tabular and surrogate benchmarks would have caused.

References

Akiba, T., Sano, S., Yanase, T., Ohta, T., and Koyama, M. (2019). Optuna: A next-generation hyperparameter optimization framework. In *International Conference on Knowledge Discovery & Data Mining*.

Arango, S., Jomaa, H., Wistuba, M., and Grabocka, J. (2021). HPO-B: A large-scale reproducible benchmark for black-box HPO based on OpenML. *arXiv:2106.06257*.

Awad, N., Mallik, N., and Hutter, F. (2021). DEHB: Evolutionary HyperBand for scalable, robust and efficient hyperparameter optimization. *arXiv:2105.09821*.

Bansal, A., Stoll, D., Janowski, M., Zela, A., and Hutter, F. (2022). JAHS-Bench-201: A foundation for research on joint architecture and hyperparameter search. In *Advances in Neural Information Processing Systems Datasets and Benchmarks Track*.

Bergstra, J., Bardenet, R., Bengio, Y., and Kégl, B. (2011). Algorithms for hyper-parameter optimization. *Advances in Neural Information Processing Systems*.

Chen, Y., Huang, A., Wang, Z., Antonoglou, I., Schrittwieser, J., Silver, D., and de Freitas, N. (2018). Bayesian optimization in AlphaGo. *arXiv:1812.06855*.

Dong, X. and Yang, Y. (2020). NAS-Bench-201: Extending the scope of reproducible neural architecture search. *arXiv:2001.00326*.

Eggensperger, K., Hutter, F., Hoos, H., and Leyton-Brown, K. (2015). Efficient benchmarking of hyperparameter optimizers via surrogates. In *AAAI Conference on Artificial Intelligence*.

Eggensperger, K., Müller, P., Mallik, N., Feurer, M., Sass, R., Klein, A., Awad, N., Lindauer, M., and Hutter, F. (2021). HPOBench: A collection of reproducible multi-fidelity benchmark problems for HPO. *arXiv:2109.06716*.

Falkner, S., Klein, A., and Hutter, F. (2018). BOHB: Robust and efficient hyperparameter optimization at scale. In *International Conference on Machine Learning*.

Henderson, P., Islam, R., Bachman, P., Pineau, J., Precup, D., and Meger, D. (2018). Deep reinforcement learning that matters. In *AAAI Conference on Artificial Intelligence*.

Jamieson, K. and Talwalkar, A. (2016). Non-stochastic best arm identification and hyperparameter optimization. In *International Conference on Artificial Intelligence and Statistics*.

Kandasamy, K., Dasarathy, G., Schneider, J., and Póczos, B. (2017). Multi-fidelity Bayesian optimisation with continuous approximations. In *International Conference on Machine Learning*.

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

Klein, A. and Hutter, F. (2019). Tabular benchmarks for joint architecture and hyperparameter optimization. *arXiv:1905.04970*.

Li, L., Jamieson, K., DeSalvo, G., Rostamizadeh, A., and Talwalkar, A. (2017). HyperBand: A novel bandit-based approach to hyperparameter optimization. *Journal of Machine Learning Research*, 18.

Li, Y., Shen, Y., Jiang, H., Zhang, W., Li, J., Liu, J., Zhang, C., and Cui, B. (2022). Hyper-Tune: towards efficient hyper-parameter tuning at scale. *arXiv:2201.06834*.

Liaw, R., Liang, E., Nishihara, R., Moritz, P., Gonzalez, J., and Stoica, I. (2018). Tune: A research platform for distributed model selection and training. *arXiv:1807.05118*.

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

Mehta, Y., White, C., Zela, A., Krishnakumar, A., Zabergja, G., Moradian, S., Safari, M., Yu, K., and Hutter, F. (2022). NAS-Bench-Suite: NAS evaluation is (now) surprisingly easy. *arXiv:2201.13396*.

Müller, S. and Hutter, F. (2021). TrivialAugment: Tuning-free yet state-of-the-art data augmentation. In *International Conference on Computer Vision*.

Pfisterer, F., Schneider, L., Moosbauer, J., Binder, M., and Bischl, B. (2022). YAHPO Gym – an efficient multi-objective multi-fidelity benchmark for hyperparameter optimization. In *International Conference on Automated Machine Learning*.

Salinas, D., Seeger, M., Klein, A., Perrone, V., Wistuba, M., and Archambeau, C. (2022). Syne Tune: A library for large scale hyperparameter tuning and reproducible research. In *International Conference on Automated Machine Learning*.

Watanabe, S. (2023). Tree-structured Parzen estimator: Understanding its algorithm components and their roles for better empirical performance. *arXiv:2304.11127*.

Zimmer, L., Lindauer, M., and Hutter, F. (2021). Auto-PyTorch: Multi-fidelity metalearning for efficient and robust AutoDL. *Transactions on Pattern Analysis and Machine Intelligence*.

A Wrapper Object (`ObjectiveFuncWrapper`)

In this section, we describe more details on our wrapper.

A.1 Arguments

The arguments of `ObjectiveFuncWrapper` object are as follows:

- `obj_func`: the objective function that takes (`eval_config`, `fidels`, `seed`, `**data_to_scatter`) as arguments and returns $f(\boldsymbol{x}|\boldsymbol{a})$ and $\tau(\boldsymbol{x}|\boldsymbol{a})$ where `eval_config` is `dict[str, Any]`,
- `launch_multiple_wrappers_from_user_side (bool)`: whether to instantiate multiple wrappers from userside (e.g. NePS that uses file-based synchronization) or not (e.g. DEHB that uses multiprocessing that spawns subprocesses),
- `ask_and_tell (bool)`: whether to use simulator for ask-and-tell interface (`True`) or not,
- `save_dir_name (str | None)`: the results and the required information will be stored in `mfhpo-simulator-info/<save_dir_name>/`,
- `n_workers (int)`: the number of parallel workers P,
- `n_evals (int)`: the number of evaluations to get,
- `n_actual_evals_in_opt (int)`: the number of HP configurations to be evaluated in an optimizer which is used only for checking if no hang happens (should take `n_evals + n_workers`),
- `continual_max_fidel (int | None)`: when users would like to restart each evaluation from intermediate states, the maximum fidelity value for the target fidelity must be provided. Note that the restart is allowed only if there is only one fidelity parameter. If `None`, no restart happens as in Line 6 of Algorithm 1,
- `runtime_key (str)`: the key of the runtime in the returned value of `obj_func`,
- `obj_keys (list[str])`: the keys of the objective and constraint names in the returned value of `obj_func` and the values of the specified keys will be stored in the result file,
- `fidel_keys (list[str] | None)`: the keys of the fidelity parameters in input `fidels`,
- `seed (int | None)`: the random seed to be used in the wrapper,
- `max_waiting_time (float)`: the maximum waiting time for each worker and if each worker did not get any update for this amount of time, it will return `inf`,
- `store_config (bool)`: whether to store configurations, fidelities, and seed used for each evaluation, and
- `check_interval_time (float)`: how often each worker should check whether a new job can be assigned to it.

Note that `data_to_scatter` is especially important when an optimizer uses multiprocessing packages such as `dask` or `multiprocessing`, which deserialize `obj_func` every time we call. By passing large-size data via `data_to_scatter`, the time for (de)serialization will be negligible if optimizers use `dask.scatter` or something similar internally. We kindly ask readers to check any updates to the arguments at `https://github.com/nabenabe0928/mfhpo-simulator/`.

A.2 Wrapper for Ask-and-Tell Interface

When optimizers take ask-and-tell interface, simulations can be run on a single worker while preserving the results, and hence simulations can be further accelerated. Note that the bottleneck of simulations is the waiting time due to the communication among each worker and simulations on a single worker can address this problem. The benefits of this option are (1) faster, (2) memory-efficient, and (3) stable. On the other hand, the downsides are that (1) this option forces optimizers to have the ask-and-tell interface and (2) simulations may fail to precisely consider a bottleneck caused by parallel execution of expensive optimizers. For more details, see `https://github.com/nabenabe0928/mfhpo-simulator/`.

B Benchmarks

We first note that since the Branin and the Hartmann functions must be minimized, our functions have different signs from the prior literature that aims to maximize objective functions and when $\boldsymbol{z} = [z_1, z_2, \ldots, z_K] \in \mathbb{R}^K$, our examples take $\boldsymbol{z} = [z, z, \ldots, z] \in \mathbb{R}^K$. However, if users wish, users can specify $\boldsymbol{z}$ as $\boldsymbol{z} = [z_1, z_2, \ldots, z_K]$ from `fidel_dim`.

B.1 Branin Function

The Branin function is the following $2D$ function that has 3 global minimizers and no local minimizer:

$$f(x_1, x_2) = a(x_2 - bx_1^2 + cx_1 - r)^2 + s(1-t)\cos x_1 + s \tag{2}$$

where $\boldsymbol{x} \in [-5, 10] \times [0, 15]$, $a = 1$, $b = 5.1/(4\pi^2)$, $c = 5/\pi$, $r = 6$, $s = 10$, and $t = 1/(8\pi)$. The multi-fidelity Branin function was invented by Kandasamy et al. (2020) and it replaces b, c, t with the following b_z, c_z, t_z:

$$\begin{aligned} b_z &\coloneqq b - \delta_b(1 - z_1), \\ c_z &\coloneqq c - \delta_c(1 - z_2), \text{ and} \\ t_z &\coloneqq t + \delta_t(1 - z_3), \end{aligned} \tag{3}$$

where $\boldsymbol{z} \in [0, 1]^3$, $\delta_b = 10^{-2}$, $\delta_c = 10^{-1}$, and $\delta_t = 5 \times 10^{-3}$. $\delta_\cdot$ controls the rank correlation between low- and high-fidelities and higher $\delta_\cdot$ yields less correlation. The runtime function for the multi-fidelity Branin function is computed as [2]:

$$\tau(\boldsymbol{z}) \coloneqq C(0.05 + 0.95 z_1^{3/2}) \tag{4}$$

where $C \in \mathbb{R}_+$ defines the maximum runtime to evaluate f.

B.2 Hartmann Function

The following Hartmann function has 4 local minimizers for the $3D$ case and 6 local minimizers for the $6D$ case:

$$f(\boldsymbol{x}) \coloneqq -\sum_{i=1}^{4} \alpha_i \exp\left[-\sum_{j=1}^{3} A_{i,j}(x_j - P_{i,j})^2\right] \tag{5}$$

where $\boldsymbol{\alpha} = [1.0, 1.2, 3.0, 3.2]^\top$, $\boldsymbol{x} \in [0, 1]^D$, A for the $3D$ case is

$$A = \begin{bmatrix} 3 & 10 & 30 \\ 0.1 & 10 & 35 \\ 3 & 10 & 30 \\ 0.1 & 10 & 35 \end{bmatrix}, \tag{6}$$

A for the $6D$ case is

$$A = \begin{bmatrix} 10 & 3 & 17 & 3.5 & 1.7 & 8 \\ 0.05 & 10 & 17 & 0.1 & 8 & 14 \\ 3 & 3.5 & 1.7 & 10 & 17 & 8 \\ 17 & 8 & 0.05 & 10 & 0.1 & 14 \end{bmatrix}, \tag{7}$$

P for the $3D$ case is

$$P = 10^{-4} \times \begin{bmatrix} 3689 & 1170 & 2673 \\ 4699 & 4387 & 7470 \\ 1091 & 8732 & 5547 \\ 381 & 5743 & 8828 \end{bmatrix}, \tag{8}$$

[2] See the implementation of Kandasamy et al. (2020): `branin_mf.py` at `https://github.com/dragonfly/dragonfly/`.

and P for the $6D$ case is

$$P = 10^{-4} \times \begin{bmatrix} 1312 & 1696 & 5569 & 124 & 8283 & 5886 \\ 2329 & 4135 & 8307 & 3736 & 1004 & 9991 \\ 2348 & 1451 & 3522 & 2883 & 3047 & 6650 \\ 4047 & 8828 & 8732 & 5743 & 1091 & 381 \end{bmatrix}. \tag{9}$$

The multi-fidelity Hartmann function was invented by Kandasamy et al. (2020) and it replaces $\boldsymbol{\alpha}$ with the following $\boldsymbol{\alpha}_{\boldsymbol{z}}$:

$$\boldsymbol{\alpha}_{\boldsymbol{z}} \coloneqq \delta(1 - \boldsymbol{z}) \tag{10}$$

where $\boldsymbol{z} \in [0, 1]^4$ and $\delta = 0.1$ is the factor that controls the rank correlation between low- and high-fidelities. Higher δ yields less correlation. The runtime function of the multi-fidelity Hartmann function is computed as [3]:

$$\tau(\boldsymbol{z}) = \frac{1}{10} + \frac{9}{10}\frac{z_1 + z_2^3 + z_3 z_4}{3} \tag{11}$$

for the $3D$ case and

$$\tau(\boldsymbol{z}) = \frac{1}{10} + \frac{9}{10}\frac{z_1 + z_2^2 + z_3 + z_4^3}{4} \tag{12}$$

for the $6D$ case where $C \in \mathbb{R}_+$ defines the maximum runtime to evaluate f.

B.3 Tabular & Surrogate Benchmarks

In this paper, we used the MLP benchmark in Table 6 of HPOBench (Eggensperger et al. (2021)), HPOlib (Klein and Hutter (2019)), JAHS-Bench-201 (Bansal et al. (2022)), and LCBench (Zimmer et al. (2021)) in YAHPOBench (Pfisterer et al. (2022)).

HPOBench is a collection of tabular, surrogate, and raw benchmarks. In our example, we have the MLP (multi-layer perceptron) benchmark, which is a tabular benchmark, in Table 6 of the HPOBench paper (Eggensperger et al. (2021)). This benchmark has 8 classification tasks and provides the validation accuracy, runtime, F1 score, and precision for each configuration at epochs of $\{3, 9, 27, 81, 243\}$. The search space of MLP benchmark in HPOBench is provided in Table 1.

HPOlib is a tabular benchmark for neural networks on regression tasks (Slice Localization, Naval Propulsion, Protein Structure, and Parkinsons Telemonitoring). This benchmark has 4 regression tasks and provides the number of parameters, runtime, and training and validation mean squared error (MSE) for each configuration at each epoch. The search space of HPOlib is provided in Table 2.

JAHS-Bench-201 is an XGBoost surrogate benchmark for neural networks on image classification tasks (CIFAR10, Fashion-MNIST, and Colorectal Histology). This benchmark has 3 image classification tasks and provides FLOPS, latency, runtime, architecture size in megabytes, test accuracy, training accuracy, and validation accuracy for each configuration with two fidelity parameters: image resolution and epoch. The search space of JAHS-Bench-201 is provided in Table 3.

LCBench is a random-forest surrogate benchmark for neural networks on OpenML datasets. This benchmark has 34 tasks and provides training/test/validation accuracy, losses, balanced accuracy, and runtime at each epoch. The search space of HPOlib is provided in Table 4.

C Optimizers

In our package, we show examples using BOHB (Falkner et al. (2018)), DEHB (Awad et al. (2021)), SMAC3 (Lindauer et al. (2022)), and NePS [4]. BOHB is a combination of HyperBand (Li et al. (2017))

[3] See the implementation of Kandasamy et al. (2020): `hartmann3_2_mf.py` for the $3D$ case and `hartmann6_4_mf.py` for the $6D$ case at `https://github.com/dragonfly/dragonfly/`.

[4] `https://github.com/automl/neps/`

Table 1: The search space of the MLP benchmark in HPOBench (5 discrete + 1 fidelity parameters). Note that we have 2 fidelity parameters only for the raw benchmark. Each benchmark has performance metrics of 30000 possible configurations with 5 random seeds.

Hyperparameter	Choices
L2 regularization	$[10^{-8}, 1.0]$ with 10 evenly distributed grids
Batch size	$[4, 256]$ with 10 evenly distributed grids
Initial learning rate	$[10^{-5}, 1.0]$ with 10 evenly distributed grids
Width	$[16, 1024]$ with 10 evenly distributed grids
Depth	$\{1, 2, 3\}$
Epoch (**Fidelity**)	$\{3, 9, 27, 81, 243\}$

Table 2: The search space of HPOlib (6 discrete + 3 categorical + 1 fidelity parameters). Each benchmark has performance metrics of 62208 possible configurations with 4 random seeds.

Hyperparameter	Choices
Batch size	$\{2^3, 2^4, 2^5, 2^6\}$
Initial learning rate	$\{5 \times 10^{-4}, 10^{-3}, 5 \times 10^{-3}, 10^{-2}, 5 \times 10^{-2}, 10^{-1}\}$
Number of units {1,2}	$\{2^4, 2^5, 2^6, 2^7, 2^8, 2^9\}$
Dropout rate {1,2}	$\{0.0, 0.3, 0.6\}$
Learning rate scheduler	{`cosine`, `constant`}
Activation function {1,2}	{`relu`, `tanh`}
Epoch (**Fidelity**)	$[1, 100]$

and tree-structured Parzen estimator (Bergstra et al. (2011); Watanabe (2023)). DEHB is a combination of HyperBand and differential evolution. SMAC3 is an HPO framework. SMAC3 supports various Bayesian optimization algorithms and uses different strategies for different scenarios. The default strategies for MFO is the random forest-based Bayesian optimization and HyperBand. NePS is another HPO framework jointly with neural architecture search.

Table 3: The search space of JAHS-Bench-201 (2 continuous + 2 discrete + 8 categorical + 2 fidelity parameters). JAHS-Bench-201 is an XGBoost surrogate benchmark and the outputs are deterministic.

Hyperparameter	Range or choices
Learning rate	$[10^{-3}, 1]$
L2 regularization	$[10^{-5}, 10^{-2}]$
Activation function	{`ReLU`, `Hardswish`, `Mish`}
Trivial augment (Müller and Hutter (2021))	{`True`, `False`}
Depth multiplier	$\{1, 2, 3\}$
Width multiplier	$\{2^2, 2^3, 2^4\}$
Cell search space (NAS-Bench-201 (Dong and Yang (2020)), Edge $1-6$)	{`none`, `avg-pool-3x3`, `bn-conv-1x1`, `bn-conv-3x3`, `skip-connection`}
Epoch (**Fidelity**)	$[1, 200]$
Resolution (**Fidelity**)	$[0.0, 1.0]$

Table 4: The search space of HPOlib (3 discrete + 4 continuous + 1 fidelity parameters). Although the original LCBench is a collection of 2000 random configurations, YAHPOBench created random-forest surrogates over the 2000 observations. Users can choose deterministic or non-deterministic outputs.

Hyperparameter	Choices
Batch size	$[2^4, 2^9]$
Max number of units	$[2^6, 2^{10}]$
Number of layers	$[1, 5]$
Initial learning rate	$[10^{-4}, 10^{-1}]$
L2 regularization	$[10^{-5}, 10^{-1}]$
Max dropout rate	$[0.1, 0.9]$
Momentum	$[0.1, 0.9]$
Epoch (**Fidelity**)	$[1, 52]$