Multi-fidelity zero-shot HPO

Fela Winkelmolen FELAWINKELMOLEN@GMAIL.COM
Nikita Ivkin IVKIN@AMAZON.COM
H. Furkan Bozkurt BOZKURH@AMAZON.COM
Zohar Karnin ZKARNIN@AMAZON.COM

Abstract

Zero-shot hyperparameter optimization (HPO) is a simple yet effective use of transfer learning for constructing a small list of default hyperparameter (HP) configurations that complement each other. That is to say, for any given dataset, at least one of them is expected to perform well. We provide an overview of the approach and introduce a novel multi-fidelity technique that decreases the number of model evaluations, and thus cost and time, required to compute zero-shot configurations. This speedup is vital in environments where the space of HPs is regularly changing due to new algorithms being introduced. We benchmark our algorithm experimentally on two different tasks and in both cases show an improvement in accuracy compared to standard zero-shot HPO with the same offline evaluation budget.

1. Introduction

Hyperparameter tuning can be described as a blackbox optimization problem, where the goal is to minimize an expensive to evaluate blackbox function $\ell : \Theta \to \mathbb{R}$, with ℓ mapping any hyperparameter configuration $\theta \in \Theta$ to its generalization loss $\ell(\theta)$, and Θ representing the search space of all configurations that are being considered. Typically we do not have access to gradient information and desire evaluating ℓ a small number of times, as each evaluation corresponds to a costly training of a machine learning model.

Most state-of-the-art HPO tools use Bayesian Optimization (BO) as the core component of their algorithms (Hutter et al., 2010, 2009; Falkner et al., 2018; Das et al., 2020). In BO the blackbox function ℓ is modelled by a probabilistic surrogate model. After each evaluation, the returned value is incorporated into the surrogate model, which is then used to determine the next hyperparameter configuration to evaluate.

This approach has room for improvement in two aspects. First, if the surrogate model is trained from scratch, this leads to poor behavior in the initial stages, where not much is learned, and we are essentially querying random configurations. This is a problem either when the HPO budget, meaning the number of queries for ℓ, is low, or when the HP space Θ is very rich, and it takes time to learn a reasonable surrogate for ℓ. The second issue is that the BO approach is by definition sequential; this poses a problem when we wish to speed up the HPO process by using multiple machines. Although there are several adaptations of BO in the literature aimed to solve both of these issues, zero-shot HPO is a natural fit to address both. It requires learning a set of HP configurations offline based on a meta-collection of datasets, thus taking advantage of external information. Furthermore, by having a pre-determined set of configurations, the HPO procedure becomes embarrassingly parallel.

To obtain the set of configurations, previous papers (Wistuba et al., 2015a,b; Pfisterer et al., 2018) run an offline meta-training job of the following nature. Given a collection of D datasets, choose n configurations and evaluate them on all D datasets. Configurations can be picked randomly or according to some heuristic such as *eps*-net. This offline process produces a performance table, (i, j) entry of which represents the loss of i-th configuration on j-th dataset. This performance table is used to find a set of K configurations that jointly minimize the loss over the dataset collection. In order to aggregate the losses over the datasets, it is common to normalize the scores and minimize the average normalized score.

Notice that the offline procedure requires querying the function ℓ at nD points, meaning it requires nD training jobs. This might seem reasonable because it is a one-time procedure. However, we claim that that a more efficient procedure would be highly beneficial. First, algorithms have new versions with new hyper-parameters being introduced. Second, even for a one-time job, the budget is limited, meaning that we have to compromise on our choice of number of datasets D or configurations n. With that in mind we design a novel multi-fidelity algorithm solving the offline subset selection problem in a budget-efficient way. Given a fixed evaluation budget B our results greatly outperform the techniques mentioned above using nD evaluations.

2. Related work

A number of previous works attempt to use evaluations of related tasks to speed up BO. Feurer et al. (2015) and Brazdil et al. (2003) propose to start the search from configurations that performed well on similar datasets and accomplish this by computing a distance metric on meta-data of the datasets. Although the results presented in the papers are positive, it is not always clear how to collect meta-data, which explicit meta-data features would be useful, and how to automatically get the correct distance metric relevant to the specific setting being solved.

Wistuba et al. (2015a) are the first to frame zero-shot HPO as an optimization problem minimizing the meta-loss over a set of datasets. Differentiable surrogate models are used to decrease the required evaluations on the meta-datasets. This, in combination with a discrete relaxation of the minimization problem, allows the authors to optimize the meta-loss using gradient descent. They show that using their method as an initialization strategy followed by standard single task BO methods matches the performance of state-of-the-art meta-learning algorithms, as well as initialization strategies that make use of meta-data based distances.

In a second paper (Wistuba et al., 2015b), the same authors use a greedy incremental algorithm to find a set of zero-shot HPO configurations, giving similar results. In this case, a full grid search is performed on all datasets, thus not requiring any surrogate model.

Pfisterer et al. (2018) frame the zero-shot HPO problem as one of finding multiple default configurations. They combine both of the above ideas, by limiting the search space to a discrete set of random configurations (as opposed to the grid search used in (Wistuba et al., 2015b)) and use an iterative greedy algorithm for finding a sequence of zero-shot configurations, but use a surrogate model for evaluations. They provide details about neither the type of surrogate model nor the amount of savings it brought.

A different line of work attempts to use the evaluations of related tasks directly in the modeling of the surrogate function (Yogatama and Mann, 2014; Perrone et al., 2018). How-

ever, developing modeling techniques that robustly and effectively apply transfer learning on real-world tasks has proven challenging, and thus is still an area under active research. See chapter 2 of Hutter et al. (2019) for a broader review of such transfer learning techniques. We note that (1) our technique is quite different and likely complementary to that line of work, and (2) these works provide outside information, but the problem of parallelism remains.

3. Multi-fidelity Approach For the Optimization Procedure

Let us start by describing the computationally efficient algorithm, known as the greedy procedure, used by previous works.

The naive procedure

We have a collection of D datasets, and a (possibly infinite) collection Θ of configurations. In the naive approach, we select n configurations $\theta_1, \ldots, \theta_n \in \Theta$, typically by sampling.

For each configuration θ and dataset d, we run a training job to compute the validation loss $\ell(d, \theta)$. Given these values, we run an optimization procedure finding K configurations that jointly obtain the best performance on the dataset collection, by minimizing $L(S) = D^{-1} \sum_{d=1}^{D} \min_{i \in S} \{\ell(d, \theta_i)\}$.

Finding the optimal subset of size K is known to be NP-hard. This was already pointed out in Pfisterer et al. (2018). Fortunately, there is a very simple approximation algorithm using the greedy approach. We note that L is monotone decreasing, meaning for any S and i, $L(S \cup \{i\}) \leq L(S)$ and L is supermodular, where S is a subset of $[n]$ of size K. As such, we know that the greedy algorithm choosing elements for S in the way that minimizes L the most in every step provides an e-approximation (see Algorithm 1 in Appendix D).

The multi-fidelity approach

We are now ready to discuss a way to reduce the query cost of the above algorithm. Let's begin with the setting of $K = 1$. Here, the problem has a straightforward reduction to the best arm identification task in multi-armed bandits. A query to a configuration consists of selecting a random dataset and computing the loss $\ell(d, \theta)$. This is an unbiased estimator of the mean over all datasets. In our setting, we have a fixed budget of queries to ℓ. As such, a first attempt would be to use the techniques provided by Karnin et al. (2013) in their "successive halving" algorithm, proven to be highly efficient in the best arm identification task. Unfortunately, we have three added complications. First, we have a finite set of datasets, meaning that there is a maximum to the number of arm pulls. Second, when choosing an aggregation function other than mean, the confidence bounds become less trivial. This is mitigated by the analysis of Jamieson and Talwalkar (2016), showing that the successive halving algorithm works well as the estimates converge to a single value. Lastly, the number n of possible configurations θ is not pre-determined as we can choose to explore either many configurations in a shallow manner or a few but exhaustively. This is exactly the problem dealt with by HyperBand (Li et al., 2017), where they build upon the successive halving algorithm.

We conclude that for the setting of $K = 1$ the problem fits the framework of HyperBand (HB) exactly. Therefore we can use either HB or an extension of it, making use of surrogate functions such as BOHB (Falkner et al., 2018). This solution for $K = 1$ suggests a solution

for $K > 1$. We aim to find a set S of K configurations and will do it sequentially, finding the elements of S one after the other, as in the naive greedy approach. We use the term *location* for a chosen item in S to denote its location in the sequential greedy process. With this term, the algorithm chooses the item in location 1 based on the ℓ values, then chooses an item for location 2 based on its improvement over the ℓ value already achieved by location 1. It moves on to the other locations sequentially in the same way it handled location 2. Since we get an approximately optimal configuration for each location, the same analysis above will show that we obtain an approximation to the truly optimal subset S of size K.

Although this method is sensible at a high level there are a few key issues to be dealt with, that require a non-trivial solution.

Information reuse: Consider a setting where we wish to evaluate the performance of a configuration θ as the second configuration in our set. If θ happened to perform well as a first element, we queried its performance on many datasets, and the evaluation actually comes for free. In general, when we start the process of discovering the configuration for location > 1 in the set, we already have information from previous locations. We design the HyperBand instance to reuse this information.

Resource Balancing: It is unclear how we should balance the resources among the K locations. Realistically, most of the value comes from the first locations. This can be made formal by noticing the magnitude of the losses and their variances become smaller as the location grows.

The issue of resource balancing brings a major challenge. If we only start exploring location j once we fixed locations $j' < j$, we cannot be adaptive. Moreover, even if we do not wish to be adaptive, by splitting the resources in advance, for example equally, we cannot control whether we end up actually using equal resources or, due to information reuse, end up with remaining resources and an unclear way to use them. The only straightforward way would be using them for the last location, but that is hardly a choice we would have done knowingly. To this end, we modify our HyperBand implementations in a way that allows exploring all locations simultaneously.

The closest previous work we are aware of is that of Streeter and Golovin (2009). The authors provide an online algorithm that sequentially selects different sets of size K and, on average, competes with the optimal set in hindsight. Their technique provably works for any monotone decreasing supermodular function. This is a slightly different setting as the objective is to minimize regret rather than identifying the best subset. In order to handle the fact that a change in location j changes the losses observed at locations $j' > j$, they require the algorithm for selecting each location to be very robust, and indeed they use EXP3.P (Auer et al., 2002), which is a multi-arm bandit algorithm for the regret setting, for adversarial realizations and a high probability guarantee. This algorithm is indeed robust enough to provide rigorous guarantees but not very practical for our setting. In particular, it is aimed for a regret guarantee rather than best arm identification. Due to this, we still have a HyperBand instance for every location with the following modifications:

- Each HyperBand (HB) instance does not know its budget in advance. Due to this we adapt the algorithm in Li et al. (2018) that provides an anytime asynchronous version of successive halving. We modify their algorithm to be an anytime asynchronous version of HyperBand.

- We invoke the instances of the different locations via an orchestrator. This orchestrator makes sure that (1) location j is invoked only after all locations $j' < j$ have at least one candidate whose performance is known on all datasets, (2) the ratios of the budget used, excluding information reuse, are roughly proportional to some agreed-upon proportion vector.
- Once location j modified its top configuration, we change the loss scores associated with all full and partial runs in the HyperBand instance of locations $j' > j$.

We provide a more detailed description of the used algorithm in Appendix D.

4. Relative error difference (RED)

Zero-shot HPO can be used with any loss ℓ and any aggregate loss L. However, as we average losses computed over different datasets, one should be careful to choose a loss that can be meaningfully averaged across datasets.

The loss we use in our experiments is what we call *relative error difference* (RED), computed comparing to a reference loss r_d that we first compute for each dataset d. Thus, if $\tilde{\ell}(d, \theta)$ is an unnormalized error metric for dataset d, such as misclassification rate, we normalize this metric by taking

$$\ell(d, \theta) = \frac{\tilde{\ell}(d, \theta) - r_d}{\max(\tilde{\ell}(d, \theta), r_d)}$$

where we define $0/0$ to be equal to 0. As an example, if one dataset has a reference error rate of 40%, and a second dataset has a reference error rate of 4%, this metric considers a reduction of the former to 30% equivalent to a reduction of the latter to 3%. RED provides us with robust aggregation thanks to its desirable properties compared to other normalization schemes, as described in more detail in Appendix A.

5. Experimental setup

We test our approach on two of the most commonly used general purpose supervised learning algorithms: XGBoost and multi layer perceptron (MLP). For simplicity, we focus only on classification for tabular data.

For XGBoost (Chen and Guestrin, 2016) we randomly sample 9 hyperparameters, and generate 4000 random configurations which we evaluate on 65 tabular datasets.

All datasets have at least 3000 rows and originate from publicly available repositories, such as Kaggle, OpenML, and UCI. Minimal standard preprocessing, such as TF-IDF, and one hot encoding, were applied to transform the input into purely numeric matrices. To assure high generalization accuracy estimates, we use .5/.25/.25 sized splits for training, validation and test sets, respectively.

In the MLP case we use the precomputed data released by Zimmer (2020). This dataset includes evaluations of 2000 random hyperparameter configurations of a funnel shaped MLP, evaluated on 35 classification datasets.

In all our experiments our loss is the average RED as described above. The unnormalized error metric we use is the misclassification rate, and our reference metric is obtained by

averaging the test metric of the 10 models with the lowest validation error in our discrete set of random configurations.

To use the limited number of datasets efficiently we evaluate our method in a leave-one-dataset-out fashion: when computing the performance of using zero-shot configurations on any given datasets we use all other datasets as source tasks. We always report the *test metric* of the hyperparameter configuration with the best *validation error* up to any given number of evaluations.

For our multi-fidelity procedure we evaluate two settings, using total budgets of 500 and 10000 evaluations, respectively. We use $\eta = 3$, $K = 5$ and each random candidate is taken from the table described above.

6. Results

To evaluate the multi-fidelity algorithms we compare its results with the standard zero-shot HPO algorithm using an equal number of meta-evaluations. This is done by applying the baseline zero-shot algorithm on a table with a reduced number of configurations, but retaining all datasets, such that nD becomes equal to the desired offline budget. For reference we also include the results of random search Bergstra and Bengio (2012), and of the ideal zero-shot HPO scenario, where we use the full table that we compiled. As can be seen in Figure 1, the multi-fidelity algorithm makes a more efficient use of our evaluations budget, thus allowing us to find better configurations with the same budget in all of the settings that we tested.

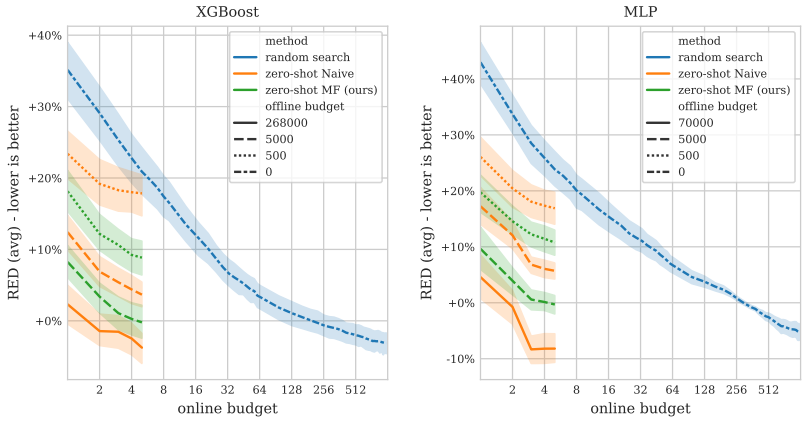

Figure 1: Results of multi-fidelity (green) and standard naive zero-shot HPO (orange), compared with random search (blue). Plotted is the average test RED of the configuration with the best validation RED, after having evaluated the number of evaluations reported on the horizontal axis, either evaluating the zero-shot configurations found, or, in the case of random search by selecting configurations at random. The style indicates the number of offline training jobs performed for each curve. The shaded areas cover one standard error in each direction.

These results clearly show that our method is beneficial both as a stand alone tool, as well as an initialization strategy to speed up BO. We hope that this will help machine learning practitioners reach state-of-the-art model performance more quickly, freeing them from some of the costly and time consuming hyperparameter tuning effort normally required.

References

Peter Auer, Nicolo Cesa-Bianchi, Yoav Freund, and Robert E Schapire. The nonstochastic multiarmed bandit problem. *SIAM journal on computing*, 32(1):48–77, 2002.

James Bergstra and Yoshua Bengio. Random search for hyper-parameter optimization. *Journal of machine learning research*, 13(Feb):281–305, 2012.

Pavel B Brazdil, Carlos Soares, and Joaquim Pinto Da Costa. Ranking learning algorithms: Using ibl and meta-learning on accuracy and time results. *Machine Learning*, 50(3): 251–277, 2003.

Tianqi Chen and Carlos Guestrin. Xgboost: A scalable tree boosting system. In *Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining*, pages 785–794, 2016.

Piali Das, Nikita Ivkin, Tanya Bansal, Laurence Rouesnel, Philip Gautier, Zohar Karnin, Leo Dirac, Lakshmi Ramakrishnan, Andre Perunicic, Iaroslav Shcherbatyi, et al. Amazon sagemaker autopilot: a white box automl solution at scale. In *Proceedings of the Fourth International Workshop on Data Management for End-to-End Machine Learning*, pages 1–7, 2020.

Stefan Falkner, Aaron Klein, and Frank Hutter. Bohb: Robust and efficient hyperparameter optimization at scale. *arXiv preprint arXiv:1807.01774*, 2018.

Matthias Feurer, Jost Tobias Springenberg, and Frank Hutter. Initializing bayesian hyperparameter optimization via meta-learning. In *Twenty-Ninth AAAI Conference on Artificial Intelligence*, 2015.

F Hutter, L Kotthoff, and J Vanschoren. Automl: methods, systems, challenges (2018). *Book in preparation. Current draft at https://www. automl. org/book/. Accessed July*, 2019.

Frank Hutter, Holger H Hoos, Kevin Leyton-Brown, and Thomas Stützle. Paramils: an automatic algorithm configuration framework. *Journal of Artificial Intelligence Research*, 36:267–306, 2009.

Frank Hutter, Holger H Hoos, and Kevin Leyton-Brown. Sequential model-based optimization for general algorithm configuration (extended version). *Technical Report TR-2010–10, University of British Columbia, Computer Science, Tech. Rep.*, 2010.

Kevin Jamieson and Ameet Talwalkar. Non-stochastic best arm identification and hyperparameter optimization. In *Artificial Intelligence and Statistics*, pages 240–248, 2016.

Zohar Karnin, Tomer Koren, and Oren Somekh. Almost optimal exploration in multi-armed bandits. In *International Conference on Machine Learning*, pages 1238–1246, 2013.

Liam Li, Kevin Jamieson, Afshin Rostamizadeh, Ekaterina Gonina, Moritz Hardt, Benjamin Recht, and Ameet Talwalkar. Massively parallel hyperparameter tuning. *arXiv preprint arXiv:1810.05934*, 2018.

Lisha Li, Kevin Jamieson, Giulia DeSalvo, Afshin Rostamizadeh, and Ameet Talwalkar. Hyperband: A novel bandit-based approach to hyperparameter optimization. *The Journal of Machine Learning Research*, 18(1):6765–6816, 2017.

Valerio Perrone, Rodolphe Jenatton, Matthias W Seeger, and Cédric Archambeau. Scalable hyperparameter transfer learning. In *Advances in Neural Information Processing Systems*, pages 6845–6855, 2018.

Florian Pfisterer, Jan N van Rijn, Philipp Probst, Andreas Müller, and Bernd Bischl. Learning multiple defaults for machine learning algorithms. *arXiv preprint arXiv:1811.09409*, 2018.

Matthew Streeter and Daniel Golovin. An online algorithm for maximizing submodular functions. In *Advances in Neural Information Processing Systems*, pages 1577–1584, 2009.

Fela Winkelmolen, Nikita Ivkin, H. Furkan Bozkurt, and Zohar Karnin. Practical and sample efficient zero-shot hpo. *arXiv preprint*, 2020.

Martin Wistuba, Nicolas Schilling, and Lars Schmidt-Thieme. Learning hyperparameter optimization initializations. In *2015 IEEE international conference on data science and advanced analytics (DSAA)*, pages 1–10. IEEE, 2015a.

Martin Wistuba, Nicolas Schilling, and Lars Schmidt-Thieme. Sequential model-free hyperparameter tuning. In *2015 IEEE international conference on data mining*, pages 1033–1038. IEEE, 2015b.

Dani Yogatama and Gideon Mann. Efficient transfer learning method for automatic hyperparameter tuning. In *Artificial intelligence and statistics*, pages 1077–1085, 2014.

Lucas Zimmer. LCBench, 2020. URL `https://github.com/automl/LCBench`.

Appendix A. Comparison of RED to other metrics

In this section we motivate our use of RED as an objective when computing the zero-shot candidates, as well as when analyzing our results by aggregating the metric across multiple datasets. In particular, we motivate the use of RED over the following normalization schemes that have been used in the literature:

- **No normalization.** We denote the unnormalized misclassification rate of the configuration θ on dataset d with $\tilde{\ell}(d, \theta)$.
- **Rank normalization.** The rank normalized score is defined as the rank among Θ, and can be normalized to be between 0 and 1:

$$\ell(d, \theta) = \frac{|\{\tilde{\theta} \in \Theta \mid \tilde{\ell}(d, \tilde{\theta}) < \tilde{\ell}(d, \theta)\}|}{|\Theta|}$$

- **Min-max normalization.** Linearly rescales the scores to keep the range between 0 and 1.
- **Stddev normalization.** Linearly rescales the scores to have mean zero and standard deviation one.
- **RED.** The relative error difference metric described in the previous section.

Wistuba et al. (2015a) use min-max normalization, while in Wistuba et al. (2015b) min-max normalization is used only when evaluating the results, but rank normalization is used in the objective. And finally, Pfisterer et al. (2018) use stddev normalization.

It is not immediately obvious what the trade-offs of different metrics are. We thus looked at the following two properties that we deem important when averaging metrics across datasets, and show that only RED has both:

Robust to rescaling. When comparing metrics averaged over multiple datasets, without using any normalization, typically only the metrics with the largest range will meaningfully affect the results. While often the true utility of any improvement depends on the use case at hand, we intuitively want—all other things being equal—that an halving of the error metric always contributes the same value. Any of the normalization schemes described above, except for no normalization, are invariant under linear rescaling.

Robust to simple datasets. There are some datasets for which in fact we do not want the results to affect the aggregate metric. These are datasets where the performance of the best configurations is very similar to the performance of the worst configuration. It is important to note that here we are talking about similarity in relative terms, not absolute terms. Reducing the error rate from 0.1% to 0.001% can be very valuable and will likely require a much better model. Conversely, reducing an error rate from 32% to 31% is, in most cases, not as impressive. All normalization schemes except RED and no normalization will magnify any difference in the case where all configurations perform very closely.

Here we focused our discussion on the per dataset metric ℓ, such that the values from different datasets can be aggregated. For the aggregation itself we use the average, as this

Table 1: Metric normalization comparison.

Normalization type	none	rank	stddev	min-max	**RED**
Robus to rescaling	no	**yes**	**yes**	**yes**	**yes**
Robust to simple datasets	**yes**	no	no	no	**yes**

Table 2: Zero-shot candidates found for XGBoost.

	zero-shot$_1$	zero-shot$_2$	zero-shot$_3$	zero-shot$_4$	zero-shot$_5$
colsample_bytree	5.70e-01	5.45e-01	8.99e-01	7.07e-01	7.43e-01
gamma	6.30e-05	1.43e-02	3.35e-06	7.6e+00	1.16e-04
learning_rate	1.41e-01	1.21e-01	4.19e-02	1.43e-02	2.87e-01
max_depth	6	5	31	15	16
min_child_weight	1	1	1	6	1
n_estimators	124	488	281	414	179
reg_alpha	5.61e-05	3.52e-04	5.39e-06	1.12e-02	1.28e+00
reg_lambda	3.57e-03	4.69e-06	1.30e-06	4.56e-04	8.65e-01
subsample	8.12e-01	8.68e-01	8.55e-01	8.03e-01	8.68e-01
	zero-shot$_6$	zero-shot$_7$	zero-shot$_8$	zero-shot$_9$	zero-shot$_{10}$
colsample_bytree	9.59e-01	7.40e-01	4.62e-01	6.66e-01	9.34e-01
gamma	1.18e-03	2.31e+00	1.58e-04	4.5e-02	9.12e-06
learning_rate	2.66e-01	4.36e-01	1.51e-02	1.07e-01	1.46e-01
max_depth	5	16	24	3	2
min_child_weight	3	1	2	1	1
n_estimators	502	19	335	398	435
reg_alpha	1.00e+00	3.62e-05	2.60e-05	1.63e-02	2.66e-04
reg_lambda	2.66e-05	1.5e-06	9.08e-05	1.05e-06	2.67e-06
subsample	9.19e-01	9.27e-01	9.39e-01	9.36e-01	7.35e-01

gives us a principled way to approximate the expected metric on new datasets. There are however two cases where it might be desirable to explore alternative aggregation functions. The fist case is when we expect the new dataset to not come from (approximately) the same distribution as the meta datasets used to learn the zero-shot configurations. The second case is when we are not interested in the expected metric, for example if even in a single dataset the metric can vary by many orders of magnitude not proportional to the actual utility of the results. In such cases different aggregation functions such as median or p90 can be warranted.

Appendix B. Zero-shot Hyperparameter Candidates

In tables 2 and 3 we report the zero-shot candidates found for XGBoost and MLP. In addition to be used to quickly tune new models, the reader can use these to easily reproduce our results.

Table 3: Zero-shot candidates found for MLP.

	zero-shot$_1$	zero-shot$_2$	zero-shot$_3$	zero-shot$_4$	zero-shot$_5$
batch_size	23	21	122	44	24
max_dropout	1.76e-01	9.80e-01	1.48e-02	2.45e-01	1.74e-01
max_units	141	76	169	261	118
num_layers	3	3	4	4	4
learning_rate	4.81e-03	1.56e-03	5.57e-02	5.59e-02	2.06e-02
momentum	7.60e-01	2.26e-01	8.33e-01	4.15e-01	5.56e-01
weight_decay	9.96e-04	7.11e-03	5.53e-03	7.47e-02	1.95e-02
	zero-shot$_6$	zero-shot$_7$	zero-shot$_8$	zero-shot$_9$	zero-shot$_{10}$
batch_size	16	68	120	128	22
max_dropout	1.65e-01	9.21e-01	4.93e-01	6.37e-01	1.90e-01
max_units	996	164	647	87	213
num_layers	4	4	4	4	4
learning_rate	8.41e-04	6.43e-03	7.09e-04	1.28e-03	2.12e-04
momentum	7.46e-01	8.63e-01	1.06e-01	2.67e-01	2.9e-01
weight_decay	4.24e-03	9.87e-02	9.46e-02	2.98e-02	6.78e-02

Appendix C. Additional results

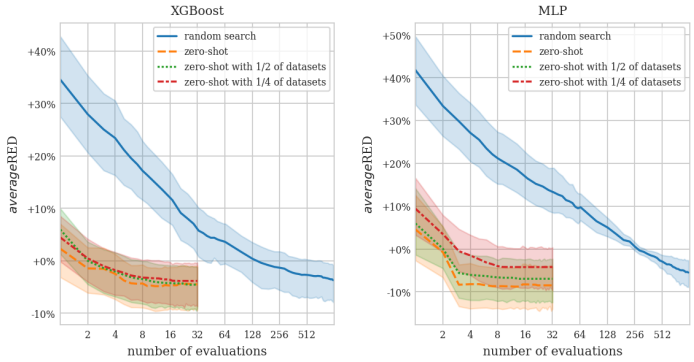

Figure 2: Ablation study of number of datasets used in optimization. We compare using different number of datasets for optimization by randomly subsampling already acquired data to 1/2 and 1/4 of it. Subsampled data include 33 and 16, 17 and 8, and 25 and 12 datasets for XGBoost, MLP and preprocessing pipelines, respectively. We repeat random subsampling 16 times.

Number of datasets. Figure 2 show the results of the ablation study comparing the number of datasets. Almost all variants match the random search baseline after using only a budget between 2 and 8 evaluations, which results in a more than 50-fold speedup. We observe that it is possible to run the offline prodecure of zero-shot HPO even with a small number of datasets and obtain considerably better performance compared to the

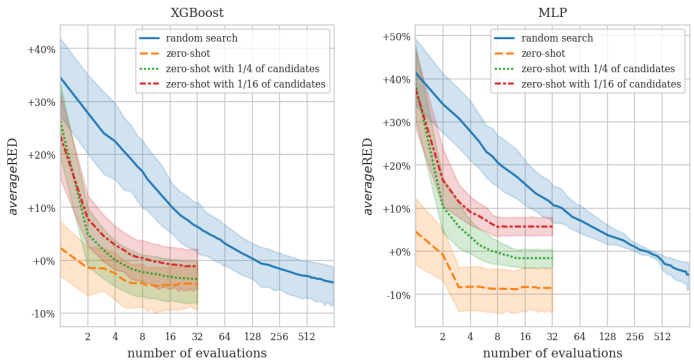

Figure 3: Ablation study of number of random candidates used in optimization. We randomly subsample random candidates to 1/4 and 1/16 of them. Subsampled data include 1000 and 250, 500 and 125, and 375 and 93 random candidates for XGBoost, MLP and preprocessing pipelines, respectively. We repeat random subsampling 10 times

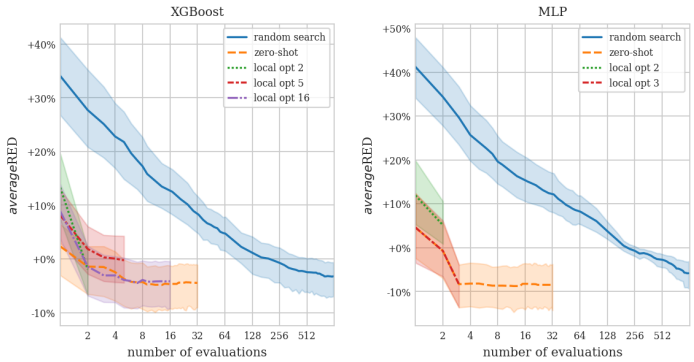

Figure 4: Ablation study of optimization strategies for varying number of zero-shot candidates used in optimization.

random search. Having more datasets however always appear to improve the quality of the configurations found.

Number of random candidates. We report the results of the ablation study comparing the number of random candidates in Figure 3. We mainly observe the effect of decreased number of random candidates when choosing the first several zero-shot candidates and the performance gap reduces as we move to more zero-shot candidates. From this we can say that it is possible to compensate running less random configurations in the offline process of zero-shot HPO by trying more zero-shot candidates.

Optimization strategies. We studied the effect of the optimization strategy used in zero-shot HPO, by running a variation of the greedy algorithm. In particular, we considered a non-incremental greedy local optimization. In this version, we start with k randomly chosen candidates and greedily optimize one of them at a time until we cannot improve in this manner anymore. We also restart the optimization several times from different set of random candidates. We compare the mentioned non-incremental optimization strategy for

varying k values to the simpler incremental approach. Note that, in the non-incremental version, one needs to rerun the optimization for different k values. We observe (Figure 4) that the non-incremental approach appears to perform consistently worse or equal than the incremental one, although this might be due to noise. This is a further indication of the simple greedy incremental approach working well in practice.

Additional results on a larger number of bigger lookup tables are provided in Winkelmolen et al. (2020), which extends the current work.

Appendix D. Details of the multi-fidelity algorithm

Algorithm 1: Greedy Optimization with Full Information

Input: number of desired zero-shot parameters K, search space Θ containing n configurations
Output: $\theta_1, ..., \theta_K$, where $\theta_i \in \Theta$
for $j = 1, 2, \ldots, K$ **do**
  set $\theta_j = \arg\min_{\theta \in \Theta} L(\{\theta_1, \ldots, \theta_{j-1}, \theta\})$
end

Algorithm 2 contains the pseudo-code for our anytime HB. Recall that the term resource used in the HB algorithm is the number of datasets. We always set the minimal resource to be 1, as our experiments showed that it gives superior results. Also, since we are choosing a random dataset, this is a very crude but unbiased estimator of the mean. Querying a configuration with resource level r means taking the average of training jobs of the HP configuration over r datasets from our collection. We split the possible resource into *rungs*, in an exponential scale. That is, for rung $s = 0$, the resource level is $r = 1$, for $s = 1$, it is $r = \eta$ (default 3), for 2 it is $r = \eta^2$, until $s = s_max$ for which it is $r = D$, the number of datasets. To avoid technical complications and cumbersome notations we simply assume D is a power of η. Algorithm 2 describes the anytime HyperBand variant. Promoting a configuration to rung s means running the HP configuration on $r = \eta^s$ datasets, or rather $\eta^s - \eta^{s-1}$ if we account for information reuse, in order to get a better estimate of its true value. The algorithm will initially query configurations in rung 0, and when it can, it will promote configurations to larger rungs. Unlike the ASHA algorithm in Li et al. (2018), that is an anytime parallel implementation of the simpler successive halving algorithm, Algorithm 2 will once in a while choose not to promote a candidate from rung s to rung $s+1$ but instead draw a new candidate and start it directly from rung $s+1$. This is exactly the idea of HyperBand that overcomes issues with regions in the configuration space that are 'unlucky' in that they are misrepresented as high loss configurations when running with low resources. The exact ratio determining when a new configuration should be chosen rather than an old one to be promoted is set so that the overall resources used by each rung towards configurations that start in that rung (as opposed to being promoted to it) are equal[1]. In other words, if we partition the configurations explored throughout the run of the algorithm according to the first rung they were launched (these are called brackets in

1. The expression of the ratio $\sum_{m=0}^{s-1} \frac{s_{\max}-s}{s_{max}-m}$ in Algorithm 2 can be shown to provide an equal balance of resources. Since this is a trivial exercise we do not prove it in this manuscript

HB), the sum of resources used by the configurations in a partition in the rung they started in should be the same.

Algorithm 2: Anytime HyperBand

Input: Configuration Selector Θ, number of datasets D, elimination ratio η (default value 3)
Output: Configuration θ
$s_{\max} = \log_\eta(D)$;
for $t \in 1, 2, \ldots$ **do**
  for $s = s_{\max} - 1, s_{\max} - 2, \ldots, 0$ **do**
    if *There exists a non-promoted candidate θ in rung s that is in the top $1/\eta$ of performers* **then**
      If we promoted over $\sum_{m=0}^{s-1} \frac{s_{\max}-s}{s_{max}-m}$ items from rung s, reset the counter of number of promoted items from rung s, select a new configuration θ_{new} from Θ, and promote it to rung $s+1$;
      Otherwise, promote θ to rung $s+1$;
    end
  end
  If no item was promoted above, select a new configuration θ_{new} from Θ and run it in rung 0;
  The best configuration θ at any time t is the one with the minimal loss among those that ran with the full resources
end

Algorithm 2 above is the building block for the main algorithm given in Algorithm 3. This algorithm provides a set of size K of default configurations that jointly minimize the loss over a collection of datasets. It acts as a coordinator over K locations of the set. For each location j we invoke an instance of Algorithm 2 to find the best configuration to improve the aggregated loss score over that obtained by the configurations found for locations $1, 2, \ldots, i$. The losses provided to location i are computed by the improvement over the configurations chosen for the smaller locations. Once the best configuration for location i is modified, we update the observed losses of all larger locations for the jobs previously run.

In order to select which location gets to query a configuration, we keep tabs of the resources used by it, in terms of number of (dataset,configuration) pairs that issued a training job, rather than a memory lookup in a query posted by the instance in charge of

the location. We balance the resources used by the different locations according to an input ratio vector ρ. For simplicity, assume ρ is the all 1 vector.

Algorithm 3: Anytime Combinatorial HyperBand

Input: Number of parameters K, Configuration Selector Θ, number of datasets D, elimination ratio η, resource ratio vector ρ

Output: A set of K configurations $\theta_1, \ldots, \theta_K$

Initialize an anytime HyperBand algorithm for each location $1, \ldots, K$;

for $t \in 1, 2, \ldots$ **do**

Let $\text{loc}_{\max}$ be the smallest location index whose HB instance does not have a ready candidate, or K if no such index exists. For each location in $i \leq \text{loc}_{\max}$, let r_i be the resource used by the HyperBand instance of location i. Choose loc as the location minimizing $r_i \cdot \rho_i$;

Run the next query issued by the HyperBand instance of location loc. The loss value observed is the (negative) improvement of losses on compared to those already achieved by the leading candidates of locations $0, 1, \ldots, \text{loc} - 1$;

If location loc changed the leading configuration based on the query, update all internal loss values for HyperBand instances of locations $\text{loc} + 1, \text{loc} + 2, \ldots$;

end

Appendix E. XGBoost performance tables and datasets

Table 4: XGBoost hyper parameters distribution.

Hyper parameter	Range	Distribution
n_estimators	$[4, 512]$	log-uniform
learning_rate	$[10^{-6}, 1]$	log-uniform
gamma	$[2^{-20}, 64]$	log-uniform
min_child_weight	$[1, 32]$	log-uniform
max_depth	$[2, 32]$	log-uniform
reg_lambda	$[2^{-20}, 1]$	log-uniform
reg_alpha	$[2^{-20}, 1]$	log-uniform
subsample	$[0.5, 1]$	uniform
colsample_bytree	$[0.3, 1]$	uniform

For performance tables we include a JSON file with the description of hyper parameters for each configuration and two 4000×65 CSV files: for validation and test metrics correspondingly.

We randomly selected 4000 configurations, each configuration consists of 9 hyper parameters, ranges and distributions are identified Table 4. All configurations were evaluated on 65 datasets selected from open source repositories, such as Kaggle, OpenML, UCI and AutoML Challenge, for detailed a list refer to the Table 5.

As mentioned earlier, we brake every dataset in ratios 50/25/25 for training, validation and test respectively. Many datasets contain missing values and non-numeric imputs, we

apply basic feature type detection (numeric, categorical and text) and preprocessing (one hot encode categorical features and apply TF-IDF transform to text features).

Table 5: XGBoost hyper parameters distribution.

UCI : Abalone, Avila, BankMarketing, BlogFeedback, ChessKingRookvsKing, ConditionBasedMaintenanceofNavalPropulsionPlants, DatasetforSensorlessDriveDiagnosis, Diabetes130UShospitalsforyears19992008, Dota2GamesResults, ElectricalGridStabilitySimulatedData, FacebookCommentVolumeDataset, Gisette, HTRU2, IDA2016Challenge, InsuranceCompanyBenchmarkCOIL2000, InternetAdvertisements, LetterRecognition, MoCapHandPostures, Nursery, OnlineShoppersPurchasingIntentionDataset, ParkinsonsTelemonitoring, PenBasedRecognitionofHandwrittenDigits, PhysicalUnclonable, PhysicochemicalPropertiesofProteinTertiaryStructure, PokerHand, SGEMMGPUkernelperformance, Spambase, StatlogLandsatSatellite, SuperconductivtyData, TurkiyeStudentEvaluation, UJIIndoorLoc, WeightLiftingExercisesmonitoredwithInertialMeasurementUnits, YearPredictionMSD, p53Mutants
AutoML Challenge : adult, albert, cadata, christine, digits, dilbert, dionis, fabert, helena, jannis, madeline, philippine, robert, sylvine, volkert, yolanda
Kaggle : blastchar/telco-customer-churn, burakhmmtgl/energy-molecule, contactprad/bike-share-daily-data, greenwing1985/housepricing, harlfoxem/housesalesprediction, jsphyg/weather-dataset-rattle-package, lodetomasi1995/income-classification, loveall/appliances-energy-prediction, lpisallerl/air-tickets-between-shanghai-and-beijing, muonneutrino/us-census-demographic-data, olgabelitskaya/classification-of-handwritten-letters, shrutimechlearn/churn-modelling, umairnsr87/predict-the-number-of-upvotes-a-post-will-get
OpenML : mushroom (24), optdigits (28), kr-vs-kp (3), pendigits (32), letter (6)