DEHB: Evolutionary Hyperband for Scalable, Robust and Efficient Hyperparameter Optimization

Noor Awad¹, Neeratyoy Mallik¹, Frank Hutter¹,²

¹Department of Computer Science, University of Freiburg, Germany
²Bosch Center for Artificial Intelligence, Renningen, Germany
{awad, mallik, fh}@cs.uni-freiburg.de

Abstract

Modern machine learning algorithms crucially rely on several design decisions to achieve strong performance, making the problem of Hyperparameter Optimization (HPO) more important than ever. Here, we combine the advantages of the popular bandit-based HPO method Hyperband (HB) and the evolutionary search approach of Differential Evolution (DE) to yield a new HPO method which we call DEHB. Comprehensive results on a very broad range of HPO problems, as well as a wide range of tabular benchmarks from neural architecture search, demonstrate that DEHB achieves strong performance far more robustly than all previous HPO methods we are aware of, especially for high-dimensional problems with discrete input dimensions. For example, DEHB is up to 1000× faster than random search. It is also efficient in computational time, conceptually simple and easy to implement, positioning it well to become a new default HPO method.

1 Introduction

Many algorithms in artificial intelligence rely crucially on good settings of their hyperparameters to achieve strong performance. This is particularly true for deep learning [Henderson et al., 2018; Melis et al., 2018], where dozens of hyperparameters concerning both the neural architecture and the optimization & regularization pipeline need to be instantiated. At the same time, modern neural networks continue to get larger and more computationally expensive, making the need for efficient hyperparameter optimization (HPO) more important.

We believe that a practical, general HPO method must fulfill many desiderata, including: (1) strong anytime performance, (2) strong final performance with a large budget, (3) effective use of parallel resources, (4) scalability w.r.t. the dimensionality and (5) robustness & flexibility. These desiderata drove the development of BOHB [Falkner et al., 2018], which satisfied them by combining the best features of Bayesian optimization via Tree Parzen estimates (TPE) [Bergstra et al., 2011] (in particular, strong final performance), and the many advantages of bandit-based HPO via Hyperband [Li et al., 2017]. While BOHB is among the best general-purpose HPO methods we are aware of, it still has problems with optimizing discrete dimensions and does not scale as well to high dimensions as one would wish. Therefore, it does not work well on high-dimensional HPO problems with discrete dimensions and also has problems with tabular neural architecture search (NAS) benchmarks (which can be tackled as high-dimensional discrete-valued HPO benchmarks, an approach followed, e.g., by regularized evolution (RE) [Real et al., 2019]).

The main contribution of this paper is to further improve upon BOHB to devise an effective general HPO method, which we dub DEHB. DEHB is based on a combination of the evolutionary optimization method of differential evolution (DE [Storn and Price, 1997]) and Hyperband and has several useful properties:

1. DEHB fulfills all the desiderata of a good HPO optimizer stated above, and in particular achieves more robust strong final performance than BOHB, especially for high-dimensional and discrete-valued problems.
2. DEHB is conceptually simple and can thus be easily re-implemented in different frameworks.
3. DEHB is computationally cheap, not incurring the overhead typical of most BO methods.
4. DEHB effectively takes advantage of parallel resources.

After discussing related work (Section 2) and background on DE and Hyperband (Section 3), Section 4 describes our new DEHB method in detail. Section 5 then presents comprehensive experiments on artificial toy functions, surrogate benchmarks, Bayesian neural networks, reinforcement learning, and 13 different tabular neural architecture search benchmarks, demonstrating that DEHB is more effective and robust than a wide range of other HPO methods, and in particular up to 1000× times faster than random search (Figure 7) and up to 32× times faster than BOHB (Figure 11) on HPO problems; on toy functions, these speedup factors even reached 33 440× and 149×, respectively (Figure 6)¹.

2 Related Work

HPO as a black-box optimization problem can be broadly tackled using two families of methods: model-free meth-
ods, such as evolutionary algorithms, and model-based Bayesian optimization methods. *Evolutionary Algorithms (EAs) are model-free population-based methods which generally include a method of initializing a population; mutation, crossover, selection operations; and a notion of fitness. EAs are known for black-box optimization in a HPO setting since the 1980s [Grefenstette, 1986]. They have also been popular for designing architectures of deep neural networks [Angelino et al., 1994; Xie and Yuille, 2017; Real et al., 2017; Liu et al., 2017]; recently, Regularized Evolution (RE) [Real et al., 2019] achieved state-of-the-art results on ImageNet.*

Bayesian optimization (BO) uses a probabilistic model based on the already observed data points to model the objective function and to trade off exploration and exploitation. The most commonly used probabilistic model in BO are Gaussian processes (GP) since they obtain well-calibrated and smooth uncertainty estimates [Snoek et al., 2012]. However, GP-based models have high complexity, do not natively scale well to high dimensions and do not apply to complex spaces without prior knowledge; alternatives include tree-based methods [Bergstra et al., 2011; Hutter et al., 2011] and Bayesian neural networks [Springenberg et al., 2016].

Recent so-called *multi-fidelity* methods exploit cheap approximations of the objective function to speed up the optimization [Liu et al., 2016; Wang et al., 2017]. Multi-fidelity optimization is also popular in BO, with Fabolas [Klein et al., 2016] and Dragonfly [Kandasamy et al., 2020] being GP-based examples. The popular method BOHB [Falkner et al., 2018], which combines BO and the bandit-based approach Hyperband [Li et al., 2017], has been shown to be a strong off-the-shelf HPO method and to the best of our knowledge is the best previous off-the-shelf multi-fidelity optimizer.

## 3 Background

### 3.1 Differential Evolution (DE)

In each generation \( g \), DE uses an evolutionary search based on difference vectors to generate new candidate solutions. DE is a population-based EA which uses three basic iterative steps (mutation, crossover and selection). At the beginning of the search on a \( D \)-dimensional problem, we initialize a population of \( N \) individuals \( x_{i,g} = (x_{i,g}^1, x_{i,g}^2, \ldots, x_{i,g}^D) \) randomly within the search range of the problem being solved. Each individual \( x_{i,g} \) is evaluated by computing its corresponding objective function value. Then the mutation operation generates a new offspring for each individual. The canonical DE uses a mutation strategy called *rand/1*, which selects three random parents \( x_{r_1}, x_{r_2}, x_{r_3} \) to generate a new mutant vector \( v_{i,g} \) for each \( x_{i,g} \) in the population as shown in Eq. 1 where \( F \) is a scaling factor parameter and takes a value within the range \((0,1)\).

\[
v_{i,g} = x_{r_1,g} + F \cdot (x_{r_2,g} - x_{r_3,g}). \tag{1}
\]

The crossover operation then combines each individual \( x_{i,g} \) and its corresponding mutant vector \( v_{i,g} \) to generate the final offspring/child \( u_{i,g} \). The canonical DE uses a simple binomial crossover to select values from \( v_{i,g} \) with a probability \( p \) (called crossover rate) and \( x_{i,g} \) otherwise. For the members \( x_{i,g+1} \) of the next generations, DE then uses the better of \( x_{i,g} \) and \( u_{i,g} \). More details on DE can be found in appendix A.

### 3.2 Successive Halving (SH) and Hyperband (HB)

Successive Halving (SH) [Jamieson and Talwalkar, 2016] is a simple yet effective multi-fidelity optimization method that exploits the fact that, for many problems, low-cost approximations of the expensive blackbox functions exist, which can be used to rule out poor parts of the search space at little computational cost. Higher-cost approximations are only used for a small fraction of the configurations to be evaluated. Specifically, an *iteration of SH* starts by sampling \( N \) configurations uniformly at random, evaluating them at the lowest-cost approximation (the so-called *lowest budget*), and forwarding a fraction of the top \( 1/\eta \) of them to the next budget (function evaluations at which are expected to be roughly \( \eta \) more expensive). This process is repeated until the highest budget, used by the expensive original blackbox function, is reached. Once the runs on the highest budget are complete, the current SH iteration ends, and the next iteration starts with the lowest budget. We call each such fixed sequence of evaluations from lowest to highest budget a *SH bracket*. While SH is often very effective, it is not guaranteed to converge to the optimal configuration even with infinite resources, because it can drop poorly-performing configurations at low budgets that actually might be the best with the highest budget.

Hyperband (HB) [Li et al., 2017] solves this problem by hedging its bets across different instantiations of SH with successively larger lowest budgets, thereby being provably at most a constant times slower than random search. In particular, this procedure also allows to find configurations that are strong for higher budgets but would have been eliminated for lower budgets. Algorithm 2 in Appendix B shows the pseudocode for HB with the SH subroutine. One iteration of HB (also called *HB bracket*) can be viewed as a sequence of SH brackets with different starting budgets and different numbers of configurations for each SH bracket. The precise budgets and number of configurations per budget are determined by HB given its 3 parameters: *minimum budget*, *maximum budget*, and \( \eta \).

The main advantages of HB are its simplicity, theoretical guarantees, and strong anytime performance compared to optimization methods operating on the full budget. However, HB can perform worse than BO and DE for longer runs since it only selects configurations based on random sampling and does not learn from previously sampled configurations.

## 4 DEHB

We design DEHB to satisfy all the desiderata described in the introduction (Section 1). DEHB inherits several advantages from HB to satisfy some of these desiderata, including its strong anytime performance, scalability and flexibility. From the DE component, it inherits robustness, simplicity, and computational efficiency. We explain DEHB in detail in the remainder of this section; full pseudocode can be found in Algorithm 3 in Appendix C.

### 4.1 High-Level Overview

A key design principle of DEHB is to share information across the runs it executes at various budgets. DEHB maintains a *subpopulation* for each of the budget levels, where
the population size for each subpopulation is assigned as the maximum number of function evaluations HB allocates for the corresponding budget.

We borrow nomenclature from HB and call the HB iterations that DEHB uses DEHB iterations. Figure 1 illustrates one such iteration, where minimum budget, maximum budget, and \( \eta \) are 1, 27, and 3, respectively. The topmost sphere for SH Bracket 1, is the first step, where 27 configurations are sampled uniformly at random and evaluated at the lowest budget 1. These evaluated configurations now form the DE subpopulation associated with budget 1. The dotted arrow pointing downwards indicates that the top-9 configurations (27/3) are promoted to be evaluated on the next higher budget 3 to create the DE subpopulation associated with budget 3, and so on until the highest budget. This progressive increase of the budget by \( \eta \) and decrease of the number of configurations evaluated by \( \eta \) is simply the vanilla SH. Indeed, each SH bracket for this first DEHB iteration is basically executing vanilla SH, starting from different minimum budgets, just like in HB.

One difference from vanilla SH is that random sampling of configurations occurs only once: in the first step of the first SH bracket of the first DEHB iteration. Every subsequent SH bracket begins by reusing the subpopulation updated in the previous SH bracket, and carrying out a DE evolution (detailed in Section 4.2). For example, for SH bracket 2 in Figure 1, the subpopulation of 9 configurations for budget 3 (topmost sphere) is propagated from SH bracket 1 and undergoes evolution. The top 3 configurations \( (9/3) \) then affect the population for the next higher budget of 9 of SH bracket 2. Specifically, these will used as the so-called parent pool for that higher budget, using the modified DE evolution to be discussed in Section 4.2. The end of SH Bracket 4 marks the end of this DEHB iteration. We dub DEHB’s first iteration its initialization iteration. At the end of this iteration, all DE subpopulations associated with the higher budgets are seeded with configurations that performed well in the lower budgets. In subsequent SH brackets, no random sampling occurs anymore, and the search runs separate DE evolutions at different budget levels, where information flows from the subpopulations at lower budgets to those at higher budgets through the modified DE mutation (Fig. 3).

### 4.2 Modified Successive Halving using DE Evolution

We now discuss the deviations from vanilla SH by elaborating on the design of a SH bracket inside DEHB, highlighted with a box in Figure 1 (SH Bracket 1). In DEHB, the top-performing configurations from a lower budget are not simply promoted and evaluated on a higher budget (except for the Initialization SH bracket). Rather, in DEHB, the top-performing configurations are collected in a Parent Pool (Figure 2). This pool is responsible for transfer of information from a lower budget to the next higher budget, but not by directly suggesting best configurations from the lower budget for re-evaluation at a higher budget. Instead, the parent pool represents a good performing region w.r.t. the lower budget, from which parents can be sampled for mutation. Figure 3b demonstrates how a parent pool contributes in a DE evolution in DEHB. Unlike in vanilla DE (Figure 3a), in DEHB, the mutants involved in DE evolution are extracted from the parent pool instead of the population itself. This allows the evolution to incorporate and combine information from the current budget, and also from the decoupled search happening on the lower budget. The selection step as shown in Figure 3 is responsible for updating the current subpopulation if the new suggested configuration is better. If not, the existing configuration is retained in the subpopulation. This guards
against cases where performance across budget levels is not correlated and good configurations from lower budgets do not improve higher budget scores. However, search on the higher budget can still progress, as the first step of every SH bracket performs vanilla DE evolution (there is no parent pool to receive information from). Thereby, search at the required budget level progresses even if lower budgets are not informative.

Additionally, we also construct a global population pool consisting of configurations from all the subpopulations. This pool does not undergo any evolution and serves as the parent pool in the edge case where the parent pool is smaller than the minimum number of individuals required for the mutation step. For the example in Figure 2, under the rand1 strategy (which requires three parents), we see that for the highest budget, only one configuration \((3/n)\) is included from the previous budget. In such a scenario, the additional two required parents are sampled from the global population pool.

### 4.3 DEHB efficiency and parallelization

As mentioned previously, DEHB carries out separate DE searches at each budget level. Moreover, the DE operations involved in evolving a configuration are constant in operation and time. Therefore, DEHB’s runtime overhead does not grow over time, even as the number of performed function evaluations increases; this is in stark contrast to model-based methods, whose time complexity is often cubic in the number of performed function evaluations. Indeed, Figure 4 demonstrates that, for a tabular benchmark with negligible cost for function evaluations, DEHB is almost 2 orders of magnitude faster than BOHB to perform 13336 function evaluations. GP-based Bayesian optimization tools would require approximations to even fit a single model with this number of function evaluations.

We also briefly describe a parallel version of DEHB (see Appendix C.3 for details of its design). Since DEHB can be viewed as a sequence of predetermined SH brackets, the SH brackets can be asynchronously distributed over free workers. A central \textit{DEHB Orchestrator} keeps a single copy of all DE subpopulations, allowing for asynchronous, immediate DE evolution updates. Figure 5 illustrates that this parallel version achieves linear speedups for similar final performance.

### 5 Experiments

We now comprehensively evaluate DEHB, illustrating that it is more robust and efficient than any other HPO method we are aware of. To keep comparisons fair and reproducible, we use a broad collection of publicly-available HPO and NAS benchmarks: all HPO benchmarks that were used to demonstrate the strength of BOHB [Falkner et al., 2018] and also a broad collection of 13 recent tabular NAS benchmarks represented as HPO problems [Awad et al., 2020].

In this section, to avoid cluttered plots we present a focused comparison of DEHB with BOHB, the best previous off-the-shelf multi-fidelity HPO method we are aware of, which has in turn outperformed a broad range of competitors (GP-BO, TPE, SMAC, HB, Fabolas, MTBO, and HB-LCNet) on these benchmarks [Falkner et al., 2018]. For reference, we also include the obligatory random search (RS) baseline in these plots, showing it to be clearly dominated, with up to 1000-fold speedups. We also provide a comparison against a broader range of methods at the end of this section (see Figure 13 and Table 1), with a full comparison in Appendix D. We also compare to the recent GP-based multi-fidelity BO tool Dragonfly in Appendix D.7. Details for the hyperparameter values of the used algorithms can be found in Appendix D.1.

We use the same parameter settings for mutation factor \(F = 0.5\) and crossover rate \(p = 0.5\) for both DE and DEHB. The population size for DEHB is not user-defined but set by its internal Hyperband component while we set it to 20 for DE following [Awad et al., 2020]. Unless specified otherwise, we report results from 50 runs for all algorithms, plotting the validation regret\(^3\) over the cumulative cost incurred by the function evaluations, and ignoring the optimizers’ overhead in order to not give DEHB what could be seen as an unfair advantage.\(^4\) We also show the speedups that DEHB achieves compared to RS and BOHB, where this is possible without adding clutter.

\(^3\)We leave out the 2-dimensional SVM surrogate benchmarks since all multi-fidelity algorithms performed similarly for this easy task, without any discernible difference.

\(^4\)This is the difference of validation score from the global best.

\(^5\)Shaded bands in plots represent the standard error of the mean.
5.1 Artificial Toy Function: Stochastic Counting Ones
This toy benchmark by Falkner et al. [2018] is useful to assess scaling behavior and ability to handle binary dimensions. The goal is to minimize the following objective function:

\[ f(x) = -\left( \sum_{x \in X_{\text{cat}}} x + \sum_{x \in X_{\text{cont}}} \mathbb{E}_b([B_p=x]) \right), \]

where the sum of the categorical variables \( x_i \in \{0, 1\} \) represents the standard discrete counting ones problem. The continuous variables \( x_j \in [0, 1] \) represent the stochastic component, with the budget \( b \) controlling the noise. The budget here represents the number of samples used to estimate the mean of the Bernoulli distribution \( (B) \) with parameters \( x_j \). Following Falkner et al. [2018], we run 4 sets of experiments with \( N_{\text{cont}} = N_{\text{cat}} = \{4, 8, 16, 32\} \), where \( N_{\text{cont}} = |X_{\text{cont}}| \) and \( N_{\text{cat}} = |X_{\text{cat}}| \), using the same budget spacing and plotting the normalized regret: \( (f(x) + d)/d \), where \( d = N_{\text{cat}} + N_{\text{cont}} \). Although this is a toy benchmark it can offer interesting insights since the search space has mixed binary/continuous dimensions which DEHB handles well (refer to C.2 in Appendix for more details). In Figure 6, we consider the 64-dimensional space \( N_{\text{cont}} = N_{\text{cat}} = 32 \); results for the lower dimensions can be found in Appendix D.2. Both BOHB and DEHB begin with a set of randomly sampled individuals evaluated on the lowest budget. It is therefore unsurprising that in Figure 6 (and in other experiments too), these two algorithms follow a similar optimization trace at the beginning of the search. Given the high dimensionality, BOHB requires many more samples to switch to model-based search which slows its convergence in comparison to the lower dimensional cases \( (N_{\text{cont}} = N_{\text{cat}} = \{4, 8, 16\}) \). In contrast, DEHB’s convergence rate is almost agnostic to the increase in dimensionality.

5.2 Surrogates for Feedforward Neural Networks
In this experiment, we optimize six architectural and training hyperparameters of a feed-forward neural network on six different datasets from OpenML [Vanschoren et al., 2014], using a surrogate benchmark built by Falkner et al. [2018]. The budgets are the training epochs for the neural networks. For all six datasets, we observe a similar pattern of the search trajectory, with DEHB and BOHB having similar anytime performance and DEHB achieving the best final score. An example is given in Figure 7, also showing a 1000-fold speedup over random search; qualitatively similar results for the other 5 datasets are in Appendix D.3.

5.3 Bayesian Neural Networks
In this benchmark, introduced by Falkner et al. [2018], a two-layer fully-connected Bayesian Neural Network is trained using stochastic gradient Hamiltonian Monte-Carlo sampling (SGHMC) [Chen et al., 2014] with scale adaptation [Springenberg et al., 2016]. The budgets were the number of MCMC steps \( (500 \text{ as minimum}; 10000 \text{ as maximum}) \). Two regression datasets from UCI [Dua and Graff, 2017] were used for the experiments: Boston Housing and Protein Structure. Figure 8 shows the results (for Boston housing; the results for Protein Structure are in Appendix D.4). For this extremely noisy benchmark, BOHB and DEHB perform similarly, and both are about \( 2 \times \) faster than RS.

5.4 Reinforcement Learning
For this benchmark used by Falkner et al. [2018]), a proximal policy optimization (PPO) [Schulman et al., 2017] implementation is parameterized with 7 hyperparameters. PPO is used to learn the cartpole swing-up task from the OpenAI Gym [Brockman et al., 2016] environment. We plot the mean number of episodes needed until convergence for a configuration over actual cumulative wall-clock time in Figure 9. Despite the strong noise in this problem, BOHB and DEHB are able to improve continuously, showing similar performance, and speeding up over random search by roughly \( 2 \times \).

5.5 NAS Benchmarks
In this series of experiments, we evaluate DEHB on a broad range of NAS benchmarks. We use a total of 13 tabular benchmarks from NAS-Bench-101 [Ying et al., 2019], NAS-Bench-1shot1 [Zela et al., 2020], NAS-Bench-201 [Dong and Yang, 2020] and NAS-HPO-Bench [Klein and Hutter, 2019]. For NAS-Bench-101, we show results on CifarC (a mixed data type encoding of the parameter space [Awad et al., 2020]) in Figure 10; BOHB and DEHB initially perform similarly as RS for this dataset, since there is only little correlation between runs with few epochs (low budgets) and many epochs (high budgets) in NAS-Bench-101. In the end, RS
stagnates, BOHB stagnates at a slightly better performance, and DEHB continues to improve. In Figure 11, we report results for ImageNet16-120 from NAS-201. In this case, DEHB is clearly the best of the methods, quickly converging to a strong solution.

Finally, Figure 12 reports results for the Protein Structure dataset provided in NAS-HPO-Bench. DEHB makes progress faster than BOHB to reach the optimum. The results on other NAS benchmarks are qualitatively similar to these 3 representative benchmarks, and are given in Appendix D.6.

5.6 Results summary

We now compare DEHB to a broader range of baseline algorithms, also including HB, TPE [Bergstra et al., 2011], SMAC [Hutter et al., 2011], regularized evolution (RE) [Real et al., 2019], and DE. Based on the mean validation regret, all algorithms can be ranked for each benchmark, for every second of the estimated wallclock time. Arranging the mean regret per timepoint across all benchmarks (except the Stochastic Counting Ones and the Bayesian Neural Network benchmarks, which do not have runtimes as budgets), we compute the average relative rank over time for each algorithm in Figure 13, where all 8 algorithms were given the mean rank of 4.5 at the beginning. The shaded region clearly indicates that DEHB is the most robust algorithm for this set of benchmarks (discussed further in Appendix D.8). In the end, RE and DE are similarly good, but these blackbox optimization algorithms perform worst for small compute budgets, while DEHB’s multi-fidelity aspect makes it robust across compute budgets. In Table 1, we show the average rank of each algorithm based on the final validation regret achieved across all benchmarks (now also including Stochastic Counting Ones and Bayesian Neural Networks; data derived from Table 1 in Appendix D.8). Next to its strong anytime performance, DEHB also yields the best final performance in this comparison, thus emerging as a strong general optimizer that works consistently across a diverse set of benchmarks. Result tables and figures for all benchmarks can be found in Appendix D.

<table>
<thead>
<tr>
<th>Algorithm</th>
</tr>
</thead>
<tbody>
<tr>
<td>Avg. rank</td>
</tr>
<tr>
<td>RS</td>
</tr>
<tr>
<td>HB</td>
</tr>
<tr>
<td>BOHB</td>
</tr>
<tr>
<td>TPE</td>
</tr>
<tr>
<td>SMAC</td>
</tr>
<tr>
<td>RE</td>
</tr>
<tr>
<td>DE</td>
</tr>
<tr>
<td>DEHB</td>
</tr>
</tbody>
</table>

Table 1: Mean ranks based on final mean validation regret for all algorithms tested for all benchmarks.

6 Conclusion

We introduced DEHB, a new, general HPO solver, built to perform efficiently and robustly across many different problem domains. As discussed, DEHB satisfies the many requirements of such an HPO solver: strong performance with both short and long compute budgets, robust results, scalability to high dimensions, flexibility to handle mixed data types, parallelizability, and low computational overhead. Our experiments show that DEHB meets these requirements and in particular yields much more robust performance for discrete and high-dimensional problems than BOHB, the previous best overall HPO method we are aware of. Indeed, in our experiments, DEHB was up to $32 \times$ faster than BOHB and up to $1000 \times$ faster than random search. DEHB does not require advanced software packages, is simple by design, and can easily be implemented across various platforms and languages, allowing for practical adoption. We thus hope that DEHB will become a new default HPO method. Our reference implementation of DEHB is available at https://github.com/automl/DEHB.

Acknowledgements. The authors acknowledge funding by the Robert Bosch GmbH, by the German Federal Ministry of Education and Research (BMBF, grant Renormalized-Flows 01IS19077C), and support by the state of Baden-Württemberg through bwHPC and the German Research Foundation (DFG) through grant no INST 39/963-1 FUGG.
References


