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

Noor Awad[1] , **Neeratyoy Mallik**[1] , **Frank Hutter**[1,2]

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

A More details on DE

Differential Evolution (DE) is a simple, well-performing evolutionary algorithm to solve a variety of optimization problems [K. Price and Lampinen, 2006] [Das *et al.*, 2016]. This algorithm was originally introduced in 1995 by Storn and Price [Storn and Price, 1997], and later attracted the attention of many researchers to propose new improved state-of-the-art algorithms [Chakraborty, 2008]. DE is based on four steps: initialization, mutation, crossover and selection. Algorithm 1 presents the DE pseudo-code.

Initialization. DE is a population-based meta-heuristic algorithm which consists of a population of N individuals. Each individual is considered a solution and expressed as a vector of D-dimensional decision variables as follows:

$$pop_g = (x^1_{i,g}, x^2_{i,g}, ..., x^D_{i,g}), i = 1, 2, ..., N, \quad (1)$$

where g is the generation number, D is the dimension of the problem being solved and N is the population size. The algorithm starts initially with randomly distributed individuals within the search space. The function value for the problem being solved is then computed for each individual, $f(x)$.

Mutation. A new child/offspring is generated using the mutation operation for each individual in the population by a so called mutation strategy. Figure 1 illustrates this operation for a 2-dimensional case. The classical DE uses the mutation operator $rand/1$, in which three random individuals/parents denoted as $x_{r_1}, x_{r_2}, x_{r_3}$ are chosen to generate a new vector v_i as follows:

$$v_{i,g} = x_{r_1,g} + F \cdot (x_{r_2,g} - x_{r_3,g}), \quad (2)$$

where $v_{i,g}$ is the mutant vector generated for each individual $x_{i,g}$ in the population. F is the scaling factor that usually takes values within the range (0, 1] and r_1, r_2, r_3 are the indices of different randomly-selected individuals. Eq.2 allows some parameters to be outside the search range, therefore, each parameter in $v_{i,g}$ is checked and reset[1] if it happens to be outside the boundaries.

Crossover. When the mutation phase is completed, the crossover operation is applied to each target vector $x_{i,g}$ and its corresponding mutant vector $v_{i,g}$ to generate a trial vector $u_{i,g}$. Classical DE uses the following uniform (binomial) crossover:

$$u^j_{i,g} = \begin{cases} v^j_{i,g} & \text{if } (rand \leq p) \text{ or } (j = j_{rand}) \\ x^j_{i,g} & \text{otherwise} \end{cases} \quad (3)$$

The crossover rate p is real-valued and is usually specified in the range [0, 1]. This variable controls the portion of parameter values that are copied from the mutant vector. The jth parameter value is copied from the mutant vector $v_{i,g}$ to the corresponding position in the trial vector $u_{i,g}$ if a random number is less than or equal to p. If the condition is not satisfied, then the jth position is copied from the target vector $x_{i,g}$. j_{rand} is a random integer in the range [1, D] to ensure that at least one dimension is copied from the mutant, in case the random number generated for all dimensions is $>p$. Figure 2 shows an illustration of the crossover operations.

Selection. After the final offspring is generated, the selection operation takes place to determine whether the target (the parent, $x_{i,g}$) or the trial (the offspring, $u_{i,g}$) vector survives to the next generation by comparing the function values. The offspring replaces its parents if it has a better[2] function

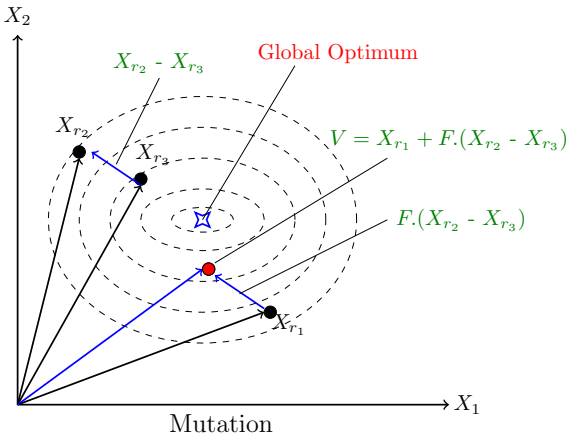

Figure 1: Illustration of DE Mutation operation for a 2-dimensional case using the *rand/1* mutation strategy. The scaled difference vector $(F.(x_{r_2} - x_{r_3}))$ is used to determine the neighbourhood of search from x_{r_1}. Depending on the diversity of the population, DE mutation's search will be explorative or exploitative

[1]a random value from $[0, 1]$ is chosen uniformly in this work

[2]DE is a minimizer

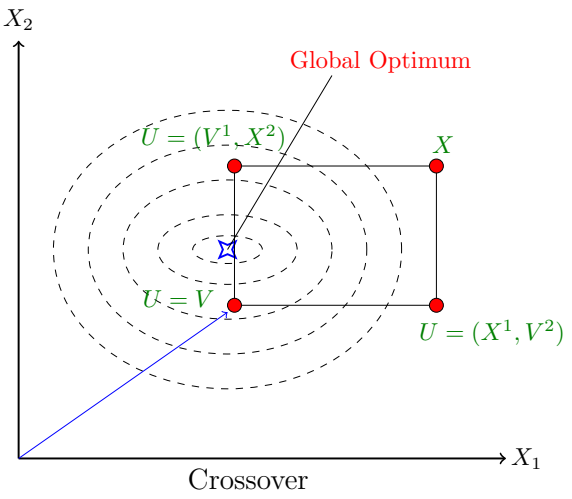

Figure 2: Illustration of DE Crossover operation for a 2-dimensional case using the *binomial* crossover. The vertex of the rectangle shows the possible solutions of between a parent x and mutant v. Based on the choice of p, the resultant individual will either be a copy of the parent, or the mutant, or incorporate either component from parent and mutant

value as shown in Equation 4. Otherwise, the new offspring is discarded, and the target vector remains in the population for the next generation.

$$x_{i,g} = \begin{cases} u_{i,g} & \text{if } (f(u_{i,g}) \leq f(x_{i,g})) \\ x_{i,g} & \text{otherwise} \end{cases} \tag{4}$$

Algorithm 1: DE_Optimizer

Input:
f - black-box problem
F - scaling factor (default $F = 0.5$)
p - crossover rate (default $p = 0.5$)
N - population size
Output: Return best found individual in pop

1 $g = 0$, $FE = 0$;
2 $pop_g \leftarrow$ initial_population(N, D);
3 $fitness_g \leftarrow$ evaluate_population(pop_g);
4 $FE = N$;
5 **while** $(FE < FE_{max})$ **do**
6 mutate(pop_g);
7 $offspring_g \leftarrow$ crossover(pop_g);
8 $fitness_g \leftarrow$ evaluate_population($offspring_g$);
9 pop_{g+1},$fitness_{g+1} \leftarrow$ select(pop_g,$offspring_g$);
10 $FE = FE + N$;
11 $g = g+1$;
12 **end**
13 **return** Individual with highest fitness seen

B More details on Hyperband

The Hyperband [Li *et al.*, 2017] (HB) algorithm was designed to perform random sampling with early stopping based on pre-determined geometrically spaced resource allocation. For DEHB we replace the random sampling with DE search. However, DEHB uses HB at its core to solve the "n versus B/n" tradeoff that HB was designed to address. Algorithm 2 shows how DEHB interfaces HB to query the sequence of how many configurations of each budget to run at each iteration. This view treats the DEHB algorithm as a sequence of predetermined (by HB), repeating Successive Halving brackets where, *iteration* number refers to the index of SH brackets run by DEHB.

Algorithm 2: A SH bracket under Hyperband

Input:
b_{min}, b_{max} - min and max budgets
η - fraction of configurations promoted
$iteration$ - iteration number
Output: List of no. of configurations and budgets

1 $s_{max} = \lfloor log_\eta \frac{b_{max}}{b_{min}} \rfloor$
2 $s = s_{max} - (iteration \bmod (s_{max} + 1))$
3 $N = \lceil \frac{s_{max}+1}{s+1} \cdot \eta^s \rceil$
4 $b_0 = \frac{b_{max}}{b_{min}} \cdot \eta^{-s}$
5 $budgets$ = $n\_configs$ = []
6 **for** $i \in \{0, ..., s\}$ **do**
7 $N_i = \lfloor N \cdot \eta^{-i} \rfloor$
8 $b = b_0 \cdot \eta^i$
9 $n\_configs$.append(N_i)
10 $budgets$.append(b)
11 **end**
12 **return** $n\_configs$, $budgets$

C More details on DEHB

C.1 DEHB algorithm

Algorithm 3 gives the pseudo code describing DEHB. DEHB takes as input the parameters for HB (b_{min}, b_{max}, η) and the parameters for DE (F, p). For the experiments in this paper, the *termination_condition* was chosen as the total number of DEHB brackets to run. However, in our implementation it can also be specified as the total absolute number of function evaluations, or a cumulative wallclock time as budget. L6 is the call to Algorithm 2 which gives a list of *budgets* which represent the sequence of increasing budgets to be used for that SH bracket. The nomenclature DE[*budgets[i]*], used in L9 and L12, indicates the DE subpopulation associated with the *budgets[i]* fidelity level. The *if...else* block from L11-15 differentiates the first DEHB bracket from the rest. During the first DEHB bracket (*bracket_counter*$== 0$) and its second SH bracket onwards (i>0), the top configurations from the lower fidelity are *promoted*[3] for evaluation in the next higher fidelity. The function *DE_trial_generation* on L14, i.e, the sequence of mutation-crossover operations, generates a candidate configuration (*config*) to be evaluated for

[3]only *evaluate* on higher budget and not evolve using mutation-crossover-selection

all other scenarios. L17 carries out the DE selection procedure by comparing the fitness score of *config* and the selected *target* for that DE evolution step. The *target* ($x_{i,g}$ from Equation 3) is selected on L9 by a rolling pointer over the subpopulation list. That is, for every iteration (every increment of j) a pointer moves forward by one index position in the subpopulation selecting an individual to be a *target*. When this pointer reaches the maximal index, it resets to point back to the starting index of the subpopulation. L18 compares the score of the last evaluated *config* with the best found score so far. If the new *config* has a better fitness score, the best found score is updated and the new *config* is marked as the incumbent, $config_{inc}$. This stores the best found configuration as an *anytime* best performing configuration.

Algorithm 3: DEHB

Input:
b_{min}, b_{max} - min and max budgets
η - (default η=3)
F - scaling factor (default $F = 0.5$)
p - crossover rate (default $p = 0.5$)
Output: Best found configuration, $config_{inc}$

1 $s_{max} = \lfloor log_{\eta} \frac{b_{max}}{b_{min}} \rfloor$
2 *Initialize* $(s_{max} + 1)$ DE subpopulations randomly
3 $bracket\_counter$ = 0
4 **while** *termination_condition* **do**
5 **for** $iteration \in \{0, 1, ..., s_{max}\}$ **do**
6 $budgets, n\_configs$ = SH_bracket_under_HB(b_{min}, b_{max}, η, $iteration$)
7 **for** $i \in \{0, 1, ..., s_{max} - iterations\}$ **do**
8 **for** $j \in \{1, 2, ..., n\_configs[i]\}$ **do**
9 target = rolling pointer for DE[$budgets[i]$]
10 $mutation\_types$ = "vanilla" **if** i is 0 **else** "altered"
11 **if** $bracket\_counter$ *is* 0 *and* $i > 0$ **then**
12 $config$ = j-th best config from DE[$budgets[i-1]$]
13 **else**
14 $config$ = DE_trial_generation($target$, $mutation\_type$)
15 **end**
16 $result$ = Evaluate $config$ on $budgets[i]$
17 DE selection using $result$, $config$ vs. $target$
18 Update incumbent, $config_{inc}$
19 **end**
20 **end**
21 **end**
22 $bracket\_counter$ += 1
23 **end**
24 **return** $config_{inc}$

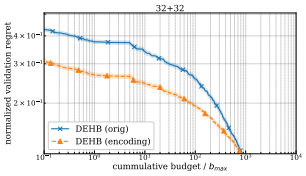

Figure 3: Comparing DEHB encodings for the Stochastic Counting Ones problem in 64 dimensional space with 32 categorical and 32 continuous hyperparameters. Results for all algorithms on 50 runs.

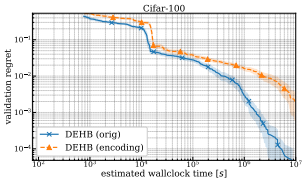

Figure 4: Comparing DEHB encodings for the Cifar-100 dataset from NAS-Bench-201's 6-dimensional space. Results for all algorithms on 50 runs.

C.2 Handling Mixed Data Types

When dealing with discrete or categorical search spaces, such as the NAS problem, the best way to apply DE with such parameters is to keep the population continuous and perform mutation and crossover normally (Eq. 2, 3); then, to evaluate a configuration we evaluate a copy of it in the original discrete space as we explain below. If we instead dealt with a discrete population, then the diversity of population would drop dramatically, leading to many individuals having the same parameter values; the resulting population would then have many duplicates, lowering the diversity of the difference distribution and making it hard for DE to explore effectively. We designed DEHB to scale all parameters of a configuration in a population to a unit hypercube $[0, 1]$, for the two broad types of parameters normally encountered:

- *Integer* and *float* parameters, $X^i \in [a_i, b_i]$ are retrieved as: $a_i + (b_i - a_i) \cdot U_{i,g}$, where the integer parameters are additionally rounded.
- *Ordinal* and *categorical* parameters, $X^i \in \{x_1, ..., x_n\}$, are treated equivalently s.t. the range [0, 1] is divided uniformly into n bins.

We also experimented with another encoding design where each category in each of the categorical variables are represented as a continuous variables $[0, 1]$ and the variable with the *max* over the continuous variables is chosen as the category [Vallati *et al.*, 2015]. For example, in Figure 3, the effective dimensionality of the search space will become 96-dimensional — 32 continuous variables + 64 continuous variables derived from 32 binary variables. We choose a representative set of benchmarks (NAS-Bench-201 and Counting Ones) to compare DEHB with the two encodings mentioned, in Figures 3, 4. It is enough to see one example which performs much worse than the DE-NAS [Awad *et al.*, 2020] encoding we chose for DEHB. The encoding from [Vallati *et al.*, 2015] did not achieve a better final performance than DEHB in any of our experiments.

C.3 Parallel Implementation

The DEHB algorithm is a sequence of DEHB Brackets, which in turn are a fixed sequence of SH brackets. This feature, along with the asynchronous nature of DE allows a parallel execution of DEHB. We dub the main process as the

DEHB Orchestrator which maintains a single copy of all DE subpopulations. An *HB bracket manager* determines which budget to run from which SH bracket. Based on this input from the bracket manager, the orchestrator can fetch a configuration[4] from the current subpopulations and make an asynchronous call for its evaluation on the assigned budget. The rest of the orchestrator continues synchronously to check for free workers, and query the HB bracket manager for the next budget and SH bracket. Once a worker finishes computation, the orchestrator collects the result, performs DE selection and updates the relevant subpopulation accordingly. This form of an update is referred to as *immediate, asynchronous* DE.

DEHB uses a *synchronous* SH routine. Though each of the function evaluations at a particular budget can be distributed, a higher budget needs to wait on all the lower budget evaluations to be finished. A higher budget evaluation can begin only once the lower budget evaluations are over and the top $1/\eta$ can be selected. However, the *asynchronous* nature of DE allows a new bracket to begin if a worker is available while existing SH brackets have pending jobs or are waiting for results. The new bracket can continue using the current state of DE subpopulations maintained by the *DEHB Orchestrator*. Once the pending jobs from previous brackets are over, the DE selection updates the *DEHB Orchestrator's* subpopulations. Thus, the utilisation of available computational resources is maximized while the central copy of subpopulations maintained by the *Orchestrator* ensures that each new SH bracket spawned works with the latest updated subpopulation.

D More details on Experiments

D.1 Baseline Algorithms

In all our experiments we keep the configuration of all the algorithms the same. These settings are well-performing setting that have been benchmarked in previous works — [Falkner *et al.*, 2018], [Ying *et al.*, 2019], [Awad *et al.*, 2020].

Random Search (RS) We sample random architectures in the configuration space from a uniform distribution in each generation.

BOHB We used the implementation from https://github.com/automl/HpBandSter. In [Ying *et al.*, 2019], they identified the settings of key hyperparameters as: η is set to 3, the minimum bandwidth for the kernel density estimator is set to 0.3 and bandwidth factor is set to 3. In our experiments, we deploy the same settings.

Hyperband (HB) We used the implementation from https://github.com/automl/HpBandSter. We set $\eta = 3$ and this parameter is not free to change since there is no other different budgets included in the NAS benchmarks.

Tree-structured Parzen estimator (TPE) We used the open-source implementation from https://github.com/hyperopt/hyperopt. We kept the settings of hyperparameters to their default.

Sequential Model-based Algorithm Configuration (SMAC) We used the implementation from https://github.com/automl/SMAC3 under its default parameter setting. Only for the Counting Ones problem with 64-dimensions, the *initial design* had to be changed to a Latin Hypercube design, instead of a Sobol design.

Regularized Evolution (RE) We used the implementation from [Real *et al.*, 2019]. We initially sample an edge or operator uniformly at random, then we perform the mutation. After reaching the population size, RE kills the oldest member at each iteration. As recommended by [Ying *et al.*, 2019], the population size (PS) and sample size (TS) are set to 100 and 10 respectively.

Differential Evolution (DE) We used the implementation from [Awad *et al.*, 2020], keeping the *rand1* strategy for mutation and *binomial crossover* as the crossover strategy. We also use the same population size of 20 as [Awad *et al.*, 2020].

All plots for all baselines were plotted for the incumbent validation regret over the estimated wallclock time, ignoring the optimization time. The x-axis therefore accounts for only the cumulative cost incurred by function evaluations for each algorithm. All algorithms were run for similar *actual* wallclock time budget. Certain algorithms under certain benchmarks may not appear to have equivalent total *estimated* wallclock time. That is an artefact of ignoring optimization time. Model-based algorithms such as SMAC, BOHB, TPE have a computational cost dependent on the observation history. They might undertake lesser number of function evaluations for the same actual wallclock time.

D.2 Artificial Toy Function: Stochastic Counting Ones

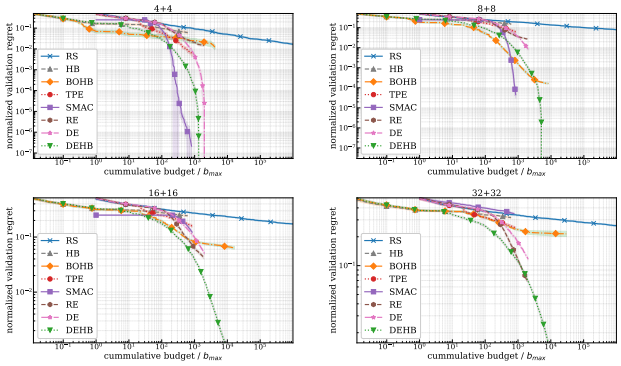

Figure 5: Results for the Stochastic Counting Ones problem for $N = \{4, 8, 16, 32\}$ respectively indicating N categorical and N continuous hyperparameters for each case. All algorithms shown were run for 50 runs.

The Counting Ones benchmark was designed to minimize the following objective function:

$$f(x) = -\left(\sum_{x_i \in X_{cat}} x_i + \sum_{x_j \in X_{cont}} \mathbb{E}_b[(B_{p=x_j})]\right),$$

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

[4] DE mutation and crossover to generate configuration

here represents the number of samples used to estimate the mean of the Bernoulli distribution (B) with parameters x_j.

The experiments on the Stochastic Counting Ones benchmark used $N = \{4, 8, 16, 32\}$, all of which are shown in Figure 5. For the low dimensional cases, BOHB and SMAC's models are able to give them an early advantage. For this toy benchmark the global optima is located at the corner of a unit hypercube. Random samples can span the lower dimensional space adequately for a model to improve the search rapidly. DEHB on the other hand may require a few extra function evaluations to reach similar convergence. However, this conservative approach aids DEHB for the high-dimensional cases where it is able to converge much more rapidly in comparison to other algorithms. Especially where SMAC and BOHB's convergence worsens significantly. DEHB thus showcases its robust performance even when the dimensionality of the problem increases exponentially.

D.3 Feed-forward networks on OpenML datasets

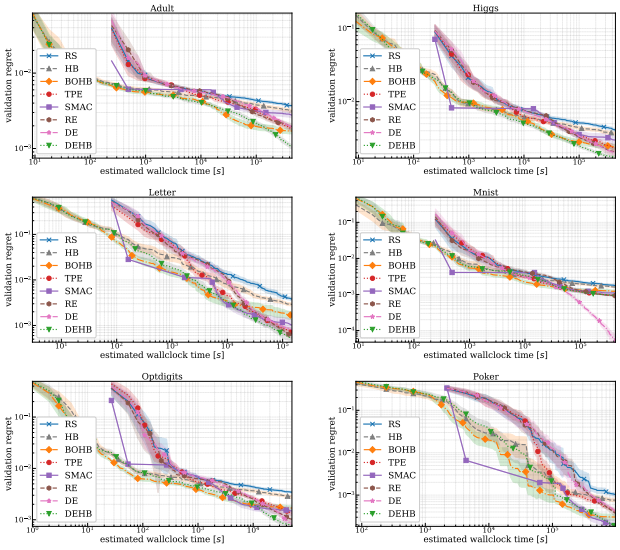

Figure 6: Results for the OpenML surrogate benchmark for the 6 datasets: Adult, Higgs, Letter, MNIST, Optdigits, Poker. The search space had 6 continuous hyperparameters. All plots shown were averaged over 50 runs of each algorithm.

Figure 6, show the results on all 6 datasets from OpenML surrogates benchmark — Adult, Letter, Higgs, MNIST, Optdigits, Poker. The surrogate model space is just 6-dimensional, allowing BOHB and TPE to build more confident models and be well-performing algorithms in this space, especially early in the optimization. However, DE and DEHB are able to remain competitive and consistently achieve an improved final performance than TPE and BOHB respectively. While even TPE achieves a better final performance than BOHB. Overall, DEHB is a competetive *anytime* performer for this benchmark with the most robust final performances.

D.4 Bayesian Neural Networks

The search space for the two-layer fully-connected Bayesian Neural Network is defined by 5 hyperparameters which are: the step length, the length of the burn-in period, the number of units in each layer, and the decay parameter of the momentum variable. In Figure 7, we show the results for the tuning of Bayesian Neural Networks on both the Boston Housing and Protein Structure datasets for the 6-dimensional Bayesian Neural Networks benchmark. We observe that SMAC, TPE and BOHB are able to build models and reach similar regions of performance with high confidence. DEHB is slower to match in such a low-dimensional noisy space. However, given the same cumulative budget, DEHB achieves a competitive final score.

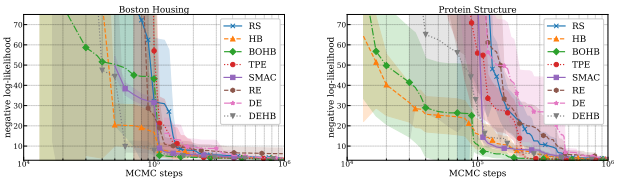

Figure 7: Results for tuning 5 hyperparameters of a Bayesian Neural Network on the the Boston Housing and Protein Structure datasets respectively, for 50 runs of each algorithm.

D.5 Reinforcement Learning

For this benchamrk, the proximal policy optimization (PPO) [Schulman *et al.*, 2017] implementation is parameterized with 7 hyperparameters: # units layer 1, # units layer 2, batch size, learning rate, discount, likelihood ratio clipping and entropy regularization. Figure 8 summarises the performance of all algorithms on the RL problem for the Cartpole benchmark. SMAC uses a SOBOL grid as its initial design and both its benefit and drawback can be seen as SMAC rapidly improves, stalls, and then improves again once model-based search begins. However, BOHB and DEHB both remain competitive and BOHB, DEHB, SMAC emerge as the top-3 for this benchmark, achieving similar final scores. We notice that the DE trace stands out as worse than RS and will explain the reason behind this. Given the late improvement for DE $pop = 20$, we posit that this is a result of the *deferred* updates of DE based on the classical DE [Awad *et al.*, 2020] update design and also the design of the benchmark.

For classical-DE, the updates are *deferred*, that is the results of the *selection* process are incorporated into the population for consideration in the next evolution step, only after *all* the individuals of the population have undergone evolution. In terms of computation, the wall-clock time for *population size* number of function evaluations are accumulated, before the population is updated. In Figure 8 we illustrate why given how this benchmark is designed, this minor detail for DE slows down convergence. Along with a DE of population size 20 as used in the experiments, we compare a DE of population size 10 in Figure 8. For the Reinforcement Learning benchmark from [Falkner *et al.*, 2018], each full budget function evaluation consists of 9 trials of a maximum of 3000 episodes. With a population of 20, DE will not

inject a new individual into a population unless all 20 individuals have participated as a parent in the crossover operation. This accumulates wallclock time equivalent to 20 individuals $times$ 9 trials $times$ time taken for a maximum of 3000 episodes. Which can explain the flat trajectories in the optimization trace for DE $pop = 20$ in Figure 8 (right). DE $pop = 10$ slashes this accumulated wallclock time in half and is able to inject newer configurations into the population *faster* and is able to search faster. Given enough runtime, we expect DE $pop = 20$ to converge to similar final scores. DEHB uses the $immediate$ update design for DE, wherein it updates the population immediately after a DE selection, and not wait for the entire population to evolve. We posit that this feature, along with lower fidelity search, and performing grid search over population sizes with Hyperband, enables DEHB to be more practical than classical-DE.

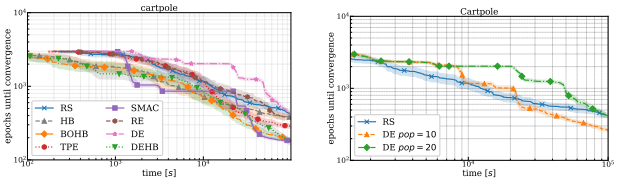

Figure 8: (left) Results for tuning PPO on OpenAI Gym cartpole environment with 7 hyperparameters. Each algorithm shown was run for 50 runs. (right) Same experiment to compare DE with a population size of 10 and 20.

D.6 NAS benchmarks

NAS-Bench-101

This benchmark was the first NAS benchmark relying on tabular lookup that was introduced to encourage research and reproducibility [Ying *et al.*, 2019]. Each architecture from the search space is represented as a stack of cells. Each cell is treated as a directed acyclic graph (DAG) and the nodes and edges of these DAGs are parameterized which serve as the hyperparameters specifying a neural network architecture. NAS-Bench-101 offers a large search space of nearly 423k unique architectures that are trained on Cifar-10. The benchmark also offers a fidelity level — training epoch length — which allows HB, BOHB, and DEHB, to run on this benchmark. We run experiments on all 3 variants provided by NAS-101: Cifar *A*, Cifar *B*, Cifar *C*. The primary search space discussed by [Ying *et al.*, 2019] is Cifar *A*; Cifar *B* and Cifar *C* are variants of the same search space with alternative encodings that deal with the hyperparameters defined on the edges of the DAG as categorical or continuous.

In the NAS-Bench-101 benchmark, the correlation between the performance scores and the different budgets are small [Ying *et al.*, 2019], and therefore BOHB and DEHB do not yield better performance than the methods using full function evaluations only. All 3 evolutionary algorithms tested are able to exploit the discrete high-dimensional space much better than model-based methods such as BOHB and TPE, as seen by the performances of DE, DEHB and RE. While DEHB appears to be the algorithm with the best *anytime* performance in the high-dimensional discrete NAS space. DE yields the final best performance, closely followed by DEHB.

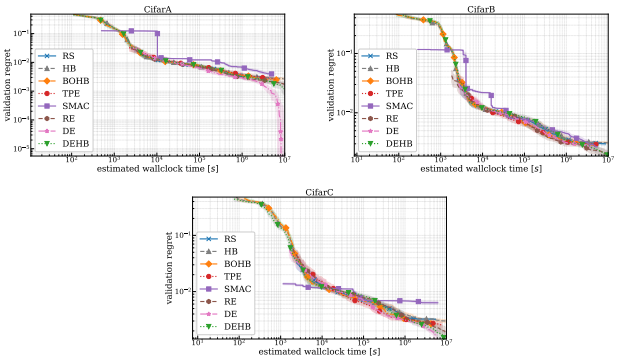

Figure 9: Results for Cifar A, B and C from NAS-Bench-101 for 26, 14, 27-dimensional spaces respectively. All algorithms reported for 50 runs.

NAS-Bench-1shot1

NAS-Bench-1shot1 was introduced by [Zela *et al.*, 2020], as a benchmark derived from the large space of architectures offered by NAS-Bench-101. This benchmark allows the use of modern *one-shot*[5] NAS methods with weight sharing ([Pham *et al.*, 2018], [Liu *et al.*, 2018]). The search space in NAS-Bench-1shot1 was modified to accommodate one-shot methods by keeping the macro network-level topology of the architectures similar and offering a different encoding design for the cell-level topology. This resulted in three search spaces: search space 1, search space 2 and search space 3 with 6240, 29160, and 363648 architectures respectively. In Figure 10, we show the results on all 3 search spaces. We exclude weight sharing methods from the algorithms compared, in order to maintain parity across all experiments, while focusing on the objective of comparing black-box solvers.

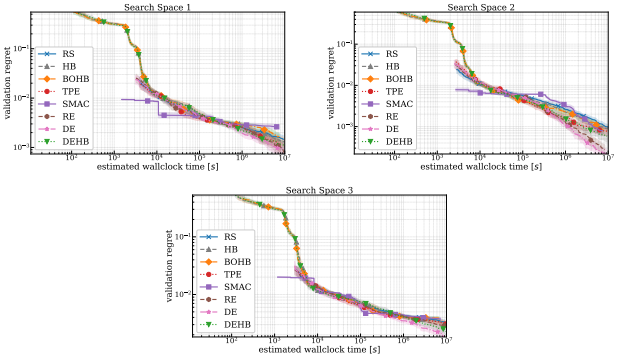

Figure 10: Results for the 3 search spaces from NAS-Bench-1shot1 for 50 runs of each algorithm. The 3 search spaces contains 9, 9, 11 categorical parameters respectively.

From among RS, TPE, SMAC, RE and DE — the full budget algorithms — only DE is able to improve significantly as

[5] training a single large architecture that contains all possible architectures in the search space

optimization proceeds. For the multi-fidelity algorithms — HB, BOHB and DEHB — only DEHB is able to improve and diverge away from HB by the end of optimization. BOHB, HB, RS, TPE and RE all appear to follow a similar trace showing the difficulty of finding good architectures in this benchmark. Nevertheless, the DE-based family of algorithms is able to further exploit the search space and show better performance than the other algorithms. Though DE performs the best, RE remains competitive, again suggesting the power of evolutionary methods on discrete spaces. Among model-based methods, only TPE competes with DEHB.

NAS-Bench-201

To alleviate issues of direct applicability of weight sharing algorithms to NAS-Bench-101, [Dong and Yang, 2020] proposed NAS-Bench-201. This benchmark contains a *fixed cell search space* having DAGs with 4 nodes as the cell structure, and the edges of the DAG cells representing operations. The search space by design contains 6 discrete/categorical hyperparameters. NAS-Bench-201 provides a lookup table for 3 datasets: Cifar-10, Cifar-100 and ImageNet16-120, along with a fidelity level as number of training epochs. The search space for all 3 datasets include 15,625 cells/architectures. From the validation regret performances in Figure 11, it is clear that DEHB quickly converges to strong solutions which are a few orders of magnitude better than BOHB and RS (in terms of regret). DE and RE are both competitive with RE converging slightly faster than DE. Notably, DEHB is the only multi-fidelity algorithm in this experiment that works well.

NAS-Bench-201 specifies the same 6-dimensional discrete hyperparameter space for the Cifar-10 and Cifar-100 datasets. Figure 11 again shows that the evolutionary algorithms perform the best in a space defined by categorical parameters. SMAC in this scenario is the *best*-of-the-rest, outside of DEHB, RE and DE. BOHB evidently struggles to be even significantly better than HB.

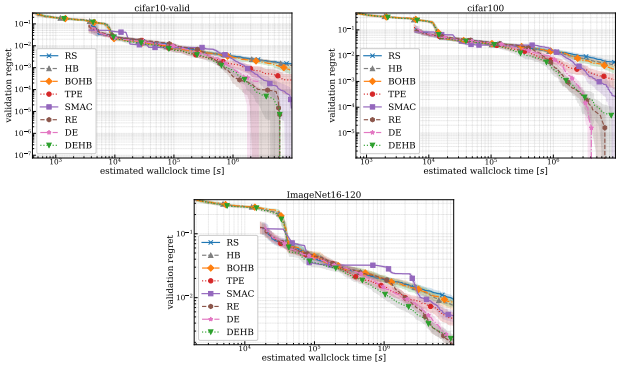

Figure 11: Results for Cifar-10, Cifar-100, ImageNet16-120 from NAS-Bench-201 for 50 runs of each algorithm. The search space contains 6 categorical parameters.

NAS-HPO-Bench

To facilitate HPO research involving feed-forward neural networks, [Klein and Hutter, 2019] introduced NAS-HPO-Bench with a search space composed of hyperparameters that parameterize the architecture of a 2-layer feed-forward network[6], along with hyperparameters for its training procedure. The primary difference between NAS-HPO from the OpenML surrogates benchmark is that in the latter, a random forest model was used as a surrogate to approximate the performance for configurations. NAS-HPO-Bench is designed in the same vein as the other NAS benchmarks discussed earlier. For the total of 9 discrete hyperparameters (4 for architecture + 5 for training), all 62208 configurations resulting from a grid search over the search space were evaluated to yield a tabular representation for configuration and performance mapping. The benchmark provides such lookup tables for 4 popular UCI regression datasets: *Protein Structure*, *Slice Localization*, *Naval Propulsion* and *Parkinsons Telemonitoring*. NAS-HPO-Bench also provides the number of training epochs as a fidelity level.

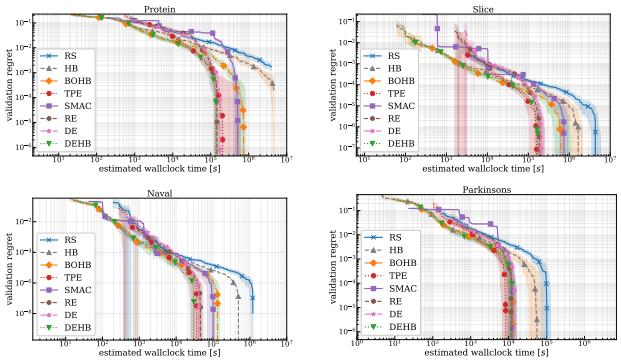

Figure 12: Results for the Protein Structure, Slice Localization, Naval Propulsions, Parkinsons Telemonitoring datasets from NAS-HPO-Bench for 50 runs of each algorithm. The search space contains 9 hyperparameters.

Figure 12 illustrates the performance of all algorithms on the 4 datasets provided in NAS-HPO-Bench. As it appears, barring RS and HB, all other algorithms are able to obtain similar final scores for the benchmark with respect to the validation set. BOHB and DEHB both diverge from HB and start improving early on. However, DEHB continues to improve and is able to converge the fastest. TPE, RE, DE all compete with each other in terms of convergence rate, while BOHB and SMAC show similar convergence speeds.

D.7 Comparison of DEHB to BO-based multi-fidelity methods

BOHB [Falkner *et al.*, 2018] showed that its KDE based BO outperformed other GP-based BO methods. Hence, BOHB was treated as the primary challenger to DEHB as a robust, general multi-fidelity based HPO solver. In this section we run experiments on the benchmarks detailed in the previous sections, to compare DEHB to another popular multi-fidelity BO optimizer, Dragonfly [Kandasamy *et al.*, 2020] (in addition to BOHB). Dragonfly implements BOCA (Kandasamy *et al.* [2017]) which performs BO with low-cost approximations of function evaluations on fidelities treated as a continuous

[6] additionally, a linear output layer

domain. However, this GP-based BO method had longer execution time compared to other algorithms for the tabular/surrogate benchmarks. In Figure 13 we therefore show average of 32 runs for each algorithm, while having to terminate runs earlier than other algorithms for certain cases. In this experiment, we optimize the median performance of a configuration over different seeds. We observe that Dragonfly shows a high variance in performance across benchmarks whereas DEHB is consistently the best or at worse, comparable to Dragonfly. Moreover, BOHB performs clearly better than Dragonfly in 8 out of the 16 cases shown in Figure 13, while being comparable to Dragonfly in at least 4 other benchmarks. Dragonfly comes out as the best optimizer only for the Cifar10 dataset in the NAS-201 benchmark in Figure 13. These experiments however, further illustrate the practicality, robustness, and generality of DEHB compared to GP-based multi-fidelity BO methods.

D.8 Results summary

In the previous experiments sections, results on all the benchmarks for DEHB and all other baselines were reported demonstrating the competitive and often superior *anytime* performance of DEHB. In Table 1, we report the mean final validation regret achieved by all algorithms across all the 26 benchmarks. DEHB got the *best* performance in nearly $1/3$-rd of the benchmarks while reporting the *second*-best performance in over $1/4$-th of all the benchmarks. The last row of Table 1 shows the rank of each algorithm averaged across their final performances on each benchmark. DEHB clearly is the *best* performing algorithm on the whole, followed by DE, which powers DEHB under the hood. Such rankings illustrate DEHB's robustness across different search spaces, including high dimensions, discrete or mixed type spaces, and even problems where response signals from lower fidelity subspaces may not be too informative. It must be noted that for all the different problems tested for with the collection of benchmarks, DEHB is never consistently outperformed by any multi-fidelity or full-budget algorithm.

It must be noted that based on the average rank plot in Figure 13, BOHB appears to be better than DEHB in the *early middle* section of the optimization. The underlying model-based search in BOHB can possibly explain this phenomenon. Though DEHB's underlying DE requires more function evaluations to explore the space, it remains competitive with BOHB and the latter is not significantly better across any of the used benchmarks. Moreover, as Figure 13 indicates, BOHB's relative performance worsens while DEHB continues to remain better even in comparison to the other full-budget black box optimizers such as DE, RE and TPE. BOHB's model-based search can again be attributed for this phenomenon. Many of the benchmarks used are high-dimensional and have mixed data types, which can affect BOHB's model certainty over the configuration space and require much more observations than DEHB requires. Overall, DEHB shows consistently good *anytime* performance with strong *final* performance scores too. As detailed earlier, DEHB's efficiency, simplicity and its speed allow the good use of available resources and make it a good practical and reliable tool for HPO in practice.

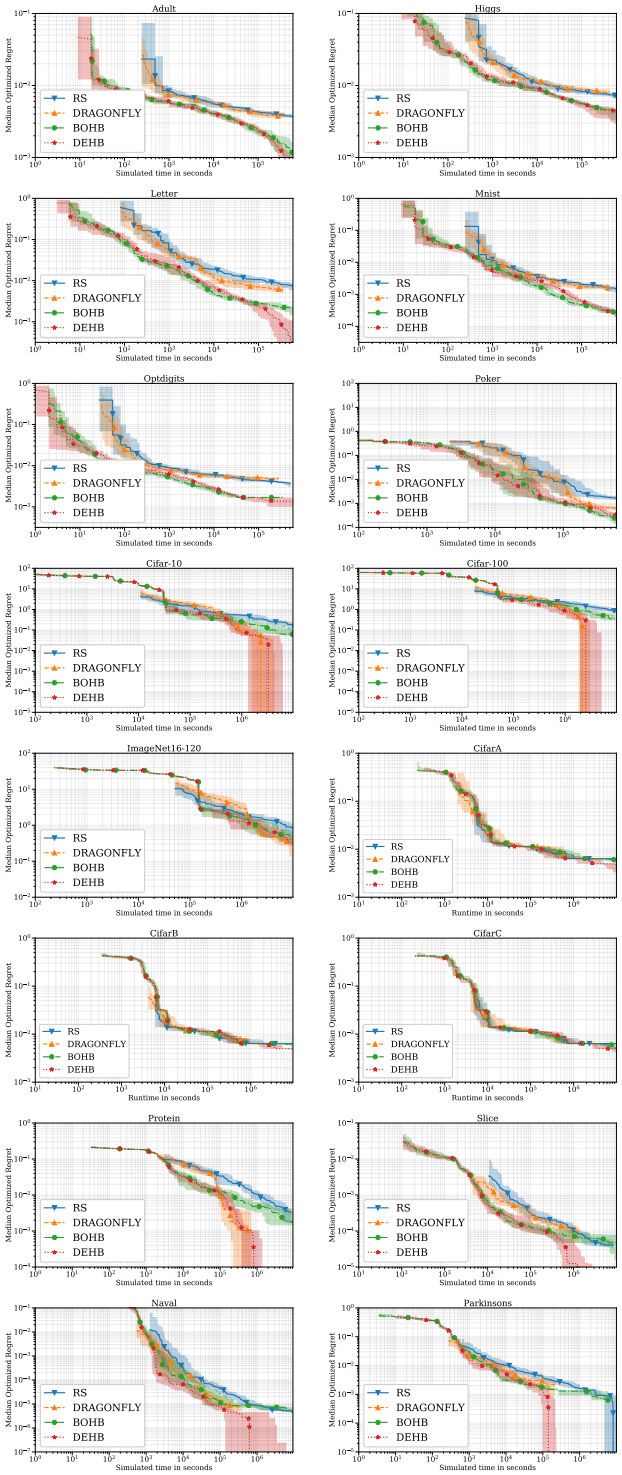

Figure 13: Comparison of GP-based multi-fidelity BO (Dragonfly), KDE based multi-fidelity BO (BOHB), DE based multi-fidelity (DEHB) methods for some of the benchmarks, averaged over 32 runs of each algorithm.

	RS	HB	BOHB	TPE	SMAC	RE	DE	DEHB
$Counting$ $4+4$	9.8e-2 ± 2.6e-2	6.1e-2 ± 1.9e-2	2.1e-2 ± 1.8e-2	8.1e-3 ± 4.8e-3	**1.1e-6** ± 3.9e-6	2.7e-2 ± 7.9e-3	1.5e-2 ± 6.6e-3	9.7e-4 ± 4.6e-4 (2)
$Counting$ $8+8$	1.9e-1 ± 2.5e-2	1.5e-1 ± 2.5e-2	3.9e-3 ± 1.1e-3	7.6e-2 ± 3.1e-2	**2.1e-3** ± 2.3e-3	5.0e-2 ± 1.2e-2	6.6e-2 ± 1.5e-2	1.4e-2 ± 3.7e-3 (3)
$Counting$ $16+16$	2.8e-1 ± 2.2e-2	2.4e-1 ± 2.2e-2	9.6e-2 ± 8.3e-3	1.7e-1 ± 2.8e-2	1.6e-1 ± 1.6e-2	9.2e-2 ± 1.5e-2	1.4e-1 ± 1.7e-2	**6.5e-2** ± 6.3e-3
$Counting$ $32+32$	3.5e-1 ± 1.4e-2	3.2e-1 ± 1.8e-2	2.4e-1 ± 1.3e-2	2.6e-1 ± 1.9e-2	3.6e-1 ± 2.e-2	1.6e-1 ± 2.e-2	2.2e-1 ± 1.8e-2	**1.4e-1** ± 1.1e-2
$OpenML$ $adult$	3.8e-3 ± 4.5e-4	3.1e-3 ± 5.8e-4	1.8e-3 ± 6.1e-4	1.9e-3 ± 5.e-4	2.8e-3 ± 8.3e-4	2.e-3 ± 6.3e-4	1.9e-3 ± 3.4e-4	**1.1e-3** ± 2.0e-4
$OpenML$ $higgs$	4.1e-3 ± 7.3e-4	3.6e-3 ± 5.5e-4	2.5e-3 ± 5.8e-4	2.3e-3 ± 7.4e-4	3.e-3 ± 1.3e-3	2.1e-3 ± 9.4e-4	2.0e-3 ± 4.6e-4	**1.8e-3** ± 2.1e-4
$OpenML$ $letter$	3.8e-3 ± 1.1e-3	2.9e-3 ± 7.5e-4	2.e-3 ± 1.5e-3	7.4e-4 ± 2.8e-4	1.0e-3 ± 9.9e-4	6.2e-4 ± 3.3e-4	8.2e-4 ± 1.5e-4	**5.5e-4** ± 3.4e-4
$OpenML$ $mnist$	1.8e-3 ± 3.1e-4	1.6e-3 ± 2.0e-4	1.3e-3 ± 4.3e-4	9.4e-4 ± 5.2e-5	1.1e-3 ± 4.6e-4	9.2e-4 ± 7.1e-5	**5.3e-5** ± 7.2e-5	9.5e-4 ± 7.6e-5 (4)
$OpenML$ $optdigits$	3.2e-3 ± 5.4e-4	2.8e-3 ± 5.3e-4	1.7e-3 ± 8.1e-4	1.4e-3 ± 4.3e-4	1.5e-3 ± 8.8e-4	1.0e-3 ± 5.5e-4	8.1e-4 ± 3.8e-4	**7.9e-4** ± 2.5e-4
$OpenML$ $poker$	1.1e-3 ± 3.1e-4	7.6e-4 ± 1.8e-4	3.e-4 ± 1.6e-4	4.4e-4 ± 1.1e-4	2.1e-4 ± 1.7e-4	4.e-4 ± 1.9e-4	4.4e-4 ± 1.6e-4	**1.9e-4** ± 5.4e-5
BNN $Boston$	4.7 ± 9.4e-1	4.3 ± 7.e-1	**3.8** ± 3.1e-1	4.0 ± 4.2e-1	4.4 ± 5.5e-1	6.6 ± 9.5e+0	5.0 ± 2.8e+0	4.0 ± 4.9e-1 (2)
BNN $Protein$	4.0 ± 9.6e-1	3.7 ± 5.2e-1	**3.3** ± 1.8e-1	3.4 ± 2.7e-1	3.3 ± 2.2e-1	5.1 ± 5.3e+0	4.9 ± 2.6e+0	3.5 ± 3.7e-1 (4)
$Cartpole$ (RL)	4.7e+2 ± 1.4e+2	3.9e+2 ± 1.1e+2	1.9e+2 ± 4.4e+1	2.9e+2 ± 6.7e+1	**1.8e+2** ± 1.9e+1	3.8e+2 ± 1.3e+2	4.9e+2 ± 1.1e+2	2.e+2 ± 7.3e+1 (3)
$NAS101$ $CifarA$	2.9e-3 ± 7.6e-4	3.e-3 ± 6.0e-4	2.6e-3 ± 1.2e-3	2.8e-3 ± 1.1e-3	4.0e-3 ± 1.5e-3	2.3e-3 ± 1.2e-3	**1.2e-3** ± 1.1e-3	2.2e-3 ± 1.4e-3 (2)
$NAS101$ $CifarB$	3.1e-3 ± 5.3e-4	3.2e-3 ± 3.8e-4	2.8e-3 ± 7.4e-4	3.e-3 ± 5.8e-4	2.9e-3 ± 1.4e-3	**2.4e-3** ± 1.0e-3	2.6e-3 ± 6.9e-4	2.6e-3 ± 1.1e-3 (2)
$NAS101$ $CifarC$	3.2e-3 ± 3.5e-4	3.1e-3 ± 5.4e-4	2.6e-3 ± 7.6e-4	2.7e-3 ± 8.3e-4	6.4e-3 ± 1.3e-3	2.3e-3 ± 1.4e-3	**1.7e-3** ± 1.1e-3	2.0e-3 ± 1.2e-3 (2)
$NAS1s1$ $SS1$	1.6e-3 ± 8.6e-4	1.5e-3 ± 9.6e-4	1.7e-3 ± 1.1e-3	1.3e-3 ± 1.1e-3	2.7e-3 ± 9.9e-4	1.1e-3 ± 1.1e-3	**9.4e-4** ± 9.1e-4	1.4e-3 ± 7.8e-4 (4)
$NAS1s1$ $SS2$	1.3e-3 ± 6.4e-4	9.8e-4 ± 4.8e-4	8.6e-4 ± 5.0e-4	8.2e-4 ± 7.e-4	7.2e-4 ± 4.3e-4	3.e-4 ± 3.e-4	**2.3e-4** ± 2.5e-4	6.4e-4 ± 5.5e-4 (3)
$NAS1s1$ $SS3$	3.5e-3 ± 9.3e-4	3.4e-3 ± 9.2e-4	3.9e-3 ± 7.2e-4	3.5e-3 ± 8.9e-4	3.8e-3 ± 8.6e-4	2.8e-3 ± 1.3e-3	**2.3e-3** ± 1.0e-3	2.6e-3 ± 1.1e-3 (2)
$NAS201$ $Cifar10$	2.7e-3 ± 1.1e-3	2.3e-3 ± 7.6e-4	2.0e-3 ± 1.4e-3	7.2e-4 ± 1.3e-3	4.1e-4 ± 5.8e-4	1.0e-4 ± 5.6e-4	2.3e-4 ± 1.2e-3	**7.8e-5** ± 1.7e-4
$NAS201$ $Cifar100$	8.1e-3 ± 3.5e-3	6.1e-3 ± 3.2e-3	5.7e-3 ± 4.0e-3	1.9e-3 ± 2.8e-3	1.3e-3 ± 2.3e-3	8.e-5 ± 2.4e-4	**0.e+0** ± 0.e+0	1.3e-4 ± 2.9e-4 (3)
$NAS201$ $ImageNet$	9.3e-3 ± 3.6e-3	7.9e-3 ± 3.9e-3	7.3e-3 ± 4.1e-3	4.8e-3 ± 3.7e-3	5.4e-3 ± 3.6e-3	**2.0e-3** ± 1.4e-3	2.3e-3 ± 8.8e-4	2.2e-3 ± 1.6e-3 (2)
$NASHPO$ $Protein$	7.4e-3 ± 4.5e-3	4.2e-3 ± 2.7e-3	2.9e-4 ± 1.1e-3	-4.7e-4 ± 1.9e-3	3.9e-4 ± 2.5e-3	-1.1e-3 ± 3.5e-4	**-1.1e-3** ± 3.1e-4	-1.0e-3 ± 5.9e-4 (3)
$NASHPO$ $Slice$	2.8e-5 ± 3.6e-5	2.9e-6 ± 2.2e-5	-1.0e-5 ± 2.7e-5	-3.1e-5 ± 1.6e-5	-1.9e-5 ± 1.9e-5	-3.5e-5 ± 1.2e-5	**-4.3e-5** ± 7.8e-6	-2.3e-5 ± 1.6e-5 (4)
$NASHPO$ $Naval$	6.8e-6 ± 6.2e-6	2.5e-6 ± 4.3e-6	-2.5e-6 ± 2.8e-6	-4.8e-6 ± 3.3e-6	-6.e-6 ± 2.6e-6	-6.7e-6 ± 1.0e-6	**-7.e-6** ± 8.6e-7	-6.e-6 ± 2.3e-6 (4)
$NASHPO$ $Parkinsons$	-6.6e-4 ± 1.1e-3	-1.0e-3 ± 1.0e-3	**-3.4e-3** ± 7.4e-4	-2.3e-3 ± 1.1e-3	-2.6e-3 ± 8.4e-4	-2.9e-3 ± 7.5e-4	-3.2e-3 ± 6.8e-4	-2.4e-3 ± 8.8e-4 (5)
$Avg.\ rank$	**7.46**	**6.54**	**4.42**	**4.35**	**4.73**	**3.16**	**2.96**	**2.39**

Table 1: Final mean validation regret ± standard deviation for 50 runs all algorithms tested for all benchmarks. Performance scores for DEHB is accompanied with its (rank) among other algorithms. The last row shows the *average relative rank* of each algorithm based on their final performance on each benchmark.

E Ablation Studies

DEHB was designed as an easy-to-use tool for HPO and NAS. This necessitated that DEHB contains as few hyperparameters as possible that require tuning or that which makes DEHB sensitive to them. Given that the HB parameters inherent to DEHB are contingent on the problem being solved, that leaves only the DE components' hyperparameters to be set adequately. We perform ablation of the mutation and crossover rates to observe how it fairs for DEHB's design for the suite of benchmarks we experiment on.

E.1 Varying F

The crossover probability p was fixed at 0.5, while mutation factor F was varied with the values $0.1, 0.3, 0.5, 0.7, 0.9$. The studies are carried out on NAS-Bench-101, OpenML Surrogate and the toy Stochastic Counting Ones benchmarks. The results are reported in Figure 14.

We observe that a low F of 0.1 allows more exploitative power to the DE search for a well correlated benchmark such as Counting Ones, while $F = 0.9$ performs the worst. However, for the other benchmarks $F = 0.1$ performs the worst with all other F performing similarly. As a result we choose the conservative option of $F = 0.5$ to ensure one general set-

ting performs acceptably across all benchmarks.

E.2 Varying CR

The scaling factor F was fixed at 0.5, while crossover factor p was varied with the values $0.1, 0.3, 0.5, 0.7, 0.9$. The studies were carried out on NAS-Bench-101, OpenML Surrogate and the toy Stochastic Counting Ones benchmarks. The results are reported in Figure 15.

The lower the p value, the less likely are the random mutant traits to be incorporated into the population. For Counting Ones, we observe that a high p slows down convergence, whereas low p speeds up convergence. However, for the others, $p = 0.5$ is consistently the best performer. Hence, we chose $p = 0.5$ for the design of DEHB.

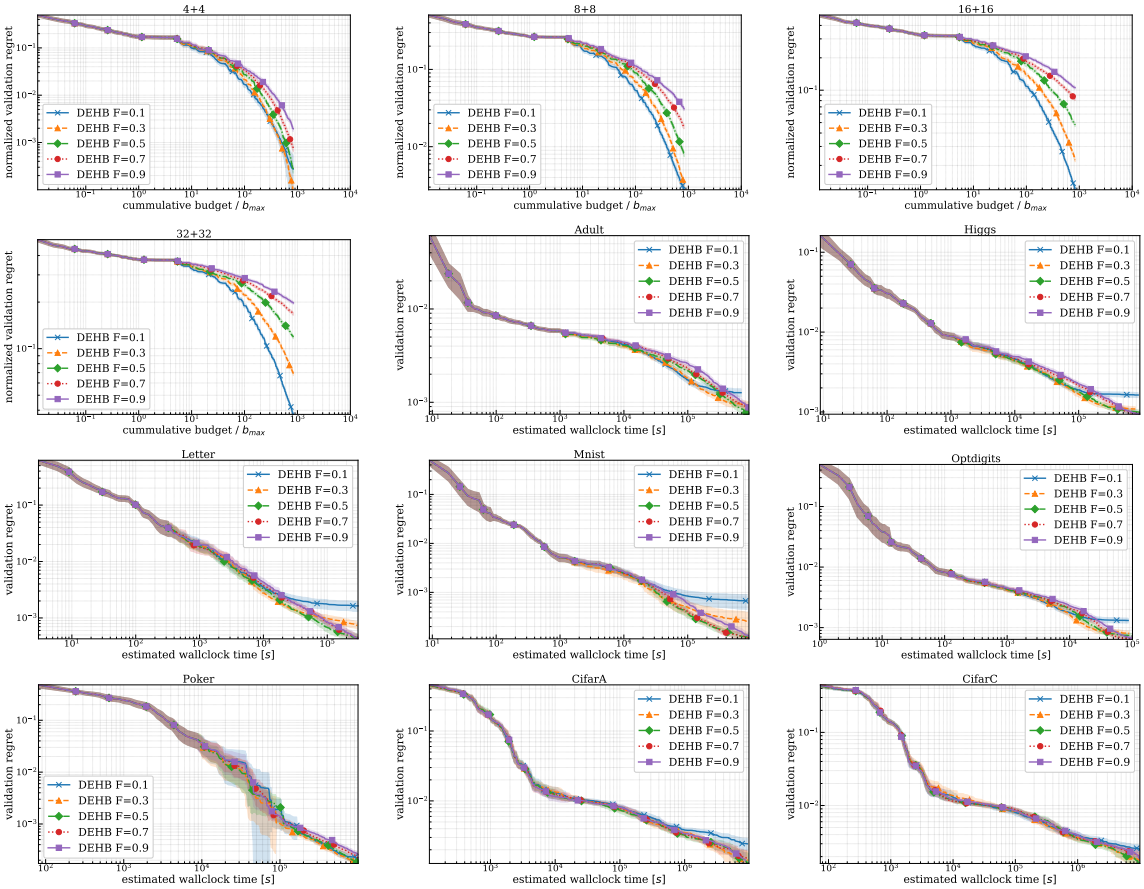

Figure 14: Ablation study for *mutation factor* F for DEHB, with *crossover probability* fixed at $p = 0.5$

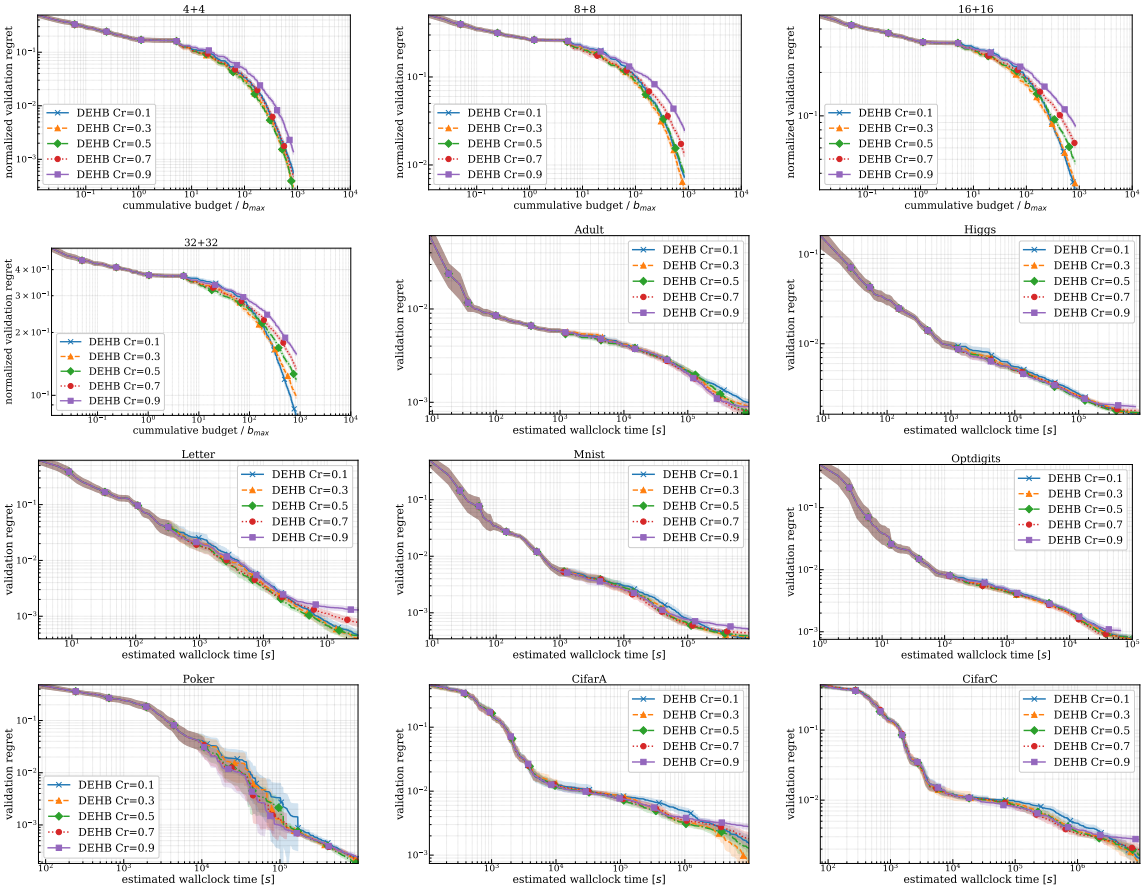

Figure 15: Ablation study for *crossover probability* p for DEHB, with *mutation factor* fixed at $F = 0.5$

References

N. Awad, N. Mallik, and F. Hutter. Differential evolution for neural architecture search. In *First ICLR Workshop on Neural Architecture Search*, 2020.

U. K. Chakraborty. *Advances in differential evolution*, volume 143. Springer, 2008.

S. Das, S. S. Mullick, and P. N. Suganthan. Recent advances in differential evolution–an updated survey. *Swarm and Evolutionary Computation*, 27:1–30, 2016.

X. Dong and Y. Yang. Nas-bench-102: Extending the scope of reproducible neural architecture search. *arXiv preprint arXiv:2001.00326*, 2020.

S. Falkner, A. Klein, and F. Hutter. BOHB: Robust and efficient hyperparameter optimization at scale. In *Proc. of ICML'18*, pages 1437–1446, 2018.

R. M. Storn K. Price and J. A. Lampinen. *Differential evolution: a practical approach to global optimization*. Springer Science & Business Media, 2006.

K. Kandasamy, G. Dasarathy, J. Schneider, and B. Póczos. Multi-fidelity bayesian optimisation with continuous approximations. *arXiv:1703.06240 [stat.ML]*, 2017.

K. Kandasamy, K. R. Vysyaraju, W. Neiswanger, B. Paria, C. R. Collins, J. Schneider, B. Poczos, and E. P. Xing. Tuning hyperparameters without grad students: Scalable and robust bayesian optimisation with dragonfly. *Journal of Machine Learning Research*, 21(81):1–27, 2020.

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

L. Li, K. Jamieson, G. DeSalvo, A. Rostamizadeh, and A. Talwalkar. Hyperband: Bandit-based configuration evaluation for hyperparameter optimization. In *Proc. of ICLR'17*, 2017.

H. Liu, K. Simonyan, and Y. Yang. Darts: Differentiable architecture search. *arXiv preprint arXiv:1806.09055*, 2018.

H. Pham, M. Y. Guan, B. Zoph, Q. V. Le, and J. Dean. Efficient neural architecture search via parameter sharing. *arXiv preprint arXiv:1802.03268*, 2018.

E. Real, A. Aggarwal, Y. Huang, and Q. V. Le. Regularized evolution for image classifier architecture search. In *Proc. of AAAI*, volume 33, pages 4780–4789, 2019.

J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

R. Storn and K. Price. Differential evolution–a simple and efficient heuristic for global optimization over continuous spaces. *Journal of global optimization*, 11(4):341–359, 1997.

M. Vallati, F. Hutter, Lukás L. Chrpa, and T. L. McCluskey. On the effective configuration of planning domain models. In *Twenty-Fourth International Joint Conference on Artificial Intelligence*, 2015.

C. Ying, A. Klein, E. Real, E. Christiansen, K. Murphy, and F. Hutter. Nas-bench-101: Towards reproducible neural architecture search. *arXiv preprint arXiv:1902.09635*, 2019.

A. Zela, J. Siems, and F. Hutter. Nas-bench-1shot1: Benchmarking and dissecting one-shot neural architecture search. *arXiv preprint arXiv:2001.10422*, 2020.