A Maintenance

In this section we present a maintenance plan that is adapted from the datasheets for datasets [92].

- **Who is maintaining the benchmarking library?** *HPOBench* is developed and maintained by the Machine Learning Lab at the University of Freiburg.
- **How can the maintainer of the dataset be contacted(e.g., email address)?** Questions should be submitted via an issue on the Github repository at `https://github.com/automl/HPOBench`.
- **Is there an erratum?** No.
- **Will the benchmarking library be updated?** We consider adding new benchmarking problems and potentially fix existing issues with existing benchmarks. Such changes will be communicated via release notes in Github releases.
- **Will older versions of the benchmarking library continue to be supported/hosted/maintained?** Older versions of the benchmarking code are available via the underlying git repository. Containers are versioned and available via Gitlab. We aim to answer questions on a best-effort basis, but will not do so for older versions of the benchmarking library.
- **If others want to extend/augment/build on/contribute to the dataset, is there a mechanism for them to do so?** We allow contributions from the community via a process that is currently described at `https://github.com/automl/HPOBench/wiki/How-to-add-a-new-benchmark-step-by-step`.
- **Any other comments?** No.

B Benchmarking efforts

In addition to Section 3 of the main paper, we provide here a non-exhaustive list of further benchmarking libraries in the area of HPO consisting of not only a publication but which constitute or constituted a long-running effort to compare methods:

- HPOLib [6] to benchmark global optimization methods
- ACLib [47] to benchmark algorithm configuration methods
- OpenAI Gym [93] to benchmark RL methods
- COCO [9] to compare continuous optimization methods
- Bayesmark [8] to benchmark Bayesian optimization methods
- OpenML benchmarking suites [94] provide a set of datasets for supervised classification
- Olympus [12] provides a set of experiment planning tasks to evaluate optimization algorithms
- HPO-B [48] provides a tabular benchmark to compare black-box HPO methods
- ExpoBench [11] provides expensive benchmark problems for HPO

C Benchmarking competitions

In addition to Section 3 of the main paper, we provide here a non-exhaustive list of benchmarking competitions on HPO and related topics:

- The AutoML challenges [49]
- The AutoDL challenge [50]
- NeurIPS 2020 Black-Box optimization challenge [51]
- The KDD cup (see `https://www.kdd.org/kdd-cup`)
- Challenges in Machine Learning (CIML) workshop series (see `https://ciml.chalearn.org/`)
- Black-box Optimization Benchmarking (BBOB) workshop series [95] (see `https://numbbo.github.io/workshops/`)

D More Details on Considered Benchmarks

In addition to the main paper, here we provide further details on our benchmarks collected. We start with issues we faced during collection and then briefly describe the existing community benchmarks (Section D.2) and the new benchmarks (Section D.3).

D.1 Conflicting Dependencies.

During benchmark collection, we also encountered a few examples of conflicting dependencies and updated interfaces making long-term maintenance of non-containerized benchmarks hard: *Net* [22] was built with the latest version of scikit-learn [72] (0.18) when it was developed but is incompatible with the current version (0.24); the *Cartpole* benchmark does not run with the latest version of TensorForce [96] due to a change in the API; *NB201* [70] changed its interface as well as the underlying data from its initial release. Additionally, in total, none of the *existing community* benchmarks we collected for this paper had a full list of dependencies given.

D.2 Existing Community Benchmarks

Here, we provide more details on the *existing community* benchmarks currently in *HPOBench* and list their hyperparameter and fidelity spaces in Table 4.

Cartpole [22] A highly stochastic benchmark having 7 hyperparameters of the *proximal policy optimization* [97] algorithm implemented in TensorForce [96] for the *cartpole swing-up* task implemented in the OpenAI Gym [93]. The number of repetitions is used as the fidelity and this benchmark is available only as a *raw* benchmark.

BNN [22] The Bayesian neural network benchmark is a 4-hyperparameter tuning task to minimize the negative log-likelihood of a Bayesian neural network trained with stochastic gradient Hamilton Monte-Carlo [98] with scale adaption [99] on two different regression datasets from the UCI repository ([100], Protein Structure and YearPredictionMSD). It is implemented with Lasagne [101] and Theano [102]. It uses the number of MCMC sampling steps and is available only as a *raw* benchmark.

Net [22] This benchmark has 6 architectural and training hyperparameters to train a feed-forward neural network on six different datasets from OpenML [75]: Adult, Higgs, Letter, MNIST, Optdigits and Poker. As fidelity it uses the number of training epochs for the neural networks. This is a surrogate benchmark and uses a random forest, which is trained on 10K randomly samples configurations.

NBHPO. [69] This benchmark is a joint neural architecture search and HPO for a 2-layer feedforward neural network. The output layer was designed as a linear layer with parameterized architecture details and training parameters while the search space is a large grid of configurations on four popular UCI datasets for regression: protein structure, slice localization, naval propulsion and parkinsons telemonitoring.

NB101. [54] This was the first introduced NAS benchmark based on tabular lookup, designed for reproducibility in NAS research. Each architecture is represented as a stack of architectural cells, where each such cell is represented as directed acyclic graphs (DAGs). The benchmarks offers a search space that includes nearly 423k unique architectures by parameterizing the nodes and edges of the DAGs. The lookup table allows to query performance of architectures on the Cifar-10 dataset. Additionally, queries can be made for intermediate training epochs too, thereby allowing multi-fidelity optimization. In contrast to the original implementation, we always return the average across the three repetitions as a score.

NB1Shot1. [71] The NAS-Bench-1shot1 was derived from the large architecture space of NAS-Bench-101, such that, weight-sharing based one-shot NAS methods can be applied for this tabular lookup. The cell-level encoding was modified to yield 3 variants of the architecture space which contains around 6k (search space 1), 29k (search space 2), 300k (search space 3) architectures. In contrast to the original implementation we always return the average across the three repetitions as a score.

NB201. [70] To further aid the use of weight sharing algorithms to NAS Benchmarks, this benchmark introduced a fixed cell search space wherein a DAG has only 4 nodes that define the cell architecture. Whereas the edges define the operations. Thus, creating a search space of around 15k unique

Table 4: Hyperparameter spaces of our benchmarks. For each benchmark, we report the hyperparameter names, type, whether they are on a log scale, and their respective range for each benchmark. Additionally, we report the same information for the fidelity space. If the spaces are different for different benchmarks within one family, we report them separately.

benchmark	name	type	log	range
Cartpole	batch_size	int	✓	$[8, 256]$
	discount	float	✗	$[0.0, 1.0]$
	entropy_regularization	float	✗	$[0.0, 1.0]$
	learning_rate	float	✓	$[1e^{-07}, 0.1]$
	likelihood_ratio_clipping	float	✗	$[1e^{-7}, 1.0]$
	n_units_{1,2}*	int	✓	$[8, 128]$
	repetitions	int	✗	$[1, 9]$
BNN	burn_in	float	✗	$[0.0, 0.8]$
	l_rate	float	✓	$[1e^{-6}, 0.1]$
	mdecay	float	✗	$[0.0, 1.0]$
	n_units_{1,2}*	int	✓	$[16, 512]$
	epochs	int	✗	$[500, 10000]$
Net	average_units_per_layer_log2	float	✗	[4.0, 8.0]
	batch_size_log2	float	✗	$[3.0, 8.0]$
	dropout	float	✗	$[0.0, 0.5]$
	final_lr_fraction_log2	float	✗	$[-4.0, 0.0]$
	initial_lr_log10	float	✗	$[-6.0, -2.0]$
	num_layers	int	✗	$[1, 5]$
adult, higgs, mnist	epochs	int	✗	$[9, 243]$
letter		int	✗	$[3, 81]$
optdigits		int	✗	$[1, 27]$
poker		int	✗	$[81, 2187]$
NBHPO	activation_fn_{1, 2}*	cat	-	{tanh, relu}
	batch_size	ord	-	{8, 16, 32, 64}
	dropout_{1, 2}*	ord	-	{0.0, 0.3, 0.6}
	init_lr	ord	-	{0.0005, 0.001, 0.005, 0.01, 0.05, 0.1}
	lr_schedule	cat	-	{cosine, const}
	n_units_{1, 2}*	ord	-	{16, 32, 64, 128, 256, 512}
	epochs	int	✗	[3, 100]
NB201	1<-0	cat	-	{none, skip_connect,
	2<-{0,1}*	cat	-	nor_conv_1x1, nor_conv_3x3,
	3<-{0,1,2}*	cat	-	avg_pool_3x3}
	epochs	int	✗	[12, 200]
NB101$_{Cf10A}$	edge_{0, 1, ..., 20}*	cat	-	{0, 1}
	op_node_{0, 1, .., 4}*	cat	-	{conv1x1-bn-relu, conv3x3-bn-relu, maxpool3x3}
	epochs	ord	✗	{[4, 12, 36, 108}
NB101$_{Cf10B}$	edge_{0, 1, ..., 8}*	cat	-	{0, 1, 2, ..., 20}
	op_node_{0, 1, ..., 4}*	cat	-	{conv1x1-bn-relu, conv3x3-bn-relu, maxpool3x3}
	epochs	ord	✗	{4, 12, 36, 108}
NB101$_{Cf10C}$	edge_{0, 1, ..., 20}*	float	✗	[0.0, 1.0]
	num_edges	int	✗	[0, 9]
	op_node_{0, 1, ..., 4}*	cat	-	{conv1x1-bn-relu, conv3x3-bn-relu, maxpool3x3}
	epochs	ord	✗	{4, 12, 36, 108}

Table 5: Table 4 continued

NB1Shot1$_1$	choice_block_{1,2,3,4}_op*	cat	-	{conv1x1-bn-relu, conv3x3-bn-relu, maxpool3x3}
	choice_block_1_parents	cat	-	{(0,)}
	choice_block_2_parents	cat	-	{(0,1)}
	choice_block_3_parents	cat	-	{(0,1), (0,2), (1,2)}
	choice_block_4_parents	cat	-	{(0,1), (0,2), (0,3), (1,2), (1,3), (2,3)}
	choice_block_5_parents	cat	-	{(0,1), (0,2), (0,3), (0,4), (1,2), (1,3), (2,3), (1,4), (2,3), (2,4), (3,4)}
	epochs	ord	✗	{4, 12, 36, 108}
NB1Shot1$_2$	choice_block_{1,2,3,4}_op*	cat	-	{conv1x1-bn-relu, conv3x3-bn-relu, maxpool3x3}
	choice_block_1_parents	cat	-	{(0,)}
	choice_block_2_parents	cat	-	{(0,), (1,)}
	choice_block_3_parents	cat	-	{(0, 1), (0, 2), (1, 2)}
	choice_block_4_parents	cat	-	{(0, 1), (0, 2), (0, 3), (1, 2), (1, 3), (2, 3)}
	choice_block_5_parents	cat	-	{(0, 1, 2), (0, 1, 3), (0, 1, 4), (0, 2, 3), (0, 2, 4), (0, 3, 4), (1, 2, 3), (1, 2, 4), (1, 3, 4), (2, 3, 4)}
	epochs	ord	✗	{4, 12, 36, 108}
NB1Shot1$_3$	choice_block_{1,2,3,4,5}_op*	cat	-	{conv1x1-bn-relu, conv3x3-bn-relu, maxpool3x3}
	choice_block_1_parents	cat	-	(0,)
	choice_block_2_parents	cat	-	{(0,), (1,)}
	choice_block_3_parents	cat	-	{(0,), (1,), (2,)}
	choice_block_4_parents	cat	-	{(0, 1), (0, 2), (0, 3), (1, 2), (1, 3), (2, 3)}
	choice_block_5_parents	cat	-	{(0, 1), (0, 2), (0, 3), (0, 4), (1, 2), (1, 3), (1, 4), (2, 3), (2, 4), (3, 4)}
	choice_block_6_parents	cat	-	{(0, 1), (0, 2), (0, 3), (0, 4), (0, 5), (1, 2), (1, 3), (1, 4), (1, 5), (2, 3), (2, 4), (2, 5), (3, 4), (3, 5), (4, 5)}
	epochs	ord	✗	{4, 12, 36, 108}

architectures. NAS-Bench-201 provides a lookup table for Cifar-10, Cifar-100, and ImageNet16-120. In contrast to the original implementation we always return the average across the three repetitions as a score.

D.3 New Benchmarks

Here, we provide more details on the *new* benchmarks and list their hyperparameter and fidelity spaces in Table 6.

SVM A 2-dimensional benchmark for a SVM model with an RBF kernel with the *regularization* and the kernel coefficient *gamma* as available hyperparameters to tune. It uses the dataset subset fraction as the fidelity and is available as both *raw* and *tabular* benchmarks. For the tabular version, we discretized each hyperparameter into 21 bins for 441 unique hyperparameter configurations and evaluated each of these on 20 datasets from the *AutoML* benchmark [74].

LogReg This benchmark has 2 hyperparameters – learning rate and regularization for a logistic regression model trained using Stochastic Gradient Descent (SGD). It uses dataset fraction and/or the number of SGD iterations as the fidelity and is available as both a *raw* and *tabular* benchmark. For the tabular version we evaluated a grid of 625 configurations on 20 datasets from the *AutoML* benchmark [74].

XGBoost This benchmark has 4 hyperparameters that tune the maximum depth per tree, the features subsampled per tree, the learning rate and the L2 regularization for the XGBoost model. It uses dataset fraction and/or the number of boosting iterations as fidelities and is available as both a *raw* and *tabular* benchmark. For the tabular version we discretized each hyperparameter into 10 bins and evaluated the resulting grid of $10k$ configurations on 20 datasets from the *AutoML* benchmark [74].

RandomForest This benchmark has 4 hyperparameters that tune the maximum depth per tree, the maximum features subsampled per split, the minimum number of samples required for splitting a node, and the minimum number of samples required in each leaf node for a random forest model. It uses dataset fraction and/or the number of trees as fidelities and is available as both a *raw* and *tabular* benchmark. For the tabular version we discretized each hyperparameter into 10 bins and evaluated the resulting grid of $10k$ configurations on 20 datasets from the *AutoML* benchmark [74].

MLP This benchmark has 5 hyperparameters – two hyperparameters that determine the depth and width of the network; three more hyperparameters tune the batch size, L2 regularization and the initial learning rate for Adam. It uses dataset fraction and/or the number of epochs as fidelities and is available as both a *raw* and *tabular* benchmark. For the tabular version, we discretized each hyperparameter into 10 bins and evaluated the resulting grid of $1k$ configurations for each of 30 different architectures, resulting in $30k$ configurations in total, on 8 datasets from the *AutoML* benchmark [74].

To collect the data for the tabular benchmark, we evaluated every configuration-fidelity pair in the discretized space on 5 different seeds; each such repetition is evaluated on the following 4 metrics: *accuracy*, *balanced accuracy*, *precision*, *f1*.

Table 6: Table detailing the configuration spaces for the *new* benchmarks included in *HPOBench*. For each model, we report the hyperparameters and their ranges (top part) and fidelities and their ranges (bottom part).

benchmark	name	type	log	range
SVM	C	float	✓	$[2^{-10}, 2^{10}]$
	gamma	float	✓	$[2^{-10}, 2^{10}]$
	subsample	float	✗	$[0.1, 1.0]$
LogReg	alpha	float	✓	$[1e-05, 1.0]$
	eta0	float	✓	$[1e-05, 1.0]$
	iter	int	✗	$[10, 1000]$
	subsample	float	✗	$[0.1, 1.0]$
XGBoost	colsample_bytree	float	✗	$[0.1, 1.0]$
	eta	float	✓	$[2^{-10}, 1.0]$
	max_depth	int	✓	$[1, 50]$
	reg_lambda	float	✓	$[2^{-10}, 2^{10}]$
	n_estimators	int	✗	$[50, 2000]$
	subsample	float	✗	$[0.1, 1.0]$
RandomForest	max_depth	int	✓	$[1, 50]$
	max_features	float	✗	$[0.0, 1.0]$
	min_samples_leaf	int	✗	$[1, 2]$
	min_samples_split	int	✓	$[2, 128]$
	n_estimators	int	✗	$[16, 512]$
	subsample	float	✗	$[0.1, 1.0]$
MLP	alpha	float	✓	$[1.0e^{-08}, 1.0]$
	batch_size	int	✓	$[4, 256]$
	depth	int	✗	$[1, 3]$
	learning_rate_init	float	✓	$[1.0e^{-05}, 1.0]$
	width	int	✓	$[16, 1024]$
	epochs	int	✗	$[3, 243]$
	subsample	float	✗	$[0.1, 1]$

Table 7: OpenML Task IDs used from the *AutoML* benchmark for *SVM*, *LogReg*, *XGBoost* and *RandomForest*. *MLP* uses only the first 8 task IDs. The table shows the total number of instances available (train + test) (#obs), and the total number of features prior to preprocessing (#feat).

name	tid	#obs	#feat
blood-transf..	10101	748	4
vehicle	53	846	18
Australian	146818	690	14
car	146821	1728	6
phoneme	9952	5404	5
segment	146822	2310	19
credit-g	31	1000	20
kc1	3917	2109	22
sylvine	168912	5124	20
kr-vs-kp	3	3196	36
jungle_che..	167119	44819	6
mfeat-factors	12	2000	216
shuttle	146212	58000	9
jasmine	168911	2984	145
cnae-9	9981	1080	856
numerai28.6	167120	96320	21
bank-mark..	14965	45211	16
higgs	146606	98050	28
adult	7592	48842	14
nomao	9977	34465	118

E Details on Hardware Used for Experiments

For our benchmark study we ran all jobs on a compute cluster equipped with Intel(R) Xeon(R) Gold 6242 CPU @ 2.80GHz. If not stated otherwise, we run all job on 1 CPU with up to 6GB RAM for at

most 4 days or till the benchmark budget was exhausted. For runs that needed more memory to load data, we allowed up to 12GB RAM (*NB101*, *NB1Shot1*, *NB201*). For collecting tabular data for the *new* benchmarks, we ran all jobs on a compute cluster equipped with Intel(R) Broadwell E5-2630v4 @ 2.2GHz with up to 6GB RAM.

F Details on Runtime

Running all optimizers on the raw versions of the existing community benchmarks would take more than 1500 CPU years, but the use of tabular and surrogate-based benchmarks in *HPOBench* reduces this amount to *only* 22.5 CPU years. While this is still a lot, we emphasize that most of this time is used by the optimizers (and not the benchmarks). For developing and evaluating a new multi-fidelity method and comparing it to computationally cheap baselines, e.g. sequentially evaluating both *RS* and *DE* on all tabular and surrogate benchmarks took < 10 CPU days, *HB* took around 50 CPU days and *DEHB* needed around 40 CPU days. To further explain the amount of time it took to obtain results for our empirical study, we look at statistics of our runs. In Table 8, we report the average runtime (in hours, maximum 96, however, we only record the last call to our objective function, so a runtime of, e.g. 95 could also mean that the optimizer did not call the objective function for 2 hours and was then forcefully terminated) and the number of calls$/100$ to the objective function for one exemplary benchmark per family. The last two rows show the total time spent on obtaining results for all raw benchmarks and surrogate plus tabular benchmarks per optimizers. Additionally, we give the overall amount of compute spent on our empirical study.

Looking at the first part of the table, we favourably see, that most optimizers on average took less than two hours to spend the simulated optimization budget. However, there are some exceptions like BO_{GP} and *DF* mostly hitting the optimization budget of 4 days resulting in fewer calls to the objective function and worse performance.

Additionally, these statistics also allow to study some failure cases of the optimizers. For *DF* on *NB1Shot1*, it only evaluated 90 configurations while taking less than 1 hour. Here *DF* stopped right after the initial design, because it could not construct a model, the same happened for the *BNN* benchmarks and thus the total runtime for the raw benchmarks is substantially lower. Finally, *RS* called $NB101_{Cf10A}$ three times more often than other black-box optimizers, because the table underlying this benchmark does not cover the complete hyperparameter space and thus returns a loss of 1 and costs of 0 for configurations not in the table. More advanced search algorithms avoid these seemingly badly performing regions and thus sample more *costly* evaluations.

Table 8: We report the median wallclock time (in hours) and number of calls$/100$ to the objective function for all optimizers and one benchmark per benchmark family.

	Net_{Adult}		$NBHPO_{Slice}$		$NB101_{Cf10A}$		$NB201_{Cf100}$		$NB1Shot1_1$		total time	
optimizer	t	#c	t	#c	t	#c	t	#c	t	#c	raw	tab+sur
RS	0	25	0	47	0	118	0	9	0	23	2295	102
DE	0	25	0	29	0	31	0	7	0	13	2290	48
BO_{KDE}	0	25	1	31	0	31	0	8	0	27	2297	716
BO_{GP}	82	13	96	12	96	7	9	7	96	12	2296	46566
BO_{RF}	2	25	2	43	2	39	0	8	1	21	2299	7326
HEBO	84	13	96	12	96	11	26	7	96	12	2300	48735
HB	0	108	3	209	1	242	0	21	0	95	2300	1299
BOHB	0	108	2	130	0	104	0	20	1	110	2298	1508
DEHB	0	108	0	126	0	118	0	17	0	63	2255	1031
SMAC-HB	8	105	8	202	8	166	0	20	2	96	2298	12370
DF	93	10	92	9	90	6	94	10	0	1	532	41867
$Optuna^{hb}_{tpe}$	0	109	0	109	0	75	0	31	0	55	2297	1606
$Optuna^{md}_{tpe}$	4	287	0	111	0	81	0	15	0	78	2278	5694
									sum in CPU years		3.2	19.3

G More Details on Considered Optimizers

Here, we provide additional details on the optimizers used in this work. We provide an overview in Table 9 and then briefly explain our baselines, black-box and multi-fidelity optimizers in detail. We note that we used the default settings for all tools and implementations.

Table 9: Overview of HPO optimizers considered in this study. For each optimizer we list the model type, what types of hyperparameters and fidelities it can handle (the tool can either handle it natively (✓), not handle it (✗) or we could transform the type ((✓))), a link to the codebase and references.

name	model	types			fidelities		link	reference	version
		cont	cat	log	disc.	cont.			
RS	-	✓	✓	✓	✗	✗	-	[81]	
BO_{GP}	GP	✓	✓	✓	✗	✗	SMAC3	[76, 78]	1.0.1
BO_{RF}	RF	✓	✓	✓	✗	✗	SMAC3	[76, 78]	1.0.1
BO_{KDE}	KDE	✓	✓	✓	✗	✗	HpBandSter	[22]	0.7.4
DE	-	✓	✓	✓	✗	✗	DEHB	[24]	git commit
HEBO	GP	✓	✓	✓	✗	✗	HEBO	[77]	0.1.0
HB	-	✓	✓	✓	(✓)	✓	HpBandSter	[19]	0.7.4
BOHB	KDE	✓	✓	✓	(✓)	✓	HpBandSter	[22]	0.7.4
DEHB	-	✓	✓	✓	(✓)	✓	DEHB	[5]	git commit
SMAC-HB	RF	✓	✓	✓	(✓)	✓	SMAC3	[76, 78]	1.0.1
DF	GP	✓	✓	(✓)	✓	✓	Dragonfly	[79]	0.1.5
$Optuna^{md}_{tpe}$	TPE	✓	✓	✓	✓	✗	Optuna	[80]	2.8.0
$Optuna^{hb}_{tpe}$	TPE	✓	✓	✓	✓	✗	Optuna	[80]	2.8.0

G.1 Baselines

Random Search (*RS*) is a simple baseline that samples new configurations uniformly at random from a prior. It was proposed as an improved baseline over grid search [81] as it can handle low intrinsic dimensionality and is easier to run in parallel.

Hyperband (*HB*) [19] is a bandit algorithm for the pure-exploration, non-stochastic infinite-armed bandit problem which we described in Section 2. We will use it as a random search baseline for multi-fidelity optimization.

G.2 Black-box Optimizers

BO$_{GP}$ is an implementation of traditional Gaussian process-based *BO* with a Matérn kernel [52] and a SOBOL sequence initial design [103]. For categorical hyperparameters it uses a Hamming kernel [104] and is implemented in the SMAC toolbox [78], thus it is using local search for acquisition function optimization [76]. Its hyperparameters were tuned for good average performance over 50 function evaluations using meta-optimization [85].

BO$_{RF}$ is similar to *BO*$_{GP}$ but uses random forests as suggested in the original *SMAC* publication [76]. In contrast to the original hyperparameter setting of *SMAC* with random forests, this version uses a SOBOL sequence initial design [103] and only 20% interleaved random samples instead of 50%. These hyperparameter settings were found via meta-optimization [85] for good average performance over 50 function evaluations.

BO$_{KDE}$ is a re-implementation of the TPE algorithm using multi-dimensional kernel density estimators as used by the *BOHB* algorithm [22]. Instead of modeling the objective function as $p(y|x)$, it models two densities, $p(x|y_{good})$ and $p(x|y_{bad})$, and uses their ratio that is proportional to the expected improvement acquisition function [14].

DE. We use the canonical DE with *rand/1* as the mutation strategy and *binomial* crossover. We set the mutation factor F and crossover rate CR to 0.5 each and the population size NP to 20 [24].

HEBO is a GP-based BO algorithm that uses input warping and output warping, an ensemble of acquisition functions [77] and won the recent NeurIPS Blackbox Optimization challenge [51].

G.3 Multi-fidelity Optimizers

BOHB [22] combines *BO* and *HB* with the goal of both algorithms complementing each other. It follows the regular *HB* scheme, but instead of sampling configurations at random it uses *BO*. For *BO* it uses a KDE model as described above. To handle multiple fidelities it builds an independent model per fidelity, but only if there is sufficient (number of hyperparameters + 1) training data available, to then always use the model from the highest fidelity for which a model is available.

SMAC-HB [78] is a straight-forward re-implementation of the *BOHB* algorithm using the BO_{RF} building blocks described in the previous section.

DEHB [5] is a new model-free successor of *BOHB* which uses the evolutionary optimization method DE instead of BO. For each fidelity, *DEHB* maintains a subpopulation and runs a separate DE evolution while the information about good configurations flows from subpopulations at lower fidelities to those at higher fidelities through a modified mutation strategy. The mutation allows the use of these good configurations from lower fidelities to be selected as parents to evolve the new subpopulation at a higher fidelity. The hyperparameters of the *DE*-part of *DEHB* are set exactly as for *DE* described above.

Dragonfly (*DF*) [79] is a BO algorithm which implements an improved version of the BOCA algorithm [3], which uses Gaussian processes and the upper confidence bound acquisition function to first decide a location to query before deciding the fidelity to query.

Optuna$^{md}_{tpe}$ is implemented in the Optuna framework [80], which is a high level optimization framework that allows to combine sampling (to propose new configurations to evaluate) and pruning (to stop configurations if they are not promising) strategies to construct optimization algorithms. *Optuna*$^{md}_{tpe}$ uses TPE as a sampling algorithm and the median stopping [25] rule as a pruning algorithm. It fits a Gaussian Mixture Model on the best so far seen configurations. The pruner stops a configuration if its best intermediate result is worse compared to the median of the other configurations on the same fidelity level.

Optuna$^{hb}_{tpe}$ is like *Optuna*$^{md}_{tpe}$ implemented in the Optuna framework [80] and uses TPE for sampling, but *HB* as a pruning algorithm.

H More Results

Here, we give more results on our large-scale empirical study. First, we report results for all optimizers in Table 10, 11 for the existing community benchmarks, and in Tables 12- 21 for the new benchmarks. Second, we report statistical tests for RQ1 and RQ2 similar to the ones in the main paper for the new benchmarks in Tables 22 and 23. Third, we report average ranking-over-time for each benchmark family in Figure 5, 6 and 7. Finally, we show performance-over-time plots for all *existing community* benchmarks in Figure 8, 9 and 10.

Table 10: Final performance of each black-box optimizer (lower is better). We report median performance (regret for tabular/surrogate benchmarks and function values for raw benchmarks) across 32 repetitions per *existing community* benchmark. We boldface the best result per row.

benchmark	black-box optimizers					
	RS	*DE*	*BO*$_{GP}$	*BO*$_{RF}$	*BO*$_{KDE}$	*HEBO*
Cartpole	786.444	851.72222	826.444	227.056	381.72222	**191.833**
BNN$_{Protein}$	3.17763	3.05335	3.10514	3.09424	**3.04537**	3.08331
BNN$_{Year}$	4.07933	4.01501	3.97039	3.88006	3.86969	**3.77791**
Net$_{Adult}$	0.00258	0.00072	0.00141	0.00110	0.00068	**0.00006**
Net$_{Higgs}$	0.00390	0.00194	0.00250	0.00277	0.00256	**0.00193**
Net$_{Letter}$	0.00263	**0.00000**	0.00095	0.00055	0.00101	0.00038
Net$_{MNIST}$	0.00097	**0.00014**	0.00040	0.00019	0.00027	0.00015
Net$_{OptDig}$	0.00229	**0.00048**	0.00225	0.00137	0.00129	0.00121
Net$_{Poker}$	0.00099	0.00054	0.00051	0.00024	0.00035	**0.00004**
NBHPO$_{Naval}$	0.00000	**0.00000**	0.00001	0.00000	0.00000	0.00000
NBHPO$_{Park}$	0.00000	**0.00000**	0.00092	0.00000	0.00000	0.00000
NBHPO$_{Prot}$	0.00328	**0.00000**	0.00000	0.00000	0.00104	0.00000
NBHPO$_{Slice}$	0.00004	**0.00000**	0.00006	0.00000	0.00001	0.00000
NB101$_{Cf10A}$	0.00638	**0.00417**	0.00638	0.00497	0.00638	0.00497
NB101$_{Cf10B}$	0.00638	0.00497	0.00638	**0.00454**	0.00603	0.00497
NB101$_{Cf10C}$	0.00638	0.00491	0.00638	0.00638	0.00604	**0.00180**
NB201$_{Cf100}$	0.86667	**0.00000**	**0.00000**	**0.00000**	1.09833	**0.00000**
NB201$_{Cf10V}$	0.16667	**0.00000**	**0.00000**	**0.00000**	0.18667	**0.00000**
NB201$_{INet}$	0.86667	0.45556	0.00000	0.27222	1.21667	**0.00000**
NB1Shot1$_1$	0.00033	0.00060	**0.00000**	0.00000	0.00087	0.00073
NB1Shot1$_2$	0.00107	**0.00000**	0.00000	0.00000	0.00107	0.00160
NB1Shot1$_3$	0.00249	**0.00114**	0.00177	0.00177	0.00307	0.00250

Table 11: Final performance of each multi-fidelity optimizer (lower is better). We report median performance (regret for tabular/surrogate benchmarks and function values for raw benchmarks) across 32 repetitions per *existing community* benchmark. We boldface the best result per row.

benchmark	multi-fidelity optimizers						
	HB	*BOHB*	*DEHB*	*SMAC-HB*	*DF*	*Optuna*$^{md}_{tpe}$	*Optuna*$^{hb}_{tpe}$
Cartpole	724.88889	232.94444	593.83333	**211.33333**	1004.38889	702.33333	523.66667
BNN$_{Protein}$	3.14047	3.03529	3.07514	3.06393	9.65112	**3.03252**	3.08817
BNN$_{Year}$	4.11971	3.92703	4.03676	**3.88357**	12.30007	3.91723	4.02678
Net$_{Adult}$	0.00232	0.00060	0.00062	0.00067	0.00298	0.00067	**0.00059**
Net$_{Higgs}$	0.00373	0.00232	**0.00206**	0.00278	0.00469	0.00212	0.00209
Net$_{Letter}$	0.00197	0.00140	**0.00032**	0.00075	0.00240	0.00147	0.00045
Net$_{MNIST}$	0.00075	0.00032	0.00018	0.00023	0.00117	0.00026	**0.00018**
Net$_{OptDig}$	0.00201	0.00153	0.00101	0.00161	0.00394	0.00153	**0.00056**
Net$_{Poker}$	0.00072	0.00018	0.00031	**0.00008**	0.00053	0.00020	0.00028
NBHPO$_{Naval}$	0.00000	0.00000	**0.00000**	0.00000	0.00001	0.00001	0.00005
NBHPO$_{Park}$	0.00000	0.00000	**0.00000**	0.00000	0.00246	0.00359	0.00149
NBHPO$_{Prot}$	0.00104	0.00414	**0.00000**	0.00000	0.00000	0.00423	0.00162
NBHPO$_{Slice}$	0.00001	0.00001	**0.00000**	0.00000	0.00008	0.00009	0.00004
NB101$_{Cf10A}$	0.00638	0.00619	0.00482	**0.00476**	0.00921	0.00863	0.00638
NB101$_{Cf10B}$	0.00638	0.00497	0.00497	**0.00442**	0.00775	0.00838	0.00608
NB101$_{Cf10C}$	0.00638	0.00497	**0.00486**	0.00638	0.00773	0.00861	0.00638
NB201$_{Cf100}$	0.76000	0.86333	**0.00000**	0.00000	0.00000	0.87333	9.99667
NB201$_{Cf10V}$	0.06267	0.10200	**0.00000**	0.01933	0.00000	0.27267	4.66800
NB201$_{INet}$	0.71111	0.63611	**0.27222**	0.27222	0.28889	0.57222	11.29444
NB1Shot1$_1$	0.00007	0.00154	**0.00000**	0.00040	0.00387	0.00544	0.00224
NB1Shot1$_2$	0.00100	0.00107	**0.00000**	0.00090	0.00569	0.00392	0.00140
NB1Shot1$_3$	0.00210	0.00210	**0.00154**	0.00177	0.00651	0.00651	0.00224

Table 12: Final performance of each black-box optimizers (lower is better). We report the median normalized regret across 32 repetitions for each *new* benchmarks collected for *SVM*. We boldface the best result per row.

optimizer	*RS*	*DE*	BO_{GP}	BO_{RF}	BO_{KDE}	*HEBO*
svm_10101	**0.00e+00**	**0.00e+00**	**0.00e+00**	**0.00e+00**	**0.00e+00**	0.046
svm_53	**0.00e+00**	**0.00e+00**	**0.00e+00**	**0.00e+00**	0.003	0.016
svm_146818	**0.00e+00**	**0.00e+00**	**0.00e+00**	**0.00e+00**	0.01	0.042
svm_146821	0.001	**0.00e+00**	**0.00e+00**	**0.00e+00**	0.001	0.011
svm_9952	0.019	0.019	**0.00e+00**	**0.00e+00**	0.02	**0.00e+00**
svm_146822	0.005	0.005	**0.00e+00**	**0.00e+00**	0.005	0.006
svm_31	1.21e-15	1.21e-15	1.21e-15	**6.06e-16**	1.21e-15	0.199
svm_3917	0.031	0.031	**0.00e+00**	**0.00e+00**	0.038	0.095
svm_168912	0.003	9.12e-04	**0.00e+00**	**0.00e+00**	0.003	**0.00e+00**
svm_3	0.001	9.08e-04	**0.00e+00**	**0.00e+00**	0.002	9.08e-04
svm_167119	0.005	1.76e-04	**0.00e+00**	**0.00e+00**	0.002	**0.00e+00**
svm_12	**3.20e-17**	**3.20e-17**	**3.20e-17**	**3.20e-17**	**3.20e-17**	**3.20e-17**
svm_146212	3.00e-04	**0.00e+00**	**0.00e+00**	**0.00e+00**	2.18e-04	**0.00e+00**
svm_168911	0.009	0.007	**0.00e+00**	**0.00e+00**	0.011	0.007
svm_9981	0.018	0.012	**0.00e+00**	**0.00e+00**	0.045	0.017
svm_167120	**0.842**	0.843	**0.842**	**0.842**	0.846	**0.842**
svm_14965	0.009	0.013	**0.00e+00**	**0.00e+00**	0.009	**0.00e+00**
svm_146606	0.017	**0.00e+00**	**0.00e+00**	**0.00e+00**	0.017	0.001
svm_7592	0.002	0.002	**0.00e+00**	**0.00e+00**	0.006	**0.00e+00**
svm_9977	0.001	0.001	**0.00e+00**	**0.00e+00**	5.12e-04	**0.00e+00**

Table 13: Final performance of each multi-fidelity optimizers (lower is better). We report the median normalized regret across 32 repetitions for each *new* benchmarks collected for *SVM*. We boldface the best result per row. We can not report results for $Optuna^{hb}_{tpe}$, since it does not support fidelity values ≤ 1, which is the case for dataset fractions for the *SVM* benchmark.

optimizer	*HB*	*BOHB*	*DEHB*	*SMAC-HB*	*DF*	$Optuna^{md}_{tpe}$	$Optuna^{hb}_{tpe}$
svm_10101	0.023	**0.00e+00**	0.023	0.023	0.167	0.126	-
svm_53	**0.00e+00**	0.003	**0.00e+00**	**0.00e+00**	0.078	**0.00e+00**	-
svm_146818	**0.00e+00**	0.01	**0.00e+00**	**0.00e+00**	0.049	**0.00e+00**	-
svm_146821	0.001	0.001	**0.00e+00**	**0.00e+00**	0.088	**0.00e+00**	-
svm_9952	0.006	0.019	**0.00e+00**	**0.00e+00**	0.074	0.003	-
svm_146822	0.005	0.005	**0.00e+00**	**0.00e+00**	0.023	**0.00e+00**	-
svm_31	**1.21e-15**	**1.21e-15**	**1.21e-15**	**1.21e-15**	0.324	**1.21e-15**	-
svm_3917	**0.031**	0.046	0.038	**0.031**	0.137	0.046	-
svm_168912	3.04e-04	3.04e-04	**0.00e+00**	**0.00e+00**	3.04e-04	**0.00e+00**	-
svm_3	9.08e-04	0.002	**4.54e-04**	9.08e-04	0.014	0.001	-
svm_167119	1.76e-04	0.001	**0.00e+00**	**0.00e+00**	**0.00e+00**	**0.00e+00**	-
svm_12	**3.20e-17**	**3.20e-17**	**3.20e-17**	**3.20e-17**	0.001	**3.20e-17**	-
svm_146212	2.18e-04	**0.00e+00**	**0.00e+00**	**0.00e+00**	**0.00e+00**	**0.00e+00**	-
svm_168911	0.009	0.012	**0.00e+00**	**0.00e+00**	0.012	**0.00e+00**	-
svm_9981	0.012	0.029	**0.00e+00**	**0.00e+00**	0.204	**0.00e+00**	-
svm_167120	**0.842**	0.845	**0.842**	**0.842**	**0.842**	**0.842**	-
svm_14965	0.004	0.004	**0.00e+00**	**0.00e+00**	0.004	0.004	-
svm_146606	0.003	0.01	0.017	**0.00e+00**	**0.00e+00**	0.003	-
svm_7592	0.002	0.005	0.001	0.001	**0.00e+00**	0.004	-
svm_9977	1.83e-04	3.66e-04	**0.00e+00**	1.83e-04	**0.00e+00**	**0.00e+00**	-

Table 14: Final performance of each black-box optimizers (lower is better). We report the median normalized regret across 32 repetitions for each *new* benchmarks collected for *LogReg*. We boldface the best result per row.

optimizer	*RS*	*DE*	BO_{GP}	BO_{RF}	BO_{KDE}	*HEBO*
lr_10101	**0.00e+00**	**0.00e+00**	**0.00e+00**	**0.00e+00**	**0.00e+00**	0.007
lr_53	0.005	0.003	**0.00e+00**	0.003	0.005	0.005
lr_146818	0.003	**0.00e+00**	**0.00e+00**	**0.00e+00**	0.003	0.007
lr_146821	0.002	0.004	**0.00e+00**	**0.00e+00**	0.004	**0.00e+00**
lr_9952	6.78e-04	3.39e-04	**0.00e+00**	**0.00e+00**	6.78e-04	**0.00e+00**
lr_146822	0.001	0.001	**0.00e+00**	**0.00e+00**	0.001	**0.00e+00**
lr_31	0.033	0.033	**4.58e-16**	**4.58e-16**	0.028	0.011
lr_3917	0.019	0.018	**0.00e+00**	0.018	0.021	0.018
lr_168912	9.47e-04	3.16e-04	**0.00e+00**	1.58e-04	7.89e-04	**0.00e+00**
lr_3	0.001	9.23e-04	**0.00e+00**	4.57e-17	0.001	4.57e-17
lr_167119	9.35e-04	3.94e-04	**0.00e+00**	**0.00e+00**	0.002	**0.00e+00**
lr_12	2.54e-17	2.54e-17	**1.27e-17**	2.54e-17	2.54e-17	2.54e-17
lr_146212	0.01	0.009	**0.007**	**0.007**	0.011	**0.007**
lr_168911	0.002	8.17e-04	**0.00e+00**	**0.00e+00**	0.007	**0.00e+00**
lr_9981	0.002	0.002	**0.00e+00**	**0.00e+00**	0.003	**0.00e+00**
lr_167120	0.002	0.003	**0.00e+00**	**0.00e+00**	0.004	**0.00e+00**
lr_14965	8.04e-04	1.46e-04	**0.00e+00**	**0.00e+00**	0.001	**0.00e+00**
lr_146606	1.42e-04	9.44e-05	**0.00e+00**	**0.00e+00**	3.30e-04	**0.00e+00**
lr_7592	1.52e-04	1.33e-04	**0.00e+00**	**0.00e+00**	1.90e-04	**0.00e+00**
lr_9977	2.83e-04	**0.00e+00**	**0.00e+00**	**0.00e+00**	2.83e-04	**0.00e+00**

Table 15: Final performance of each multi-fidelity optimizers (lower is better). We report the median normalized regret across 32 repetitions for each *new* benchmarks collected for *LogReg*. We boldface the best result per row. We note that there are negative regret values for some cases. For these the optimizer did not evaluate a configuration on the highest fidelity within the given *optimization budget* and the observed final function value was better than the best possible function value on the highest budget (which we used to compute regret).

optimizer	*HB*	*BOHB*	*DEHB*	*SMAC-HB*	*DF*	$Optuna_{tpe}^{md}$	$Optuna_{tpe}^{hb}$
lr_10101	**0.00e+00**	0.007	**0.00e+00**	**0.00e+00**	0.094	**0.00e+00**	**0.00e+00**
lr_53	0.003	0.005	**0.00e+00**	0.003	0.068	**0.00e+00**	**0.00e+00**
lr_146818	**0.003**	0.009	**0.003**	**0.003**	0.06	0.004	**0.003**
lr_146821	0.002	0.009	**0.00e+00**	0.003	0.324	0.003	0.001
lr_9952	0.008	0.009	0.008	**0.004**	0.064	0.013	0.008
lr_146822	0.001	0.002	0.001	0.001	0.033	**9.51e-04**	**9.51e-04**
lr_31	**4.58e-16**	0.011	**4.58e-16**	0.011	0.131	**4.58e-16**	**4.58e-16**
lr_3917	0.018	0.021	**0.00e+00**	0.021	0.092	0.018	0.018
lr_168912	3.16e-04	0.001	**0.00e+00**	3.16e-04	0.015	0.001	4.73e-04
lr_3	**4.57e-17**	0.002	**4.57e-17**	9.23e-04	0.063	9.23e-04	**4.57e-17**
lr_167119	9.84e-05	0.001	**0.00e+00**	9.84e-05	0.034	2.95e-04	4.92e-05
lr_12	**2.54e-17**	6.17e-04	**2.54e-17**	**2.54e-17**	0.015	4.11e-04	**2.54e-17**
lr_146212	**0.01**	0.012	**0.01**	0.011	0.108	**0.01**	0.011
lr_168911	0.002	0.003	**8.17e-04**	0.002	0.062	0.003	0.002
lr_9981	0.003	0.004	**0.002**	**0.002**	0.013	0.004	0.003
lr_167120	0.002	0.004	**0.002**	0.002	0.516	0.003	0.002
lr_14965	0.001	0.002	2.56e-04	**0.00e+00**	0.034	**0.00e+00**	0.001
lr_146606	**0.00e+00**	1.89e-04	**0.00e+00**	9.44e-05	0.206	2.83e-04	**0.00e+00**
lr_7592	**1.14e-04**	5.88e-04	1.90e-04	1.90e-04	0.017	3.03e-04	1.52e-04
lr_9977	1.41e-04	6.72e-04	**0.00e+00**	2.83e-04	0.041	3.18e-04	**0.00e+00**

Table 16: Final performance of each black-box optimizers (lower is better). We report the median normalized regret across 32 repetitions for each *new* benchmarks collected for *XGBoost*. We boldface the best result per row.

optimizer	*RS*	*DE*	BO_{GP}	BO_{RF}	BO_{KDE}	*HEBO*
xgb_10101	0.013	0.013	**0.00e+00**	0.013	0.013	0.013
xgb_53	0.004	0.004	**0.002**	**0.002**	0.004	**0.002**
xgb_146818	**0.00e+00**	**0.00e+00**	0.009	**0.00e+00**	**0.00e+00**	0.005
xgb_146821	0.01	0.01	0.01	0.005	0.009	**0.00e+00**
xgb_9952	**0.006**	0.009	0.008	**0.006**	**0.006**	**0.006**
xgb_146822	0.013	0.015	0.013	0.013	0.013	**0.00e+00**
xgb_31	0.012	0.01	0.008	0.006	0.008	**0.00e+00**
xgb_3917	**0.008**	**0.008**	**0.008**	**0.008**	**0.008**	**0.008**
xgb_168912	0.012	0.011	**0.005**	0.01	**0.005**	**0.005**
xgb_3	0.004	**0.00e+00**	0.002	0.004	0.004	**0.00e+00**
xgb_167119	4.41e-04	0.002	0.001	**0.00e+00**	0.001	**0.00e+00**
xgb_12	0.02	0.02	0.02	0.02	0.02	**1.31e-15**
xgb_146212	**0.00e+00**	**0.00e+00**	**0.00e+00**	**0.00e+00**	**0.00e+00**	**0.00e+00**
xgb_168911	0.01	0.01	0.009	0.01	**0.008**	**0.008**
xgb_9981	0.004	**0.00e+00**	**0.00e+00**	**0.00e+00**	**0.00e+00**	**0.00e+00**
xgb_167120	0.001	0.001	0.001	0.001	0.001	**8.31e-04**
xgb_14965	0.002	0.003	0.002	4.64e-04	0.002	**4.64e-04**
xgb_146606	0.002	0.002	0.002	0.001	0.002	**2.35e-04**
xgb_7592	0.002	0.001	0.001	**0.00e+00**	0.001	**0.00e+00**
xgb_9977	0.006	0.004	0.003	0.004	0.002	**0.002**

Table 17: Final performance of each multi-fidelity optimizers (lower is better). We report the median normalized regret across 32 repetitions for each *new* benchmarks collected for *XGBoost*. We boldface the best result per row. We note that there are negative regret values for some cases. For these the optimizer did not evaluate a configuration on the highest fidelity within the given *optimization budget* and the observed final function value was better than the best possible function value on the highest budget (which we used to compute regret).

optimizer	*HB*	*BOHB*	*DEHB*	*SMAC-HB*	*DF*	$Optuna^{md}_{tpe}$	$Optuna^{hb}_{tpe}$
xgb_10101	**0.013**	**0.013**	**0.013**	**0.013**	0.597	**0.013**	**0.013**
xgb_53	**1.47e-16**	**1.47e-16**	**1.47e-16**	**1.47e-16**	0.053	0.008	0.004
xgb_146818	**0.00e+00**	**0.00e+00**	**0.00e+00**	**0.00e+00**	0.065	**0.00e+00**	**0.00e+00**
xgb_146821	0.01	0.01	0.01	**0.006**	0.027	0.012	0.01
xgb_9952	**0.006**	0.011	0.008	0.008	0.218	0.008	0.008
xgb_146822	0.009	0.009	0.013	**0.00e+00**	0.06	0.013	0.004
xgb_31	0.012	0.012	0.006	0.008	0.07	**0.004**	0.008
xgb_3917	0.012	0.016	0.012	0.012	0.055	0.016	**0.01**
xgb_168912	0.005	0.012	**0.005**	0.005	0.052	0.012	0.011
xgb_3	**0.004**	**0.004**	**0.004**	**0.004**	0.016	**0.004**	**0.004**
xgb_167119	4.41e-04	0.002	4.41e-04	**0.00e+00**	0.51	0.001	0.001
xgb_12	1.31e-15	0.02	0.01	0.02	0.137	0.02	**6.57e-16**
xgb_146212	**0.00e+00**	**0.00e+00**	**0.00e+00**	**0.00e+00**	0.019	**0.00e+00**	**0.00e+00**
xgb_168911	0.007	0.01	0.008	**0.00e+00**	0.054	0.01	**0.00e+00**
xgb_9981	**0.00e+00**	**0.00e+00**	**0.00e+00**	**0.00e+00**	0.042	**0.00e+00**	**0.00e+00**
xgb_167120	0.001	0.001	0.001	**0.001**	0.011	0.001	0.001
xgb_14965	0.003	0.005	0.004	**0.002**	0.071	0.004	0.003
xgb_146606	0.002	0.002	0.002	**2.35e-04**	0.081	0.002	0.001
xgb_7592	0.002	0.002	0.003	**0.001**	0.352	0.003	0.002
xgb_9977	0.006	0.006	**0.005**	**0.005**	0.029	0.007	**0.005**

Table 18: Final performance of each black-box optimizers (lower is better). We report the median normalized regret across 32 repetitions for each *new* benchmarks collected for *RandomForest*. We boldface the best result per row.

optimizer	*RS*	*DE*	BO_{GP}	BO_{RF}	BO_{KDE}	*HEBO*
rf_10101	0.102	0.092	0.049	**0.00e+00**	0.068	**0.00e+00**
rf_53	0.013	0.008	0.003	**0.00e+00**	0.004	**0.00e+00**
rf_146818	0.033	0.019	0.008	**0.00e+00**	0.015	**0.00e+00**
rf_146821	0.032	0.02	0.001	**0.00e+00**	0.007	**0.00e+00**
rf_9952	0.009	0.009	0.003	**0.00e+00**	0.008	**0.00e+00**
rf_146822	0.005	0.006	0.001	**0.00e+00**	0.003	**0.00e+00**
rf_31	0.074	0.066	0.053	**0.007**	0.079	0.01
rf_3917	0.066	0.055	0.024	**4.52e-16**	0.038	**4.52e-16**
rf_168912	0.053	0.08	0.012	0.004	0.014	**0.00e+00**
rf_3	0.006	0.009	5.38e-04	**0.00e+00**	0.002	**0.00e+00**
rf_167119	0.061	0.034	0.003	**0.00e+00**	0.006	**0.00e+00**
rf_12	0.002	0.001	5.83e-04	3.91e-17	0.002	**0.00e+00**
rf_146212	1.09e-04	5.43e-05	1.09e-04	**0.00e+00**	1.09e-04	**0.00e+00**
rf_168911	0.038	0.044	0.005	**0.002**	0.013	**0.002**
rf_9981	0.012	0.019	0.003	**0.00e+00**	0.02	**0.00e+00**
rf_167120	0.003	0.002	0.002	0.002	0.003	**0.001**
rf_14965	0.091	0.084	0.02	9.08e-04	0.008	**0.00e+00**
rf_146606	0.008	0.008	0.001	**0.00e+00**	0.004	**0.00e+00**
rf_7592	0.112	0.106	0.026	**0.00e+00**	0.095	**0.00e+00**
rf_9977	0.005	0.004	0.001	1.43e-04	0.001	**0.00e+00**

Table 19: Final performance of each multi-fidelity optimizers (lower is better). We report the median normalized regret across 32 repetitions for each *new* benchmarks collected for *RandomForest*. We boldface the best result per row.

optimizer	*HB*	*BOHB*	*DEHB*	*SMAC-HB*	*DF*	$Optuna^{md}_{tpe}$	$Optuna^{hb}_{tpe}$
rf_10101	0.049	**0.00e+00**	**0.00e+00**	**0.00e+00**	0.689	**0.00e+00**	**0.00e+00**
rf_53	0.007	0.003	0.003	0.002	0.295	0.002	**0.002**
rf_146818	0.015	0.015	0.011	**0.00e+00**	0.377	0.002	0.006
rf_146821	0.013	0.001	0.001	**0.00e+00**	0.695	**0.00e+00**	0.001
rf_9952	0.002	0.004	4.83e-04	**0.00e+00**	0.371	4.83e-04	4.83e-04
rf_146822	0.002	0.002	0.002	**0.00e+00**	0.125	5.32e-04	0.001
rf_31	0.028	0.063	0.056	**0.01**	0.914	0.01	0.023
rf_3917	0.029	0.019	0.012	**4.52e-16**	0.794	0.01	0.007
rf_168912	0.016	0.004	0.012	**0.00e+00**	0.793	0.011	0.01
rf_3	0.002	0.001	5.38e-04	**0.00e+00**	0.128	**0.00e+00**	2.69e-04
rf_167119	0.009	0.005	0.005	**0.00e+00**	0.515	9.31e-04	8.62e-04
rf_12	8.75e-04	5.83e-04	5.83e-04	**0.00e+00**	0.038	3.91e-17	3.91e-17
rf_146212	5.43e-05	5.43e-05	**0.00e+00**	**0.00e+00**	0.009	2.71e-05	5.43e-05
rf_168911	0.016	0.016	0.016	**0.002**	0.749	0.004	0.005
rf_9981	0.004	0.006	0.004	**0.00e+00**	0.355	7.42e-04	7.42e-04
rf_167120	0.002	0.003	**0.002**	0.002	0.873	0.002	0.002
rf_14965	0.027	0.008	0.037	**0.002**	0.886	0.031	0.008
rf_146606	0.004	0.004	0.002	**1.86e-05**	0.601	5.86e-04	0.001
rf_7592	0.031	0.003	0.126	**0.00e+00**	0.782	0.032	0.012
rf_9977	0.002	0.001	0.003	**1.43e-04**	0.191	4.29e-04	4.29e-04

Table 20: Final performance of each black-box optimizers (lower is better). We report the median normalized regret across 32 repetitions for each *new* benchmarks collected for *MLP*. We boldface the best result per row.

optimizer	*RS*	*DE*	BO_{GP}	BO_{RF}	BO_{KDE}	*HEBO*
nn_10101	0.047	0.034	**0.03**	0.035	**0.03**	0.042
nn_53	0.005	0.004	0.004	**0.003**	0.004	0.003
nn_146818	0.011	0.009	0.008	**0.006**	0.009	0.009
nn_146821	8.78e-04	5.85e-04	5.85e-04	**2.93e-04**	8.78e-04	0.001
nn_9952	0.012	0.013	0.01	0.009	0.01	**0.008**
nn_146822	0.005	0.005	**0.003**	0.004	**0.003**	0.004
nn_31	0.014	0.012	**0.009**	0.009	0.012	0.01
nn_3917	0.009	0.01	**0.008**	0.01	0.01	0.01

Table 21: Final performance of each multi-fidelity optimizers (lower is better). We report the median normalized regret across 32 repetitions for each *new* benchmarks collected for *MLP*. We boldface the best result per row.

optimizer	*HB*	*BOHB*	*DEHB*	*SMAC-HB*	*DF*	$Optuna^{md}_{tpe}$	$Optuna^{hb}_{tpe}$
nn_10101	0.042	0.049	**0.04**	0.045	0.132	0.04	0.042
nn_53	0.006	0.008	**0.005**	0.006	0.196	0.006	0.006
nn_146818	0.013	0.016	**0.011**	**0.011**	0.111	0.015	0.011
nn_146821	**5.85e-04**	0.001	0.001	0.001	0.046	0.001	0.001
nn_9952	0.012	0.01	0.009	0.01	0.278	0.01	**0.009**
nn_146822	0.005	0.007	0.004	0.005	0.082	**0.004**	0.005
nn_31	0.013	0.013	**0.009**	0.013	0.215	0.014	0.013
nn_3917	0.011	0.012	0.011	0.013	0.153	**0.01**	0.012

Table 22: P-value of a sign test for the hypothesis that advanced methods outperform the baseline *RS* for black-box optimization and *HB* for multi-fidelity optimization for the new benchmarks. We underline p-values that are below $\alpha = 0.05$ and also boldface p-values that are below $\alpha = 0.05$ after multiple comparison correction (dividing α by the number of comparisons, i.e. 5 and 4; boldface/underlined implies that the advanced method is better). We also give the wins/ties/losses of *RS* and *HB* against the challengers.

	DE	BO_{GP}	BO_{RF}	*HEBO*	BO_{KDE}
p-value against *RS*	**0.00091**	**0.00000**	**0.00000**	**0.00000**	0.16870
wins/ties/losses against *RS*	45/28/15	69/16/3	70/17/1	68/8/12	33/32/23
	BOHB	*DEHB*	*SMAC-HB*	*DF*	
p-value against *HB*	0.99995	**0.00251**	**0.00058**	1.00000	
wins/ties/losses against *HB*	14/25/49	41/33/14	44/31/13	5/3/80	

Table 23: P-values of a sign test for the hypothesis that multi-fidelity outperform their black-box counterparts for the new benchmarks. We boldface p-values that are below $\alpha = 0.05$ (boldface implies that the multi-fidelity method is better).

Budget		*RS* vs *HB*	*DE* vs *DEHB*	BO_{KDE} vs *BOHB*	BO_{RF} vs *SMAC-HB*
100%	p-values	**0.00003**	**0.00091**	0.66586	0.99992
	w/t/l	47/32/9	49/20/19	30/25/33	7/40/41
10%	p-values	**0.00011**	**0.00058**	0.37466	**0.00042**
	w/t/l	53/17/18	55/9/24	42/8/38	54/12/22
1%	p-values	**0.00000**	**0.00000**	**0.00000**	**0.00000**
	w/t/l	76/0/12	76/2/10	78/1/9	78/2/8

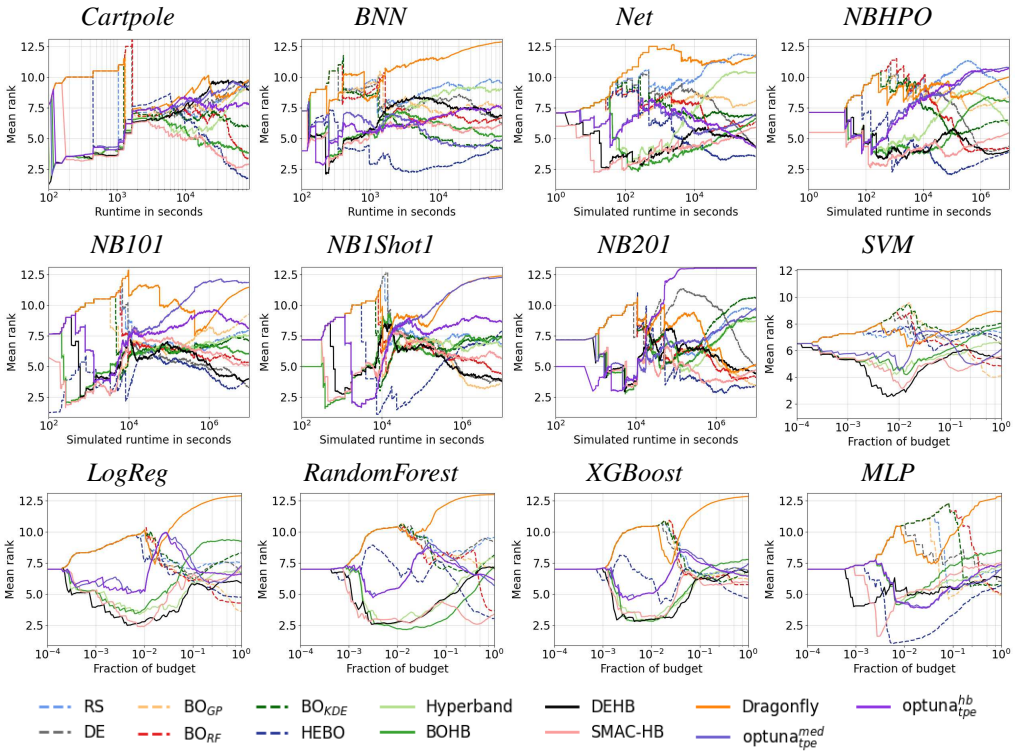

Figure 5: Median rank over time. We report the median rank of the performance across all benchmarks of a benchmark family (see Table 1) for **all** optimizers.

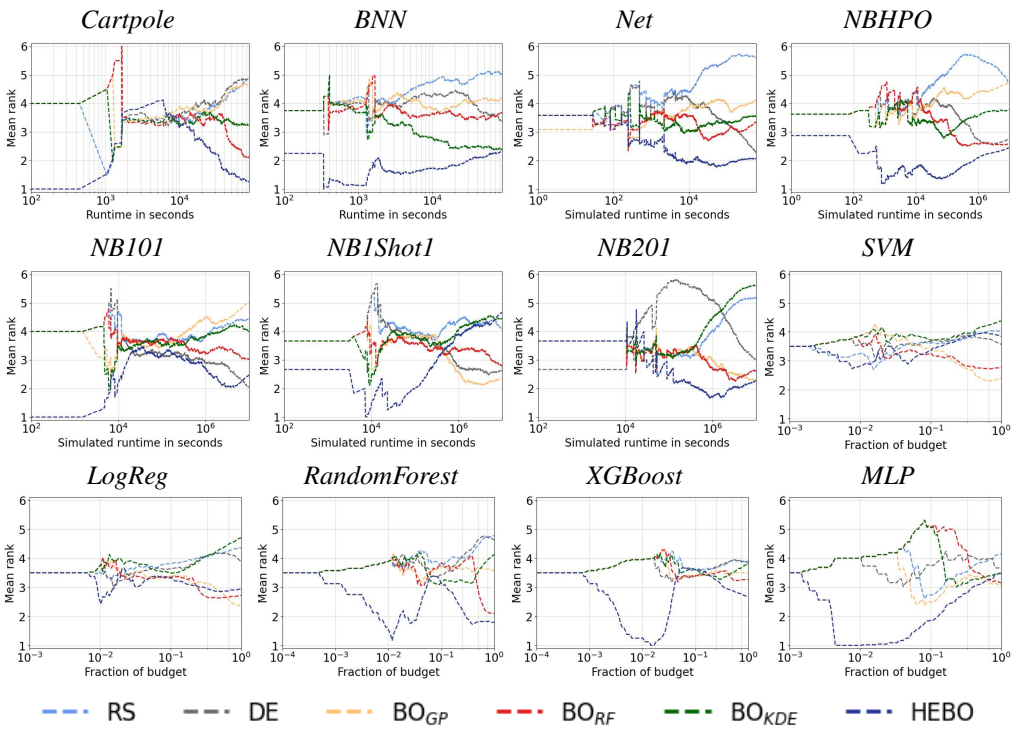

Figure 6: Median rank over time. We report the median rank of the performance across all benchmarks of a benchmark family (see Table 1) for **black-box** optimizers.

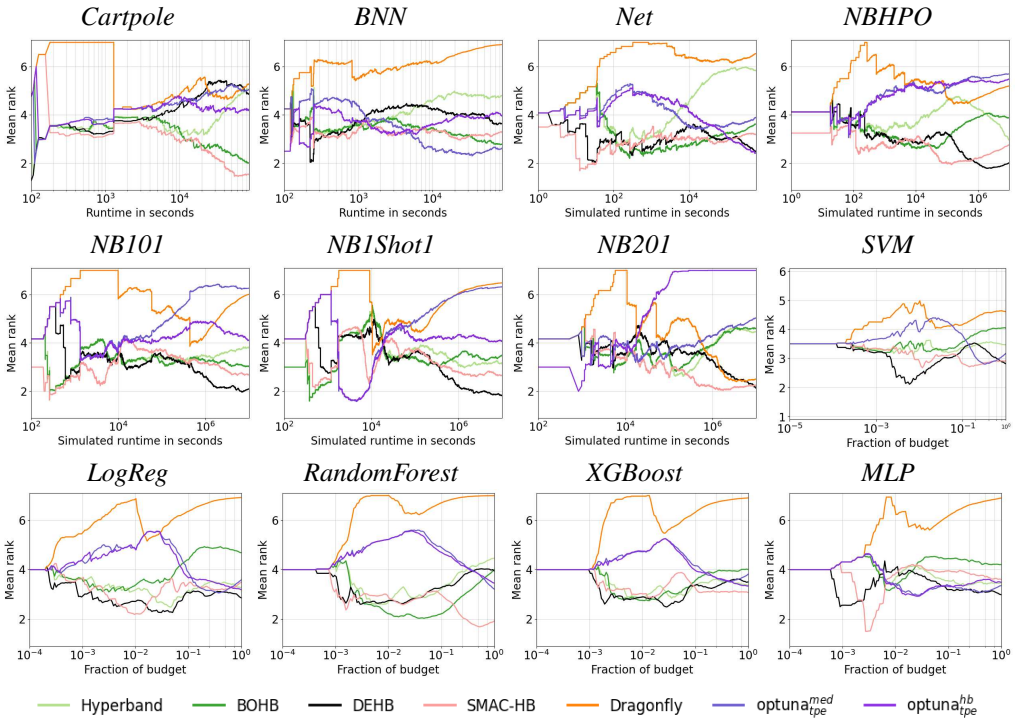

Figure 7: Median rank over time. We report the median rank of the performance across all benchmarks of a benchmark family (see Table 1) for **multi-fidelity** optimizers.

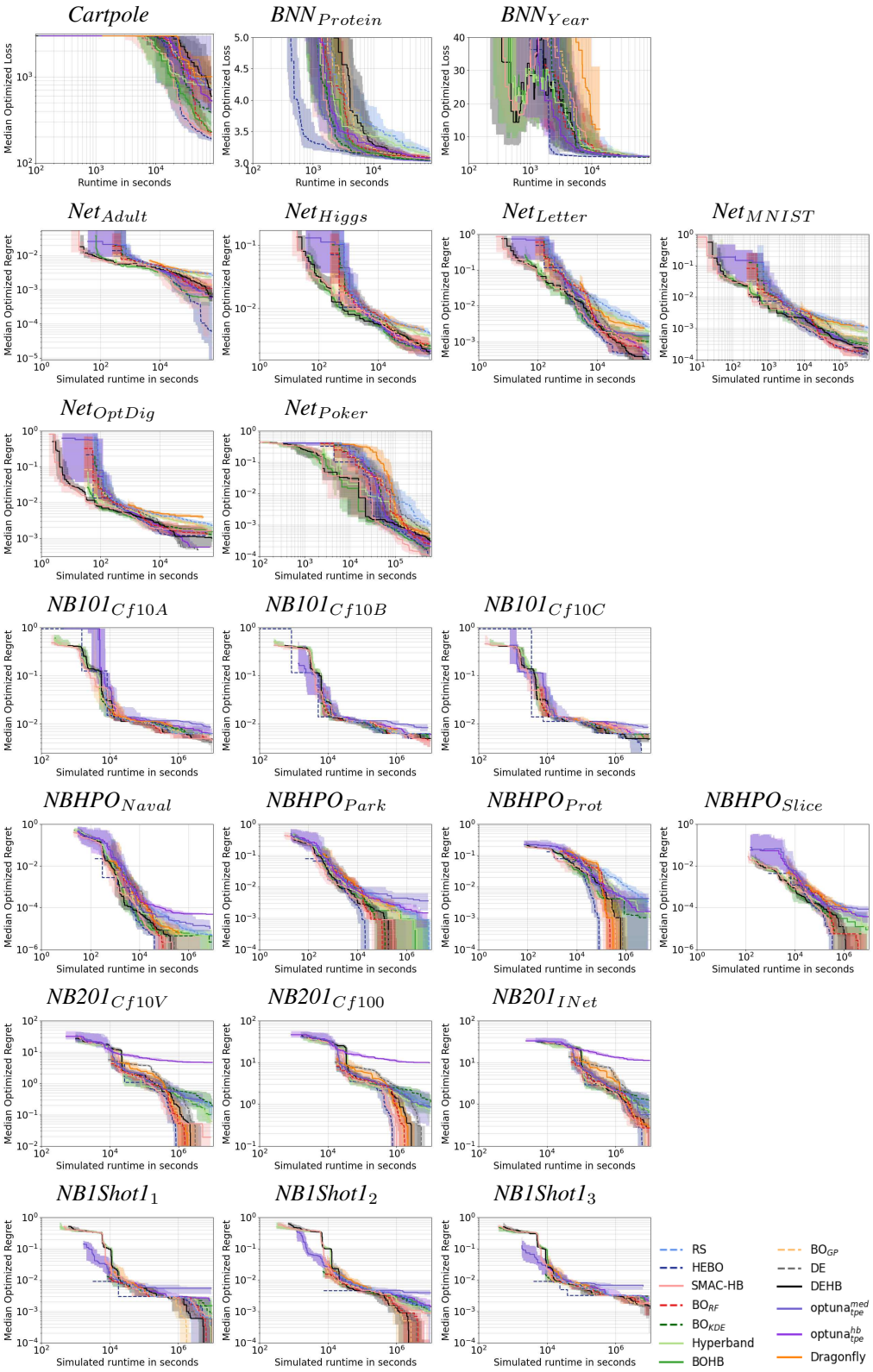

Figure 8: Median performance-over-time for **all** optimizers.

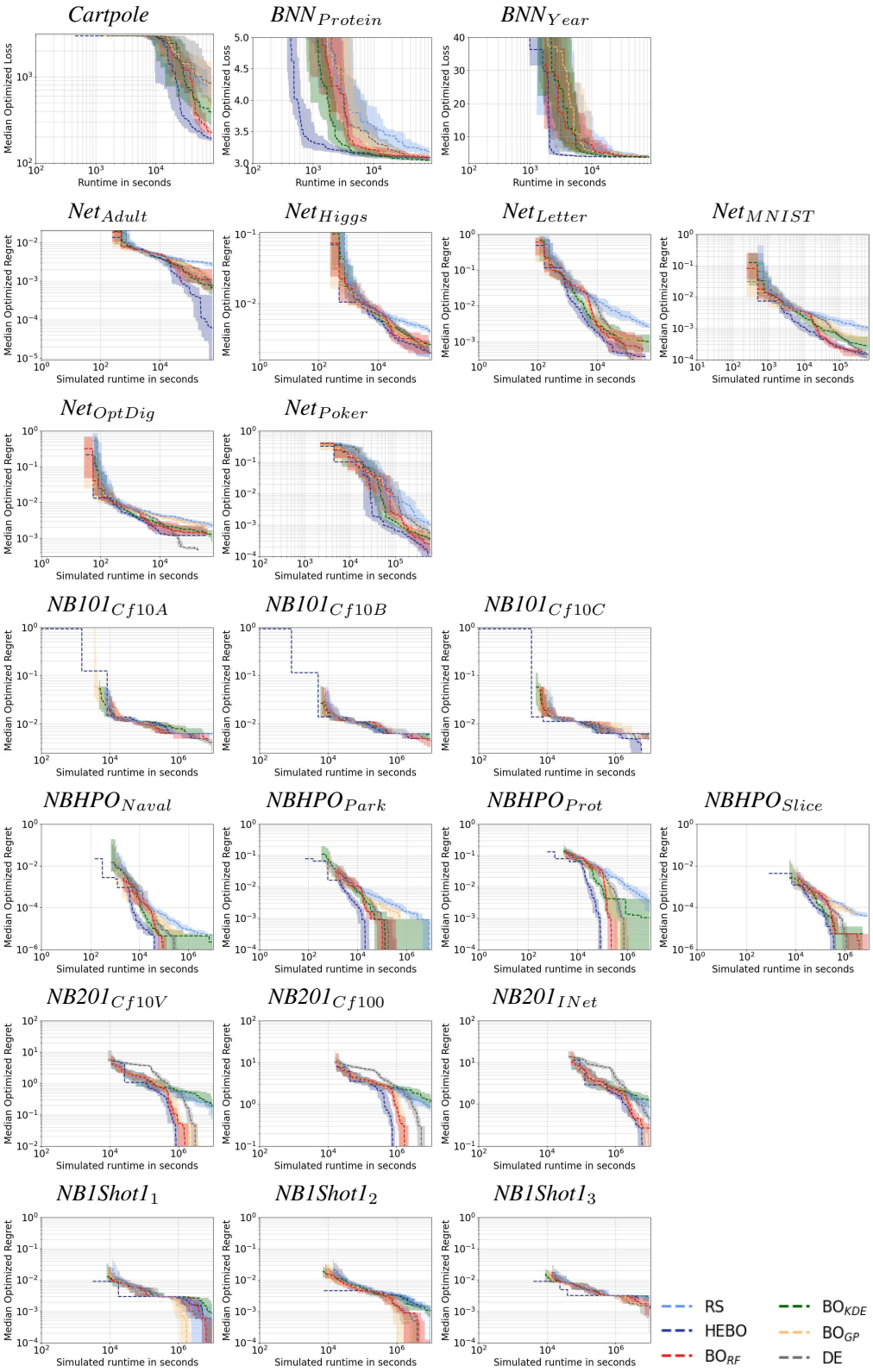

Figure 9: Median performance-over-time for **black-box** optimizers.

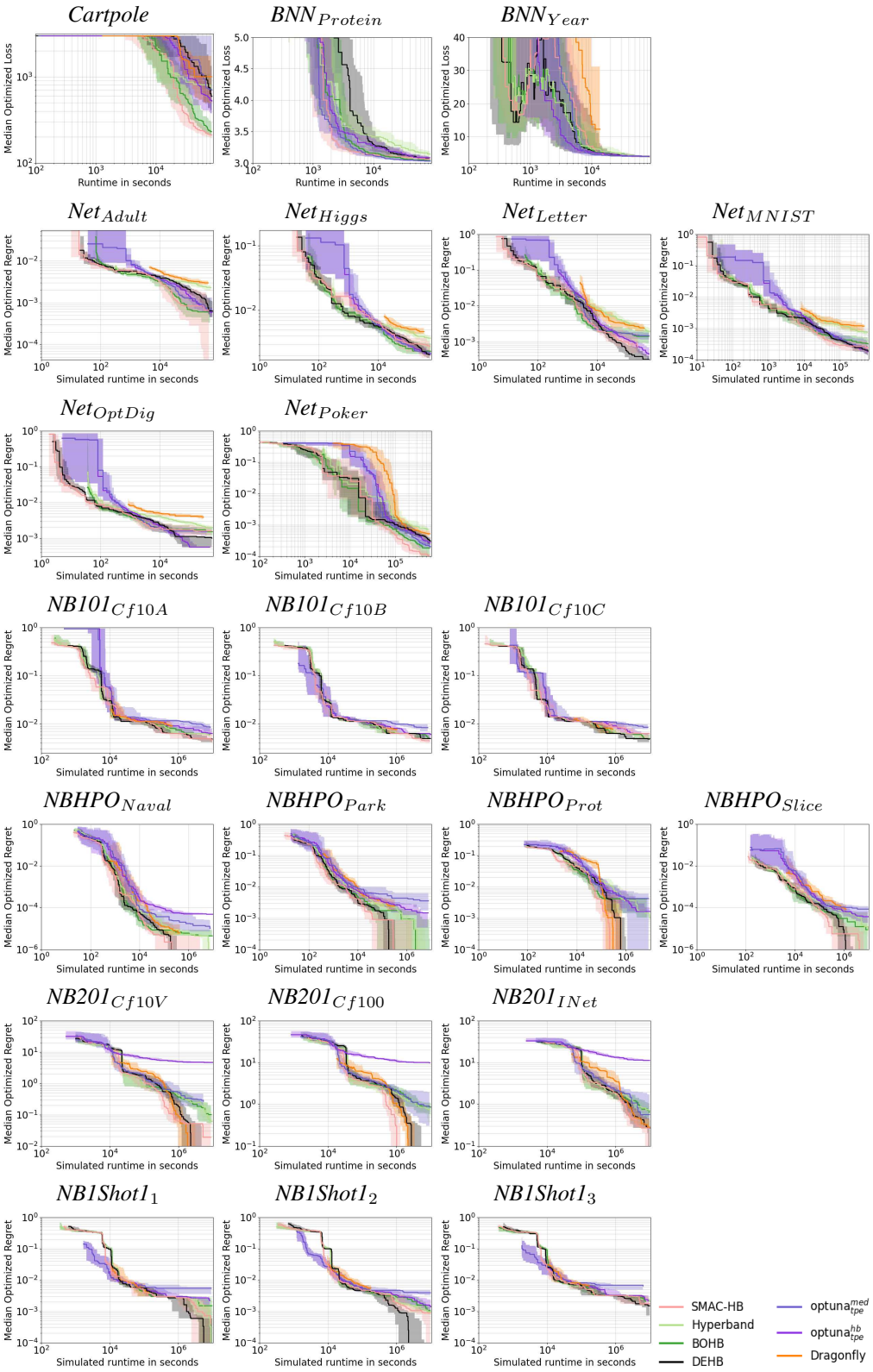

Figure 10: Median performance-over-time for **multi-fidelity** optimizers.