A Related Work

A.1 Existing NAS benchmarks

Benchmarks for NAS were introduced only recently with NAS-Bench-101 (Ying et al., 2019) as the first among them. NAS-Bench-101 is a tabular benchmark consisting of $\sim$423k unique architectures in a cell structured search space evaluated on CIFAR-10 (Krizhevsky, 2009). To restrict the number of architectures in the search space, the number of nodes and edges was given an upper bound and only three operations are considered. One result of this limitation is that One-Shot NAS methods can only be applied to subspaces of NAS-Bench-101 as demonstrated in NAS-Bench-1Shot1 (Zela et al., 2020b).

NAS-Bench-201 (Dong & Yang, 2020), in contrast, uses a search space with a fixed number of nodes and edges, hence allowing for a straight-forward application of one-shot NAS methods. However, this limits the total number of unique architectures to as few as 6466. NAS-Bench-201 includes evaluations of all these architectures on three different datasets, namely CIFAR-10, CIFAR-100 (Krizhevsky, 2009) and Downsampled Imagenet 16×16 (Chrabaszcz et al., 2017), allowing for transfer learning experiments.

NAS-Bench-NLP (Klyuchnikov et al., 2020) was recently proposed as a tabular benchmark for NAS in the Natural Language Processing domain. The search space resembles NAS-Bench-101 as it limits the number of edges and nodes to constrain the search space size resulting in 14k evaluated architectures.

A.2 Neural Network Performance Prediction

In the past, several works have attempted to predict the performance of neural networks by extrapolating learning curves (Domhan et al., 2015; Klein et al., 2017; Baker et al., 2017). A more recent line of work in performance prediction focuses more on feature encoding of neural architectures. Peephole (Deng et al., 2017) and TAPAS (Istrate et al., 2019) both use an LSTM to aggregate information about the operations in chain-structured architectures. On the other hand, BANANAS (White et al., 2019) introduces a path-based encoding of cells that automatically resolves the computational equivalence of architectures.

Graph Neural Networks (GNNs) (Gori et al., 2005; Kipf & Welling, 2017; Zhou et al., 2018; Wu et al., 2019) with their capability of learning representations of graph-structured data appear to be a natural choice to learning embeddings of NN architectures. Shi et al. (2019) and Wen et al. (2019) trained a Graph Convolutional Network (GCN) on a subset of NAS-Bench-101 (Ying et al., 2019) showing its effectiveness in predicting the performance of unseen architectures. Moreover, Friede et al. (2019) propose a new variational-sequential graph autoencoder (VS-GAE) which utilizes a GNN encoder-decoder model in the space of architectures and generates valid graphs in the learned latent space.

Several recent works further adapt the GNN message passing to embed architecture bias via extra weights to simulate the operations such as in GATES (Ning et al., 2020) or integrate additional information on the operations (e.g. flop count) (Xu et al., 2019b). Tang et al. (2020) chose to operate GNNs on relation graphs based on architecture embeddings in a metric learning setting, allowing to pose NAS performance prediction as a semi-supervised setting.

B Training Details for the GIN in the Motivation

We set the GIN to have a hidden dimension of 64 with 4 hidden layers resulting in around $\sim$40k parameters. We trained for 30 epochs with a batch size of 128. We chose the MSE loss function and add a logarithmic transformation to emphasize the data fit on well-performing architectures.

Model	Mean Squared Error (MSE)			Kendall tau		
	1, [2, 3]	2, [1, 3]	3, [1, 2]	1, [2, 3]	2, [1, 3]	3, [1, 2]
Tab.	5.44e-5	5.43e-5	5.34e-5	0.83	0.83	0.83
Surr.	**3.02e-5**	**3.07e-5**	**3.02e-5**	**0.87**	**0.87**	**0.87**

Table 6: MSE and Kendall tau correlation between performance predicted by a tab./surr. benchmark fitted with one seed each, and the true performance of evaluations with the two other seeds (see Section 2). Test seeds in brackets.

C NAS-BENCH-301 DATASET

C.1 SEARCH SPACE

We use the same architecture search space as in DARTS (Liu et al., 2019b). Specifically, the normal and reduction cell each consist of a DAG with 2 input nodes (receiving the output feature maps from the previous and previous-previous cell), 4 intermediate nodes (each adding element-wise feature maps from two previous nodes in the cell) and 1 output node (concatenating the outputs of all intermediate nodes). Input and intermediate nodes are connected by directed edges representing one of the following operations: Sep. conv 3×3, Sep. conv 5×5, Dil. conv 3×3, Dil. conv 5×5, Max pooling 3×3, Avg. pooling 3×3, Skip connection.

C.2 DATA COLLECTION

To achieve good global coverage, we use random search to evaluate $\sim$23k architectures. We note that space-filling designs such as quasi-random sequences, e.g. Sobol sequences (Sobol', 1967), or Latin Hypercubes (McKay et al., 2000) and Adaptive Submodularity (Golovin & Krause, 2011) may also provide good initial coverage.

Random search is supplemented by data which we collect from running a variety of optimizers, representing Bayesian Optimization (BO), evolutionary algorithms and One-Shot Optimizers. We used Tree-of-Parzen-Estimators (TPE) (Bergstra et al., 2011) as implemented by Falkner et al. (2018) as a baseline BO method. Since several recent works have proposed to apply BO over combinatorial spaces (Oh et al., 2019; Baptista & Poloczek, 2018) we also used COMBO (Oh et al., 2019). We included BANANAS (White et al., 2019) as our third BO method, which uses a neural network with a path-based encoding as a surrogate model and hence scales better with the number of function evaluations. As two representatives of evolutionary approaches to NAS, we chose Regularized Evolution (RE) (Real et al., 2019) as it is still one of the state-of-the art methods in discrete NAS and Differential Evolution (Price et al., 2006) as implemented by Awad et al. (2020). Accounting for the surge in interest in One-Shot NAS, our collected data collection also entails evaluation of architectures from search trajectories of DARTS (Liu et al., 2019b), GDAS (Dong & Yang, 2019), DrNAS (Chen et al., 2020) and PC-DARTS (Xu et al., 2020). For details on the architecture training details, we refer to Section C.6.

For each architecture $a \in \mathcal{A}$, the dataset contains the following metrics: train/validation/test accuracy, training time and number of model parameters.

C.3 DETAILS ON EACH OPTIMIZER

In this section we provide the hyperparameters used for the evaluations of NAS optimizers for the collection of our dataset. Many of the optimizers require a specialized representation to function on an architecture space because most of them are general HPO optimizers. As recently shown by White et al. (2020a), this representation can be critical for the performance of a NAS optimizer. Whenever the representation used by the Optimizer did not act directly on the graph representation, such as in RE, we detail how we represented the architecture for the optimizer. All optimizers were set to optimize the validation error.

BANANAS. We initialized BANANAS with 100 random architectures and modified the optimization of the surrogate model neural network, by adding early stopping based on a 90%/10% train/validation split and lowering the number of ensemble models to be trained from 5 to 3. These changes to bananas avoided a computational bottleneck in the training of the neural network.

COMBO. COMBO only attempts to maximize the acquisition function after the entire initial design (100 architectures) has completed. For workers which are done earlier, we sample a random architecture, hence increasing the initial design by the number of workers (30) we used for running the experiments. The search space considered in our work is larger than all search spaces evaluated in COMBO (Oh et al., 2019) and we regard not simply binary architectural choices, as we have to make choices about pairs of edges. Hence, we increased the number of initial samples for ascent acquisition function optimization from 20 to 30. Unfortunately, the optimization of the GP already became the bottleneck of the BO after around 600 function evaluations, leading to many workers waiting for new jobs to be assigned.

Representation: In contrast to the COMBO's original experimental setting, the DARTS search requires choices based on pairs of parents of intermediate nodes where the number of choices increase with the index of the intermediate nodes. The COMBO representation therefore consists of the graph cartesian product of the combinatorial choice graphs, increasing in size with each intermediate node. In addition, there exist 8 choices over the number of parameters for the operation in a cell.

Differential Evolution. DE was started with a generation size of 100. As we used a parallelized implementation, the workers would have to wait for one generation plus its mutations to be completed for selection to start. We decided to keep the workers busy by training randomly sampled architectures in this case, as random architectures provide us good coverage of the space. However, different methods using asynchronous DE selection would also be possible. Note, that the DE implementation by Awad et al. (2020), performs boundary checks and resamples components of any individual that exceeds 1.0. We use the rand1 mutation operation which generally favors exploration over exploitation.

Representation: DE uses a vector representation for each individual in the population. Categorical choices are scaled to lie within the unit interval [0, 1] and are rounded to the nearest category when converting back to the discrete representation in the implementation by Awad et al. (2020). Similarly to COMBO, we represent the increasing number of parent pair choices for the intermediate nodes by interpreting the respective entries to have an increasing number of sub-intervals in [0, 1].

DARTS, GDAS, PC-DARTS and DrNAS. We collected the architectures found by all of the above one-shot optimizers with their default search hyperparameters. We performed multiple searches for each one-shot optimizer.

RE. To allow for a good initial coverage before mutations start, we decided to randomly sample 3000 architectures as initial population. RE then proceeds with a sample size of 100 to extract well performing architectures from the population and mutates them. During mutations RE first decides whether to mutate the normal or reduction cell and then proceeds to perform either a parent change, an operation change or no mutation.

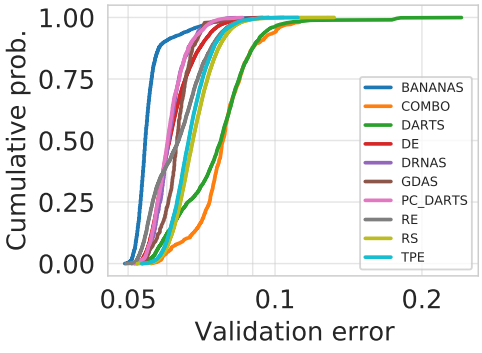


Figure 8: Empirical Cumulative Density Function (ECDF) plot comparing all optimizers in the dataset. Optimizers which cover good regions of the search space feature higher values in the low validation error region.

TPE. For TPE we use the default settings as also used by BOHB. We use the Kernel-Density-Estimator surrogate model and build two models where the good configs are chosen as the top 15%. The acquisition function's expected improvement is optimized by sampling 64 points.

C.4 Optimizer Performance

The trajectories from the different NAS optimizers yield quite different performance distributions. This can be seen in Figure 8 which shows the ECDF of the validation errors of the architectures evaluated by each optimizer. As the computational budgets allocated to each optimizer vary widely, this data does not allow for a fair comparison between the optimizers. However, it is worth mentioning that the evaluations of BANANAS feature the best distribution of architecture performances, followed by PC-DARTS, DrNAS, DE, GDAS, and RE. TPE only evaluated marginally better architectures than RS, while COMBO and DARTS evaluated the worst architectures.

We also perform a t-SNE analysis on the data collected by the different optimizers in Figure 9. We find that RE discovers well-performing architectures which form clusters distinct from the architectures found via RS. We observe that COMBO searched previously unexplored areas of the search space. BANANAS, which found some of the best architectures, explores clusters outside the main cluster. However, it heavily exploits regions at the cost of exploration. We argue that this is a result of the optimization of the acquisition function via random mutations based on the previously found iterates, rather than on new random architectures. DE is the only optimizer which finds well performing architectures in the center of the embedding space.

C.5 Cell topology, operations and noise

In this section, we investigate the influence of the cell topology and the operations on the performance of the architectures in our setting. The discovered properties of the search space inform our choice of metrics for the evaluation of different surrogate models.

First, we study how the validation error depends on the depth of architectures. Figure 10 visualizes the performance distribution of normal and reduction cells of different depth[5] by approximating empirical distributions with a kernel density estimation used in violin plots (Hwang et al., 1994). We observe that the performance distributions are similar for the normal and reduction cells with the same cell depth. Although cells of all depths can reach high performances, shallower cells seem slightly favored. Note that these observations are subject to changes in the hyperparameter setting, e.g. training for more epochs may render deeper cells more competitive. The best-found architecture features a normal and reduction cell of depth 4. Color-coding the cell depth in our t-SNE projection also confirms that the t-SNE analysis captures the cell depth well as a structural property (c.f. Figure 13). It also reinforces that the search space is well-covered.

We also show the distribution of normal and reduction cell depths of each optimizer in Figure 11 to get a sense for the diversity between the discovered architectures. We observe that DARTS and BANANAS generally find architectures with a shallow reduction cell and a deeper normal cell, while the reverse is true for RE. DE, TPE, COMBO and RS appear to find normal and reduction cells with similar cell depth.

Aside from the cell topology, we can also use our dataset to study the influence of operations to the architecture performance. The DARTS search space contains operation choices without parameters such as Skip-Connection, Max Pooling 3×3 and Avg Pooling 3×3. We visualize the influence of these parameter-free operations on the validation error in the normal and reduction cell in Figure 18a, respectively Figure 14. While pooling operations in the normal cell seem to have a negative impact on performance, a small number of skip connections improves the overall performance. This is somewhat expected, since the normal cell is dimension preserving and skip connections help training by improving gradient flow like in ResNets (He et al., 2016). In the reduction cell, the number of parameter-free operations has less effect as shown in Figure 14. In contrast to the normal cell where 2-3 skip-

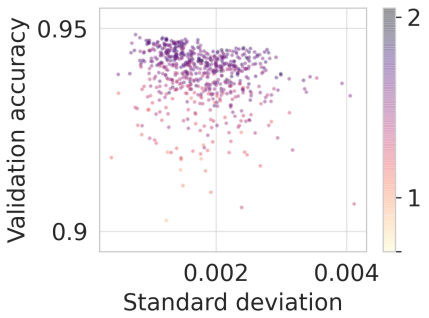


Figure 12: Standard deviation of the val. accuracy for multiple architecture evaluations.

[5]We follow the definition of cell depth used by Shu et al. (2020), i.e. the length of the longest simple path through the cell.

connections lead to generally better performance, the reduction cell shows no similar trend. For both cells, however, featuring many parameter-free operations significantly deteriorates performance. We therefore expect that a good surrogate also models this case as a poorly performing region.

C.6 TRAINING DETAILS

Each architecture was evaluated on CIFAR-10 (Krizhevsky, 2009) using the standard 40k, 10k, 10k split for train, validation and test set. The networks were trained using SGD with momentum 0.9, initial learning rate of 0.025 and a cosine annealing schedule (Loshchilov & Hutter, 2017), annealing towards 10^{-8}.

We apply a variety of common data augmentation techniques which differs from previous NAS benchmarks where the training accuracy of many evaluated architectures reached 100% (Ying et al., 2019; Dong & Yang, 2020) indicating overfitting on the training set. We used CutOut (DeVries & Taylor, 2017) with cutout length 16 and MixUp (Zhang et al., 2018) with alpha 0.2. For regularization, we used an auxiliary tower (Szegedy et al., 2015) with a weight of 0.4 and DropPath (Larsson et al., 2017) with drop probability of 0.2. We trained each architecture for 100 epochs with a batch

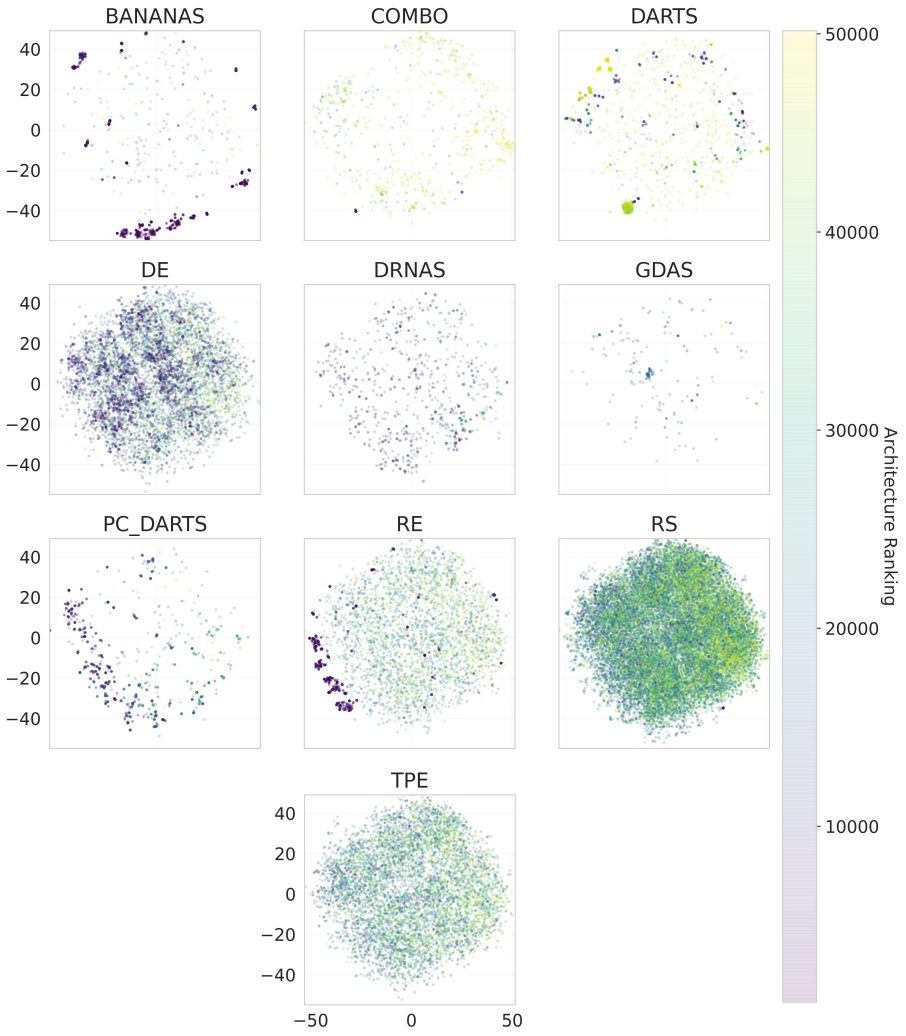


Figure 9: Visualization of the exploration of different parts of the architectural t-SNE embedding space for all optimizers used for data collection. The architecture ranking by validation accuracy (lower is better) is global over the entire data collection of all optimizers.

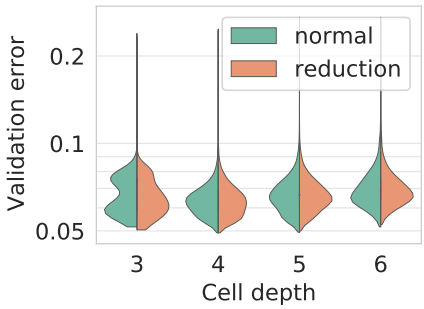


Figure 10: Distribution of the validation error for different cell depth.

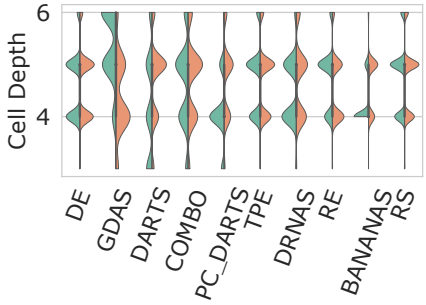


Figure 11: Comparison between the normal and reduction cell depth for the architectures found by each optimizer.

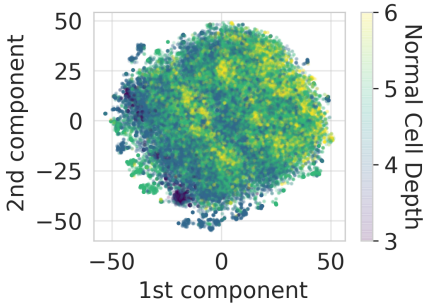


Figure 13: t-SNE projection colored by the depth of the normal cell.

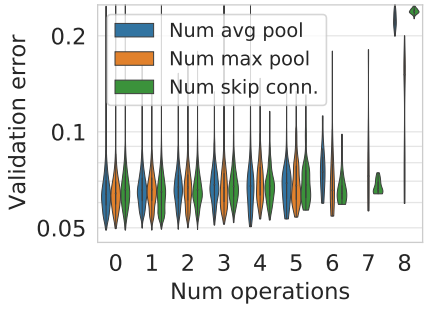


Figure 14: Distribution of validation error in dependence of the number of parameter-free operations in the reduction cell. Violin plots are cut off at the respective observed minimum and maximum value.

size of 96, using 32 initial channels and 8 cell layers. We chose these values to be close to the proxy model used by DARTS while also achieving good performance.

D Surrogate Model Analysis

D.1 Preprocessing of the graph topology

DGN preprocessing All DGN were implemented using PyTorch Geometric (Fey & Lenssen, 2019) which supports the aggregation of edge attributes. Hence, we can naturally represent the DARTS architecture cells, by assigning the embedded operations to the edges. The nodes are labeled as input, intermediate and output nodes. We represent the DARTS graph as shown in Figure 15, by connecting the output node of each cell type with the inputs of the other cell, allowing information from both cells to be aggregated per node during message passing. Note the self-loop on the output node of the normal cell, which we found necessary to get the best performance.

Preprocessing for other surrogate models Since we make use of the framework implemented by BOHB (Falkner et al., 2018) to easily parallelize the architecture search algorithms across many compute nodes, we also represent our search space using ConfigSpace [6] (Lindauer et al., 2019). More precisely, we encode each pair of incoming edges for a cell as one choice of a categorical parameter. For instance, for node 4 in the normal cell, we add a parameter `inputs_node_normal_4` with the choices of edge pairs `0_1, 0_2, 0_3, 1_2, 1_3, 2_3`. The edge operations are then implemented as categorical parameters for each edge and are only active if the corresponding edge was

[6] `https://github.com/automl/ConfigSpace`

chosen. For instance, in the example above, if the incoming edge 0 is sampled, the parameter associated with the edge from node 0 to node 4 becomes activate and one operation is sampled. We provide the configuration space with our code. For all non-DGN based surrogate models, we use the vector representation of a configuration given by ConfigSpace as input to the model. This vector representation contains one value between 0 and 1 for each parameter in the configuration space.

D.2 Details on the GIN

The GIN implementation on the Open Graph Benchmark (OGB) (Hu et al., 2020) uses virtual nodes (additional nodes which are connected to all nodes in the graph) to boost performance as well as generalization and consistently achieves good performance on their public leaderboards. Other GNNs from Errica et al. (2020), such as DGCNN and DiffPool, performed worse in our initial experiments and are therefore not considered.

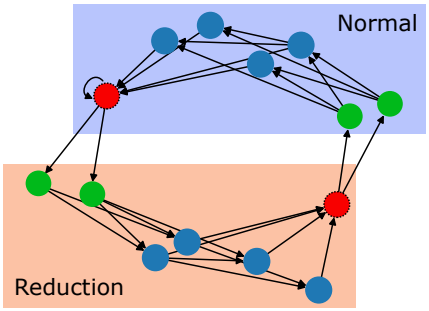


Figure 15: Architecture with inputs in green, intermediate nodes in blue and outputs in red.

Following recent work in Predictor-based NAS (Ning et al., 2020; Xu et al., 2019b), we use a per batch ranking loss because the ranking of an architecture is equally important to an accurate prediction of the validation accuracy in a NAS setting. We use the ranking loss formulation by GATES (Ning et al., 2020) which is a hinge pair-wise ranking loss with margin m=0.1.

D.3 Details on HPO

All detailed table for the hyperparameter ranges for the HPO and the best values found by BOHB are listed in Table 7.

D.4 HPO for runtime prediction model

Our runtime prediction model is an LGB model trained on the runtimes of architecture evaluations of DE. This is because we partially evaluated the architectures utilizing different CPUs. Hence, we only choose to train on the evaluations carried out by the same optimizer on the same hardware to keep a consistent estimate of the runtime. DE is a good choice in this case because it both explored and exploited the architecture space well. The HPO space used for the LGB runtime model is the same used for the LGB surrogate model.

D.5 Leave One-Optimizer-Out Analysis

A detailed scatter plot of the predicted performance against the true performance for each optimizer and surrogate model in an LOOO analysis is provided in Figure 16 and Figure 17.

D.6 Parameter-free Operations

Several works have found that methods based on DARTS (Liu et al., 2019b) are prone to finding sub-optimal architectures that contain many, or even only, parameter-free operations (max. pooling, avg. pooling or skip connections) and perform poorly (Zela et al., 2020a). We therefore evaluated the surrogate models on such architectures by replacing a random selection of operations in a cell with one type of parameter-free operations to match a certain ratio of parameter-free operations in a cell. This analysis is carried out over the test set of the surrogate models and hence contains architectures collected by all optimizers. For a more robust analysis, we repeated this experiment 4 times for each ratio of operations to replace.

Results Figure 18 shows that both the GIN and the XGB model correctly predict that the accuracy drops with too many parameter-free operations, particularly for skip connections. The groundtruth of architectures with only parameter-free operations is displayed as scatter plot. Out of the two models,

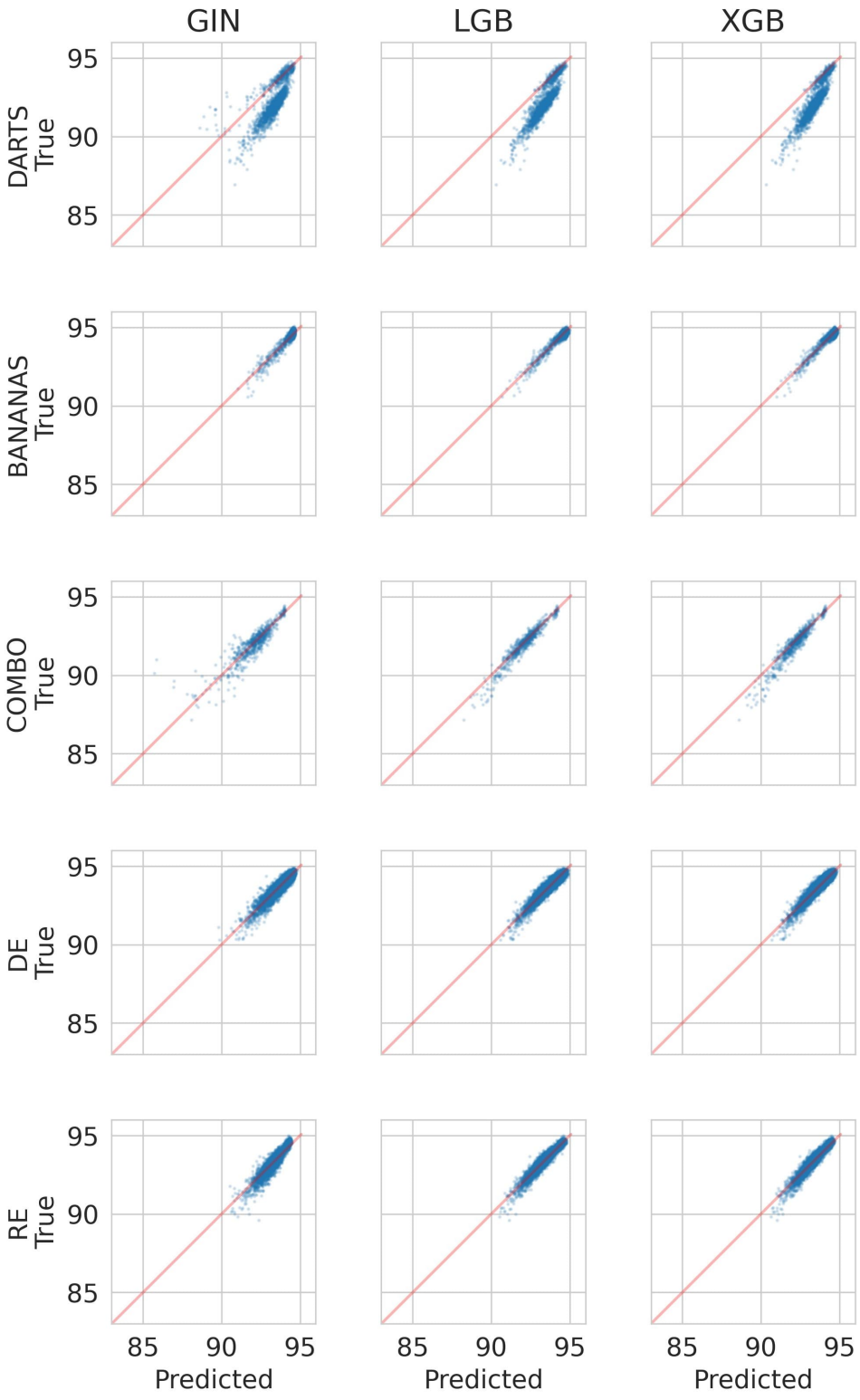


Figure 16: Scatter plots of the predicted performance against the true performance of different surrogate models on the test set in a Leave-One-Optimizer-Out setting.

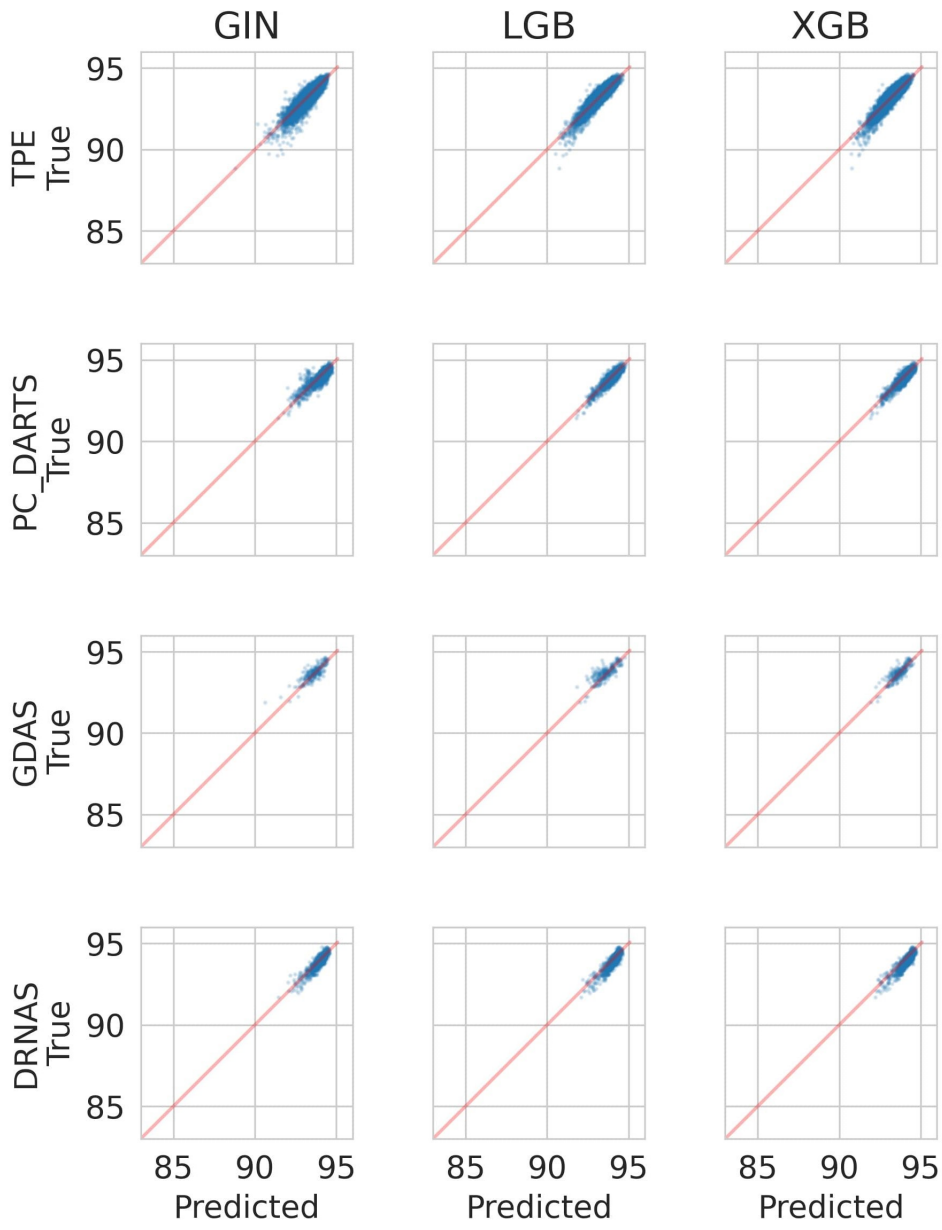
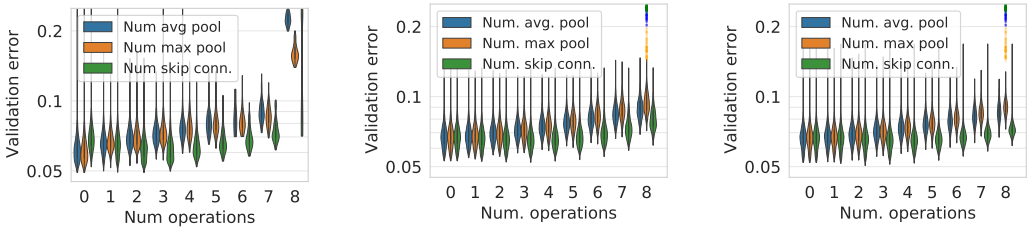


Figure 17: (continued) Scatter plots of the predicted performance against the true performance of different surrogate models on the test set in a Leave-One-Optimizer-Out setting.

Model	Hyperparameter	Range	Log-transform	Default Value
GIN	Hidden dim.	[16, 256]	true	24
	Num. Layers	[2, 10]	false	8
	Dropout Prob.	[0, 1]	false	0.035
	Learning rate	[1e-3, 1e-2]	true	0.0777
	Learning rate min.	const.	-	0.0
	Batch size	const.	-	51
	Undirected graph	[true, false]	-	false
	Pairwise ranking loss	[true, false]	-	true
	Self-Loops	[true, false]	-	false
	Loss log transform	[true, false]	-	true
	Node degree one-hot	const.	-	true
BANANAS	Num. Layers	[1, 10]	true	17
	Layer width	[16, 256]	true	31
	Dropout Prob.	const.	-	0.0
	Learning rate	[1e-3, 1e-1]	true	0.0021
	Learning rate min.	const.	-	0.0
	Batch size	[16, 128]	-	122
	Loss log transform	[true, false]	-	true
	Pairwise ranking loss	[true, false]	-	false
XGBoost	Early Stopping Rounds	const.	-	100
	Booster	const.	-	gbtree
	Max. depth	[1, 15]	false	13
	Min. child weight	[1, 100]	true	39
	Col. sample bylevel	[0.0, 1.0]	false	0.6909
	Col. sample bytree	[0.0, 1.0]	false	0.2545
	lambda	[0.001, 1000]	true	31.3933
	alpha	[0.001, 1000]	true	0.2417
	Learning rate	[0.001, 0.1]	true	0.00824
LGBoost	Early stop. rounds	const.	-	100
	Max. depth	[1, 25]	false	18
	Num. leaves	[10, 100]	false	40
	Max. bin	[100, 400]	false	336
	Feature Fraction	[0.1, 1.0]	false	0.1532
	Min. child weight	[0.001, 10]	true	0.5822
	Lambda L1	[0.001, 1000]	true	0.0115
	Lambda L2	[0.001, 1000]	true	134.5075
	Boosting type	const.	-	gbdt
	Learning rate	[0.001, 0.1]	true	0.0218
Random Forest	Num. estimators	[16, 128]	true	116
	Min. samples split.	[2, 20]	false	2
	Min. samples leaf	[1, 20]	false	2
	Max. features	[0.1, 1.0]	false	0.1706
	Bootstrap	[true, false]	-	false
ϵ-SVR	C	[1.0, 20.0]	true	3.066
	coef. 0	[-0.5, 0.5]	false	0.1627
	degree	[1, 128]	true	1
	epsilon	[0.01, 0.99]	true	0.0251
	gamma	[scale, auto]	-	auto
	kernel	[linear, rbf, poly, sigmoid]	-	sigmoid
	shrinking	[true, false]	-	false
	tol	[0.0001, 0.01]	-	0.0021
μ-SVR	C	[1.0, 20.0]	true	5.3131
	coef. 0	[-0.5, 0.5]	false	-0.3316
	degree	[1, 128]	true	128
	gamma	[scale, auto]	-	scale
	kernel	[linear, rbf, poly, sigmoid]	-	rbf
	nu	[0.01, 1.0]	false	0.1839
	shrinking	[true, false]	-	true
	tol	[0.0001, 0.01]	-	0.003

Table 7: Hyperparameters of the surrogate models and the default values found via HPO.

XGB captures the slight performance improvement of using a few skip connections better. LGB failed to capture this trend but performed very similarly to XGB for the high number of parameter-free operations.



(a) NAS-Bench-301 dataset (b) GIN (c) XGB

Figure 18: (Left) Distribution of validation error in dependence of the number of parameter-free operations in the normal cell on the NAS-Bench-301 dataset. (Middle and Right) Predictions of the GIN and XGB surrogate model. The collected groundtruth data is shown as scatter plot. Violin plots are cut off at the respective observed minimum and maximum value.

D.7 Cell Topology Analysis

Furthermore, we analyze how accurate changes in the cell topology (rather than in the operations) are modeled by the surrogates. We collected groundtruth data by evaluating all $\prod_{k=1}^{4} \frac{(k+1)k}{2} = 180$ different cell topologies (not accounting for isomorphisms) with fixed sets of operations. We assigned the same architecture to the normal and reduction cell, to focus on the effect of the cell topology. We sampled 10 operation sets uniformly at random, leading to 1800 architectures as groundtruth for this analysis.

We evaluated all architectures and group the results based on the cell depth. For each of the possible cell depths, we then computed the sparse Kendall τ rank correlation between the predicted and true validation accuracy.

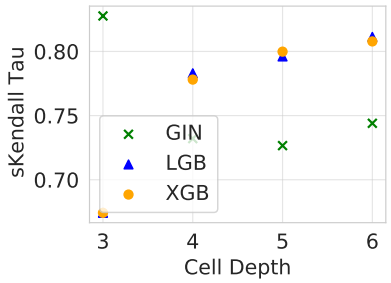


Figure 19: Comparison between GIN, XGB and LGB in the cell topology analysis.

Results Results of the cell topology analysis are shown in Figure 19. We observe that LGB slightly outperforms XGB, both of which perform better on deeper cells. The GIN performs best for the shallowest cells.

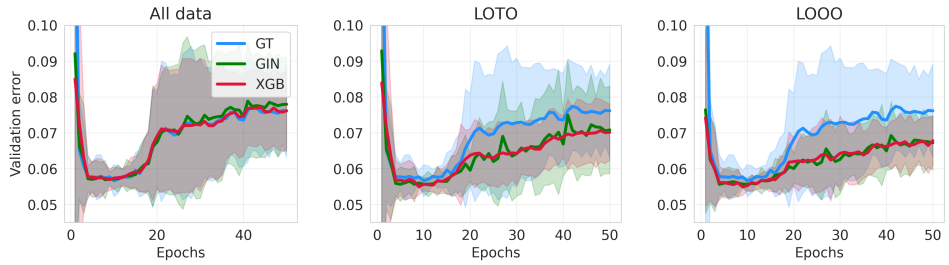


Figure 20: Ground truth (GT) and surrogate trajectories on a constrained search space where the surrogates are trained with all data, leaving out the trajectories under consideration (LOTO), and leaving out all DARTS architectures (LOOO).

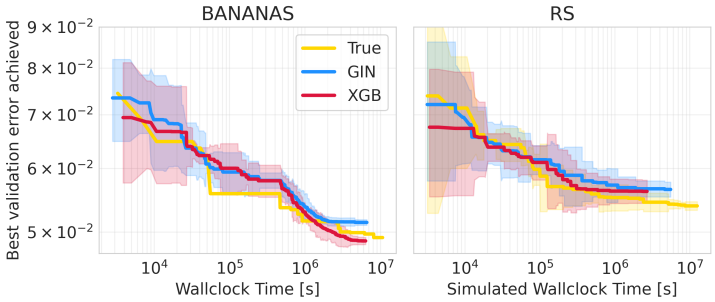


Figure 22: Comparison between the observed true trajectory of BANANAS and RS with the surrogate benchmarks only trained on well performing regions of the space

E Benchmark Analysis

E.1 One-Shot Trajectories

To obtain groundtruth trajectories for DARTS, PC-DARTS and GDAS, we performed 5 runs for each optimizer with 50 search epochs and evaluated the architecture obtained by discretizing the one-shot model at each search epoch. For DARTS, in addition to the default search space, we collected trajectories on the constrained search spaces from Zela et al. (2020a) to cover a failure case where DARTS diverges and finds architectures that only contain skip connections in the normal cell. To show that our benchmark is able to predict this divergent behavior, we show surrogate trajectories when training on all data, when leaving out the trajectories under consideration from the training data, and when leaving out all DARTS data in Figure 20.

While the surrogates model the divergence in all cases, they still overpredict the architectures with only skip connections in the normal cell especially when leaving out all data from DARTS. The bad performance of these architectures is predicted more accurately when including data from other DARTS runs. This can be attributed to the fact that the surrogate models have not seen any, respectively very few data, in this region of the search space. Nevertheless, it is modeled as a bad-performing region and we expect that this could be further improved on by including additional training data accordingly, since including all data in training shows that the models are capable to of capturing this behavior.

E.2 Ablation Study: Fitting surrogate models only on well-performing regions of the search space

To assess whether poorly-performing architectures are important for the surrogate benchmark, we fitted a GIN ensemble and an XGB ensemble model only on architectures that achieved a validation accuracy above 92%. We then tested on all architectures that achieved a validation below 92%.

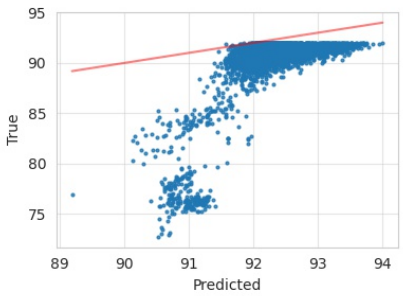


Figure 21: Scatter plot of GIN predictions on architectures that achieved below 92% validation accuracy.

Indeed, we observe that the resulting surrogate model overpredicts accuracy in regions of the space with poor performance, resulting in a low R^2 of -0.142 and sparse Kendall tau of 0.293 for the GIN. The results for one member of the GIN ensemble are shown in Figure 21. The XGB model achieved similar results. Next, to study whether these weaker surrogate models can still be used to benchmark NAS optimizers, we also studied optimization trajectories of NAS optimizers on surrogate benchmarks based on these surrogate models. Figure 22 shows that these surrogate models indeed suffice to accurately predict the performance achieved by Random Search and BANANAS as a function of time.

E.3 Ablation Study: Fitting surrogate models only with random data

In this section, we would like to take the Leave-One-Optimizer-Out analysis from Section 5.1 one step further by leaving out *all* architectures that were collected from NAS optimizers other than random search. While the LOOO analysis removes some "bias" from the benchmark ("bias" referring to its precision in a subspace), there still is the possibility that different optimizers we used explore similar subspaces, and leaving out one of them still yields "bias" induced by architectures from a similar optimizer used for generating training data. For instance, the t-SNE analysis from Figure 9 suggests that some optimizers exploit very distinct regions (e.g., BANANAS and DE) while others exploit regions somewhat similar to others (e.g., RE and PC-DARTS). The exploration behavior, on the other hand, is quite similar across optimizers since most of them perform random sampling in the beginning. Thus, in the following, we investigate whether we can create a benchmark that has no prior information about solutions any optimizer might find.

To that end, we studied surrogate models based i) only on the 23746 architectures explored by random search and ii) only on 23 746 (47.3%) architectures of the original training set (sampled in a stratified manner, i.e., using 47.3% of the architectures from each of our sources of architectures).

First, we investigated the difference in the predictive performance of surrogates based on these two different types of architectures. Specifically, we fitted our GNN and XGB surrogate models on different subsets of the respective training sets and assess their predictions on unseen architectures from all optimziers as a test set. Figure 23 shows that including architectures from optimizer trajectories in the training set consistently yields significantly better generalization.

Next, we also studied the usefulness of surrogate benchmarks based on the 23 746 random architectures, compared to surrogate benchmarks based on the 23 746 architectures sampled in a stratified manner from the original set of architectures. Specifically, we used them to assess the best performance achieved by various NAS optimizers as a function of time. Comparing the trajectories in Figure 24 (based on purely random architectures for training) and Figure 25 (based on 23 746 architectures sampled in a stratified manner), we find that the surrogates fitted only on random architectures work just as well for this task as the surrogates that use architectures from NAS optimizers in their training set.

Given this positive result for surrogates based purely on random architectures, we conclude that it is indeed possible to create surrogate NAS benchmarks that are by design free of bias towards any particular NAS optimizer (other than random search). While the inclusion of architectures generated with NAS optimizers in the training set substantially improves performance predictions of individual architectures, realistic trajectories of incumbent performance as a function of time can also be obtained with surrogate benchmarks based solely on random architectures. We note that the "unbiased" benchmark could possibly be further improved by utilizing more sophisticated space-filling sampling methods, such as the ones mentioned in Appendix C.2, or by deploying surrogate models that extrapolate well.

F Guidelines for Creating Surrogate Benchmarks

In order to help with the design of realistic surrogate benchmarks in the future, we provide the following list of guidelines:

- Data Collection: The data collected for the NAS benchmark should provide (1) a good overall coverage, (2) explore strong regions of the space well, and (3) optimally also cover special areas in which poor generalization performance may otherwise be expected. We would like to stress that depending on the search space, a good overall coverage may already be sufficient to correctly assess the ranking of different optimizers, but as shown in Appendix E.3 additional architectures from strong regions of the space allow to increase the fidelity of the surrogate model.
    1. A good overall coverage can be obtained by random search (as in our case), but one could also imagine using better space-filling designs or adaptive methods for covering the space even better. In order to add additional varied architectures, one could also think about fitting one or more surrogate models to the data collected thus far, finding the regions of maximal predicted uncertainty, evaluate architectures there and add them to the collected data, and iterate. This would constitute an active learning approach.

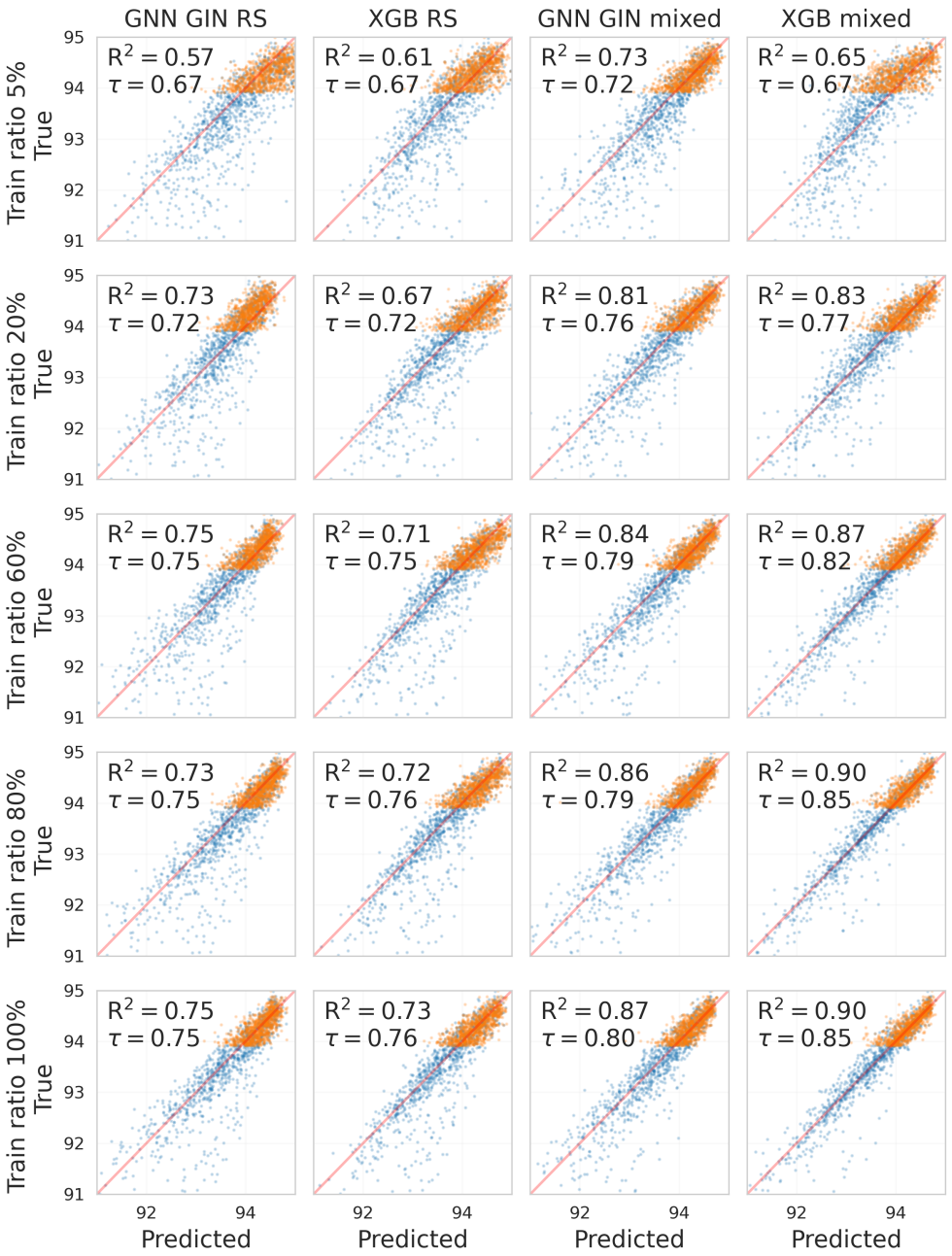


Figure 23: Scatter plots of the predicted performance against the true performance of the GNN GIN/XGB surrogate models trained with different ratios of training data. ”RS” indicates that the training set only includes architectures from random search, ”mixed” indicates the training set includes architectures from all optimizers. Training set sizes are identical for the two cases. The test set contains architectures from all optimizers. For better display, we show 1000 randomly sampled architectures (blue) and 1000 architectures sampled from the top 1000 architectures (orange). For each case we also show the R^2 and Kendall-τ coefficients on the whole test set.

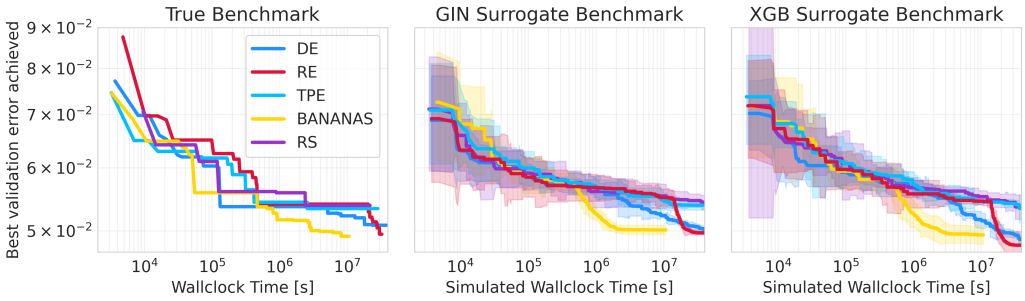


Figure 24: Anytime performance of different optimizers on the real benchmark (left) and the surrogate benchmark (GIN (middle) and XGB (right)) when training ensembles only on data collected by random search. Trajectories on the surrogate benchmark are averaged over 5 optimizer runs.

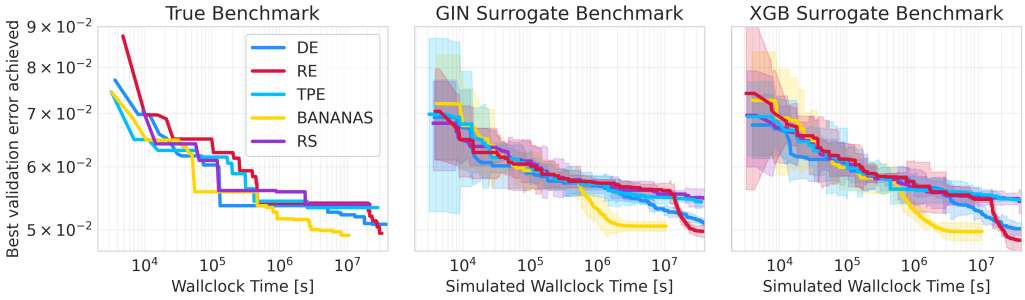


Figure 25: Anytime performance of different optimizers on the real benchmark (left) and the surrogate benchmark (GIN (middle) and XGB (right)) when training ensembles on 47.3% of the data collected from all optimizers. Trajectories on the surrogate benchmark are averaged over 5 optimizer runs.

2. A convenient and efficient way to identify regions of strong architectures is to run NAS methods. In this case, the found regions should not only be based on the strong architectures one NAS method finds but rather on a set of strong and varied NAS methods (such as, in our case, one-shot methods and different types of discrete methods, such as Bayesian optimization and evolution). In order to add additional strong architectures, one could also think about fitting one or more several surrogate models to the data collected thus far, finding the predicted optima of these models, evaluate and add them to the collected data and iterate. This would constitute a special type of Bayesian optimization.
3. Special areas in which poor generalization performance may otherwise be expected may, as in our case, e.g., include architectures with many parameterless connections, and in particular, skip connections. Other types of failure modes the community learns about would also be useful to cover.

- Surrogate Models: As mentioned in the guidelines for using a surrogate benchmark (see Section 7), benchmarking an algorithm that internally uses the same model type as the surrogate model should be avoided. Therefore, to provide a benchmark for a diverse set of algorithms, we recommend providing different types of surrogate models with a surrogate benchmark. Also, in order to guard against a possible case of "bias" in a surrogate benchmark (in the sense of making more accurate predictions for architectures explored by a particular type of NAS optimizer), we recommend to provide two versions of a surrogate: one based on all available training architectures (including those found by NAS optimizers), and one based only on the data gathered for overall coverage (1. above).

- Verification: As a means to verify surrogate models, we stress the importance of leave-one-optimizer-out experiments both for data fit and benchmarking, which simulate the benchmarking of 'unseen' optimizers.

- Since most surrogate benchmarks will continue to grow for some time after their first release, to allow apples-to-apples comparisons, we strongly encourage to only release surrogate benchmarks with a version number.
- In order to allow the evaluation of multi-objective NAS methods, we encourage the logging of as many relevant metrics of the evaluated architectures other than accuracy as possible, including training time, number of parameters, and multiply-adds.
- Alongside a released surrogate benchmark, we strongly encourage to release the training data its surrogate(s) were constructed on, as well as the test data used to validate it.
- In order to facilitate checking hypotheses gained using the surrogate benchmarks in real experiments, the complete source code for training the architectures should be open-sourced alongside the repository, allowing to easily go back and forth between querying the model and gathering new data.