How Powerful are Performance Predictors in Neural Architecture Search?

Colin White[1*], **Arber Zela**[2], **Binxin Ru**[3], **Yang Liu**[1], **Frank Hutter**[2,4]
[1] Abacus.AI, [2] University of Freiburg, [3] University of Oxford,
[4] Bosch Center for Artificial Intelligence

Abstract

Early methods in the rapidly developing field of neural architecture search (NAS) required fully training thousands of neural networks. To reduce this extreme computational cost, dozens of techniques have since been proposed to predict the final performance of neural architectures. Despite the success of such performance prediction methods, it is not well-understood how different families of techniques compare to one another, due to the lack of an agreed-upon evaluation metric and optimization for different constraints on the initialization time and query time. In this work, we give the first large-scale study of performance predictors by analyzing 31 techniques ranging from learning curve extrapolation, to weight-sharing, to supervised learning, to zero-cost proxies. We test a number of correlation- and rank-based performance measures in a variety of settings, as well as the ability of each technique to speed up predictor-based NAS frameworks. Our results act as recommendations for the best predictors to use in different settings, and we show that certain families of predictors can be combined to achieve even better predictive power, opening up promising research directions. Our code, featuring a library of 31 performance predictors, is available at `https://github.com/automl/naslib`.

1 Introduction

Neural architecture search (NAS) is a popular area of machine learning, which aims to automate the process of developing neural architectures for a given dataset. Since 2017, a wide variety of NAS techniques have been proposed [78, 45, 32, 49]. While the first NAS techniques trained thousands of architectures to completion and then evaluated the performance using the final validation accuracy [78], modern algorithms use more efficient strategies to estimate the performance of partially-trained or even untrained neural networks [11, 2, 54, 34, 38].

Recently, many performance prediction methods have been proposed based on training a model to predict the final validation accuracy of an architecture just from an encoding of the architecture. Popular choices for these models include Gaussian processes [60, 17, 51], neural networks [36, 54, 65, 69], tree-based methods [33, 55], and so on. However, these methods often require hundreds of fully-trained architectures to be used as training data, thus incurring high initialization time. In contrast, learning curve extrapolation methods [11, 2, 20] need little or no initialization time, but each individual prediction requires partially training the architecture, incurring high query time. Very recently, a few techniques have been introduced which are fast both in query time and initialization time [38, 1], computing predictions based on a single minibatch of data. Finally, using shared weights [45, 4, 32] is a popular paradigm for NAS [73, 25], although the effectiveness of these methods in ranking architectures is disputed [53, 74, 76].

Despite the widespread use of performance predictors, it is not known how methods from different families compare to one another. While there have been some analyses on the best predictors within

[*]{colin, yang}@abacus.ai, {zelaa, fh}@cs.uni-freiburg.de, robin@robots.ox.ac.uk

35th Conference on Neural Information Processing Systems (NeurIPS 2021).

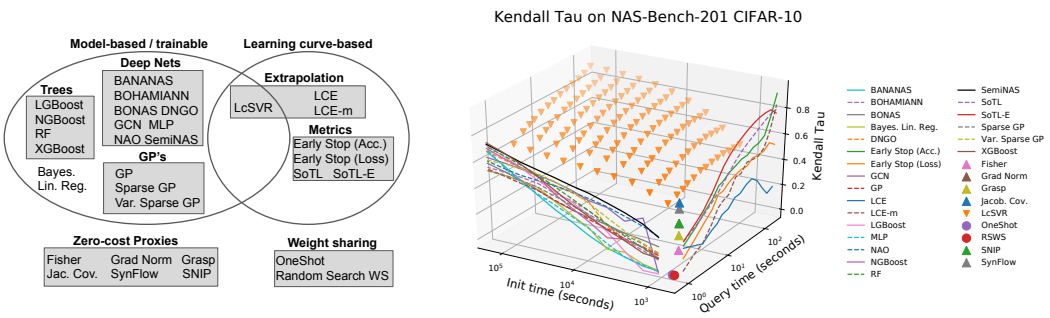

Figure 1: Categories of performance predictors (left). Kendall Tau rank correlation for performance predictors with respect to initialization time and query time (right). Each type of predictor is plotted differently based on whether it allows variable initialization time and/or variable query time. For example, the sixteen model-based predictors have a fixed query time and variable initialization time, so they are plotted as curves parallel to the X-Z plane.

each class [41, 72], for many predictors, the only evaluation is from the original work that proposed the method. Furthermore, no work has previously compared the predictors *across* different families of performance predictors. This leads to two natural questions: how do zero-cost methods, model-based methods, learning curve extrapolation methods, and weight sharing methods compare to one another across different constraints on initialization time and query time? Furthermore, can predictors from different families be combined to achieve even better performance?

In this work, we answer the above questions by giving the first large-scale study of performance predictors for NAS. We study 31 predictors across four popular search spaces and four datasets: NAS-Bench-201 [13] with CIFAR-10, CIFAR-100, and ImageNet16-120, NAS-Bench-101 [71] and DARTS [32] with CIFAR-10, and NAS-Bench-NLP [21] with Penn TreeBank. In order to give a fair comparison among different classes of predictors, we run a full portfolio of experiments, measuring the Pearson correlation and rank correlation metrics (Spearman, Kendall Tau, and sparse Kendall Tau), across a variety of initialization time and query time budgets. We run experiments using a training and test set of architectures generated both uniformly at random, as well as by mutating the highest-performing architectures (the latter potentially more closely resembling distributions encountered during an actual NAS run). Finally, we test the ability of each predictor to speed up NAS algorithms, namely Bayesian optimization [36, 54, 69, 51] and predictor-guided evolution [66, 59].

Since many predictors so far had only been evaluated on one search space, our work shows which predictors have consistent performance across search spaces. Furthermore, by conducting a study with three axes of comparison (see Figure 1), and by comparing various types of predictors, we see a more complete view of the state of performance predictor techniques that leads to interesting insights. Notably, we show that the performance of predictors from different families are complementary and can be combined to achieve significantly higher performance. The success of these experiments opens up promising avenues for future work.

Overall, our experiments bridge multiple areas of NAS research and act as recommendations for the best predictors to use under different runtime constraints. Our code, based on the NASLib library [52], can be used as a testing ground for future performance prediction techniques. In order to ensure reproducibility of the original results, we created a table to clarify which of the 31 predictors had previously published results on a NAS-Bench search space, and how these published results compared to our results (Table 7). We also adhere to the NeurIPS 2021 checklist along with the specialized NAS best practices checklist [31].

Our contributions. We summarize our main contributions below.

- We conduct the first large-scale study of performance predictors for neural architecture search by comparing model-based methods, learning curve extrapolation methods, zero-cost methods, and weight sharing methods across a variety of settings.
- We release a comprehensive library of 31 performance predictors on four different search spaces.
- We show that different families of performance predictors can be combined to achieve substantially better predictive power than any single predictor.

2 Related Work

NAS has been studied since at least the 1990s [19, 58], and has been revitalized in the last few years [78]. While initial techniques focused on reinforcement learning [78, 45] and evolutionary search [37, 49], one-shot NAS algorithms [32, 12, 4] and predictor-based NAS algorithms [65, 54, 69] have recently become popular. We give a brief survey of performance prediction techniques in Section 3. For a survey on NAS, see [15]. The most widely used type of search space in prior work is the cell-based search space [79], where the architecture search is over a relatively small directed acyclic graph representing an architecture.

A few recent works have compared different performance predictors on popular cell-based search spaces for NAS. Siems et al. [55] studied graph neural networks and tree-based methods, and found that gradient-boosted trees and graph isomorphism networks performed the best. However, the comparison was only on a single search space and dataset, and the explicit goal was to achieve maximum performance given a training set of around 60 000 architectures. Another recent paper [41] studied various aspects of supernetwork training, and separately compared four model-based methods: random forest, MLP, LSTM, and GATES [42]. However, the comparisons were again on a single search space and dataset and did not compare between multiple families of performance predictors. Other papers have proposed new model-based predictors and compared the new predictors to other model-based baselines [34, 65, 54, 69]. Finally, a recent paper analyzed training heuristics to make weight-sharing more effective at ranking architectures [72]. To the best of our knowledge, no prior work has conducted comparisons across multiple families of performance predictors.

3 Performance Prediction Methods for NAS

In NAS, given a search space $\mathcal{A}$, the goal is to find $a^* = \text{argmin}_{a \in \mathcal{A}} f(a)$, where f denotes the validation error of architecture a after training on a fixed dataset for a fixed number of epochs E. Since evaluating $f(a)$ typically takes hours (as it requires training a neural network from scratch), many NAS algorithms make use of performance predictors to speed up this process. A *performance predictor* f' is defined generally as any function which predicts the final accuracy or ranking of architectures, without fully training the architectures. That is, evaluating f' should take less time than evaluating f, and $\{f'(a) \mid a \in \mathcal{A}\}$ should ideally have high correlation or rank correlation with $\{f(a) \mid a \in \mathcal{A}\}$.

Each performance predictor is defined by two main routines: an **initialization** routine which performs general pre-computation, and a **query** routine which performs the final architecture-specific computation: it takes as input an architecture specification, and outputs its predicted accuracy. For example, one of the simplest performance predictors is early stopping: for any **query**(a), train a for $E/2$ epochs instead of E [77]. In this case, there is no general pre-computation, so initialization time is zero. On the other hand, the query time for each input architecture is high because it involves training the architecture for $E/2$ epochs. In fact, the runtime of the initialization and query routines varies substantially based on the type of predictor. In the context of NAS algorithms, the initialization routine is typically performed once at the start of the algorithm, and the query routine is typically performed many times throughout the NAS algorithm. Some performance predictors also make use of an **update** routine, when part of the computation from initialization needs to be updated without running the full procedure again (for example, in a NAS algorithm, a model may be updated periodically based on newly trained architectures). Now we give an overview of the main families of predictors. See Figure 1 (left) for a taxonomy of performance predictors.

Model-based (trainable) methods. The most common type of predictor, the model-based predictor, is based on supervised learning. The initialization routine consists of fully training many architectures (i.e., evaluating $f(a)$ for many architectures $a \in \mathcal{A}$) to build a training set of datapoints $\{a, f(a)\}$. Then a model f' is trained to predict $f(a)$ given a. While the initialization time for model-based predictors is very high, the query time typically takes less than a second, which allows thousands of predictions to be made throughout a NAS algorithm. The model is also updated regularly based on the new datapoints. These predictors are typically used within BO frameworks [36, 54], evolutionary frameworks [66], or by themselves [67], to perform NAS. Popular choices for the model include tree-based methods (where the features are the adjacency matrix representation of

the architectures) [33, 55], graph neural networks [36, 54], Gaussian processes [47, 51], and neural networks based on specialized encodings of the architecture [69, 42].

Learning curve-based methods. Another family predicts the final performance of architectures using only a partially trained network, by extrapolating the learning curve. This is accomplished by fitting the partial learning curve to an ensemble of parametric models [11], or by simply summing the training losses observed so far [50]. Early stopping as described earlier is also a learning curve-based method. Learning curve methods do not require any initialization time, yet the query time typically takes minutes or hours, which is orders of magnitude slower than the query time in model-based methods. Learning curve-based methods can be used in conjunction with multi-fidelty algorithms, such as Hyperband or BOHB [27, 16, 24].

Hybrid methods. Some predictors are hybrids between learning curve and model-based methods. These predictors train a model at initialization time to predict $f(a)$ given both a and a partial learning curve of a as features. Models in prior work include an SVR [2], or a Bayesian neural network [20]. Although the query time and initialization time are both high, hybrid predictors tend to have strong performance.

Zero-cost methods. Another class of predictors have no initialization time and very short query times (so-called "zero-cost" methods). These predictors compute statistics from just a single forward/backward propagation pass for a single minibatch of data, by computing the correlation of activations within a network [38], or by adapting saliency metrics proposed in pruning-at-initialization literatures [23, 1]. Similar to learning curve-based methods, since the only computation is specific to each architecture, the initialization time is zero. Zero-cost methods have recently been used to warm start NAS algorithms [1].

Weight sharing methods. Weight sharing [45] is a popular approach to substantially speed up NAS, especially in conjunction with a one-shot algorithm [32, 12]. In this approach, all architectures in the search space are combined to form a single over-parameterized supernetwork. By training the weights of the supernetwork, all architectures in the search space can be evaluated quickly using this set of weights. To this end, the supernetwork can be used as a performance predictor. This results in NAS algorithms [32, 28] which are significantly faster than sequential NAS algorithms, such as evolution or Bayesian optimization. Recent work has shown that although the shared weights are sometimes not effective at ranking architectures [53, 74, 76], one-shot NAS techniques using shared weights still achieve strong performance [73, 25].

Tradeoff between intialization and query time. The main families mentioned above all have different initialization and query times. The tradeoffs between initialization time, query time, and performance depend on a few factors such as the type of NAS algorithm and its total runtime budget, and different settings are needed in different situations. For example, if there are many architectures whose performance we want to estimate, then we should have a low query time, and if we have a high total runtime budget, then we can afford a high initialization time. We may also change our runtime budget throughout the run of a single NAS algorithm. For example, at the start of a NAS algorithm, we may want to have coarse estimates of a large number of architectures (low initialization time, low query time such as zero-cost predictors). As the NAS algorithm progresses, it is more desirable to receive higher-fidelity predictions on a smaller set of architectures (model-based or hybrid predictors). The exact budgets depend on the type of NAS algorithm.

Choice of performance predictors. We analyze 31 performance predictors defined in prior work: BANANAS [69], Bayesian Linear Regression [6], BOHAMIANN [57], BONAS [54], DNGO [56], Early Stopping with Val. Acc. (e.g. [77, 27, 16, 79]) Early Stopping with Val. Loss. [50], Fisher [1], Gaussian Process (GP) [48], GCN [75], Grad Norm [1], Grasp [64], Jacobian Covariance [38], LCE [11], LCE-m [20], LcSVR [2], LGBoost/GBDT [33], MLP [69], NAO [35], NGBoost [55], OneShot [73], Random Forest (RF) [55], Random Search with Weight Sharing (RSWS) [26], SemiNAS [34], SNIP [23], SoTL [50], SoTL-E [50], Sparse GP [3], SynFlow [61], Variational Sparse GP [63], and XGBoost [55]. For any method that did not have an architecture encoding already defined (such as the tree-based methods, GP-based methods, and Bayesian Linear Regression), we use the standard adjacency matrix encoding, which consists of the adjacency matrix of the architecture along with a one-hot list of the operations [71, 68]. By open-sourcing our code, we encourage

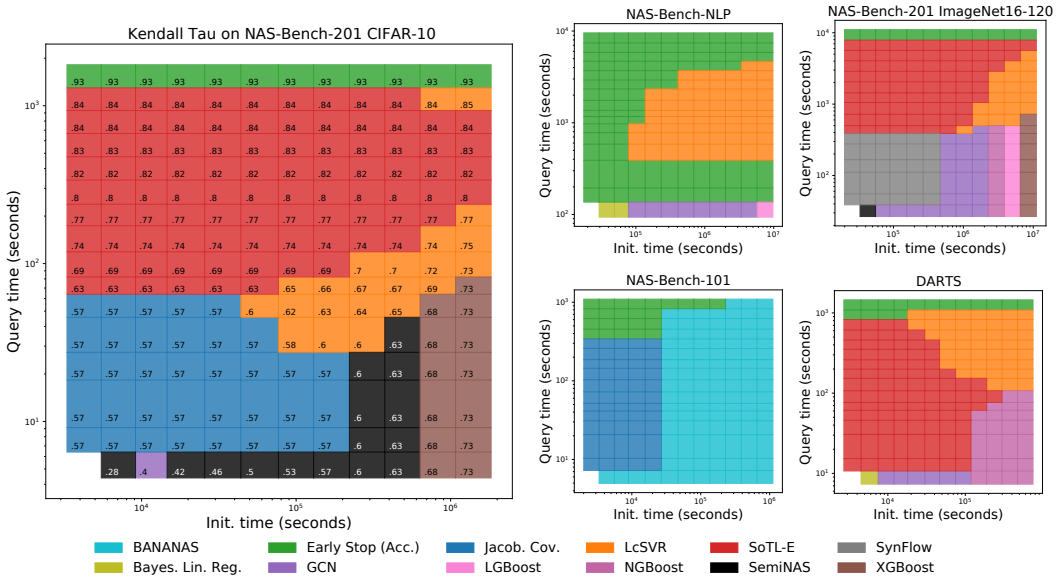

Figure 2: The performance predictors with the highest Kendall Tau values for all initialization time and query time budgets on NAS-Bench-201, NAS-Bench-101, NAS-Bench-NLP and DARTS. For example, on NAS-Bench-201 CIFAR-10 (left) with an initialization time of 10^6 seconds and query time of 10 seconds, XGBoost achieves a Kendall Tau value of .73 which is the highest value out of the 31 predictors that we tested at that budget.

implementing more (existing and future) performance predictors which can then be compared to the 31 which we focus on in this work. In Section B.1, we give descriptions and detailed implementation details for each performance predictor. In Section D, we give a table that describes for which predictors we were able to reproduce published results, and for which predictors it is not possible (e.g., since some predictors were released before the creation of NAS benchmarks).

4 Experiments

We now discuss our experimental setup and results. We discuss reproducibility in Sections A and D, and our code (based on the NASLib library [52]) is available at `https://github.com/automl/naslib`. We split up our experiments into two categories: evaluating the performance of each predictor with respect to various correlation metrics (Section 4.1), and evaluating the ability of each predictor to speed up predictor-based NAS algorithms (Section 4.2). We start by describing the four NAS benchmarks used in our experiments.

NAS benchmark datasets. NAS-Bench-101 [71] consists of over 423 000 unique neural architectures with precomputed training, validation, and test accuracies after training for 4, 12, 36, and 108 epochs on CIFAR-10 [71]. The cell-based search space consists of five nodes which can take on any directed acyclic graph (DAG) structure, and each node can be one of three operations. Since learning curve information is only available at four epochs, it is not possible to run most learning curve extrapolation methods on NAS-Bench-101. NAS-Bench-201 [13] consists of 15 625 architectures (out of which 6 466 are unique after removing isomorphisms [13]). Each architecture has full learning curve information for training, validation, and test losses/accuracies for 200 epochs on CIFAR-10 [22], CIFAR-100, and ImageNet-16-120 [10]. The search space consists of a cell which is a complete DAG with 4 nodes. Each edge can take one of five different operations. The DARTS search space [32] is significantly larger with roughly 10^{18} architectures. The search space consists of two cells, each with seven nodes. The first two nodes are inputs from previous layers, and the intermediate four nodes can take on any DAG structure such that each node has two incident edges. The last node is the output node. Each edge can take one of eight operations. In our experiments, we make use of the training data from NAS-Bench-301 [55], which consists of 23 000 architectures drawn uniformly at random and trained on CIFAR-10 for 100 epochs. Finally, the NAS-Bench-NLP search space [21] is even

larger, at 10^{53} LSTM-like cells, each with at most 25 nodes in any DAG structure. Each cell can take one of seven operations. In our experiments, we use the NAS-Bench-NLP dataset, which consists of $14\,000$ architectures drawn uniformly at random and trained on Penn Tree Bank [40] for 50 epochs.

Hyperparameter tuning. Although we used the code directly from the original repositories (sometimes making changes when necessary to adapt to NAS-Bench search spaces), the predictors had significantly different levels of hyperparameter tuning. For example, some of the predictors had undergone heavy hyperparameter tuning on the DARTS search space (used in NAS-Bench-301), while other predictors (particularly those from 2017 or earlier) had never been run on cell-based search spaces. Furthermore, most predictor-based NAS algorithms can utilize cross-validation to tune the predictor periodically throughout the NAS algorithm. This is because the bottleneck for predictor-based NAS algorithms is typically the training of architectures, not fitting the predictor [56, 30, 16]. Therefore, it is fairer and also more informative to compare performance predictors which have had the same level of hyperparameter tuning through cross-validation. For each search space, we run random search on each performance predictor for 5000 iterations, with a maximum total runtime of 15 minutes. The final evaluation uses a separate test set. The hyperparameter value ranges for each predictor can be found in Section B.2.

4.1 Performance Predictor Evaluation

We evaluate each predictor based on three axes of comparison: initialization time, query time, and performance. We measured performance with respect to several different metrics: Pearson correlation and three different rank correlation metrics (Spearman, Kendall Tau, and sparse Kendall Tau [72, 55]). The experimental setup is as follows: the predictors are tested with 11 different initialization time budgets and 14 different query time budgets, leading to a total of 154 settings. On NAS-Bench-201 CIFAR-10, the 11 initialization time budgets are spaced logarithmically from 1 second to 1.8×10^7 seconds on a 1080 Ti GPU (which corresponds to training 1000 random architectures on average) which is consistent with experiments conducted in prior work [65, 69, 34]. For other search spaces, these times are adjusted based on the average time to train 1000 architectures. The 14 query time budgets are spaced logarithmically from 1 second to 1.8×10^4 seconds (which corresponds to training an architecture for 199 epochs). These times are adjusted for other search spaces based on the training time and different number of epochs. Once the predictor is initialized, we draw a test set of 200 architectures uniformly at random from the search space. For each architecture in the test set, the predictor uses the specified query time budget to make a prediction. We then evaluate the quality of the predictions using the metrics described above. We average the results over 100 trials for each (initialization time, query time) pair.

Results and discussion. Figure 1 shows a full three-dimensional plot for NAS-Bench-201 on CIFAR-10 over initialization time, query time, and Kendall Tau rank correlation. Of the 31 predictors we tested, we found that just seven of them are Pareto-optimal with respect to Kendall Tau, initialization time, and query time. That is, only seven algorithms have the highest Kendall Tau value for at least one of the 154 query time/initialization time budgets on NAS-Bench-201 CIFAR-10. This can be seen more clearly in Figure 2 (left), which is a view from above Figure 1: each lattice point displays the predictor with the highest Kendall Tau value for the corresponding budget. In Figure 2 (right), we plot the Pareto-optimal predictors for five different dataset/search space combinations. In Section B.3, we give the full 3D plots and report the variance across trials for each method. In Figure 4 (left), we also plot the Pearson and Spearman correlation coefficients for NAS-Bench-201 CIFAR-10. The trends between these measures are largely the same, although we see that SemiNAS performs better on the rank-based metrics. For the rest of this section, we focus on the popular Kendall Tau metric, giving the full results for the other metrics in Section B.3.

We see similar trends across DARTS and the two NAS-Bench-201 datasets. NAS-Bench-NLP also has fairly similar trends, although early stopping performs comparatively stronger. NAS-Bench-101 is different from the other search spaces both in terms of the topology and the benchmark itself, which we discuss later in this section.

In the low initialization time, low query time region, Jacobian covariance or SynFlow perform well across NAS-Bench-101 and NAS-Bench-201. However, none of the six zero-cost methods perform well on the larger DARTS search space. Weight sharing (which also has low initialization and low query time, as seen in Figure 1), did not yield high Kendall Tau values for these search spaces, either,

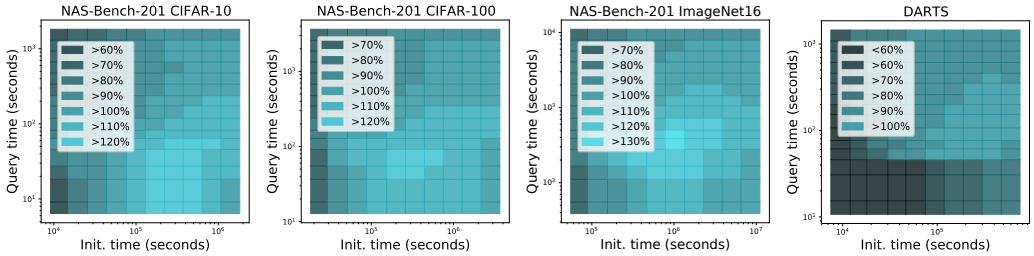

Figure 3: Percentage of OMNI's Kendall Tau value compared to the next-best predictors for each budget constraint.

which is consistent with recent work [53, 74, 76]. However, rank correlation is not as crucial to one-shot NAS algorithms as it is for black box predictor-based methods, as demonstrated by prior one-shot NAS methods that do perform well [25, 73, 32, 28, 12]. In the low initialization time, high query time region, sum of training losses (SoTL-E) consistently performed the best, outperforming other learning-curve based methods.

The high initialization time, low query time region (especially the bottom row of the plots, corresponding to a query time of 1 second) is by far the most competitive region in the recent NAS literature. Sixteen of the 31 predictors had query times under one second, because many NAS algorithms are designed to initialize (and continually update) performance predictors that are used to quickly query thousands of candidate architectures. GCN and SemiNAS, the specialized GCN/semi-supervised methods, perform especially well in the first half of this critical region, when the initialization time is relatively low. However, boosted tree methods actually performed best in the second half of the critical region where the initialization time is high, which is consistent with prior work [33, 55]. Recall that for model-based methods, the initialization time corresponds to training architectures to be used as training data for the performance predictor. Therefore, our results suggest that techniques which can extract better latent features of the architectures can make up for a small training dataset, but methods based purely on performance data work better when there is enough such data.

Perhaps the most interesting finding is that on NAS-Bench-101/201, SynFlow and Jacobian covariance, which take three seconds each to compute, both outperform all model-based methods even after *30 hours* of initialization. Put another way, NAS algorithms that make use of model-based predictors may be able to see substantial improvements by using Jacobian covariance instead of a model-based predictor in the early iterations.

The Omnipotent Predictor. One conclusion from Figure 2 is that different types of predictors are specialized for specific initialization time and query time constraints. A natural follow-up question is whether different families are complementary and can be combined to achieve stronger performance. In this section, we run a proof-of-concept to answer this question. We combine the best-performing predictors from three different families in a simple way: the best learning curve method (SoTL-E), and the best zero-cost method (Jacobian covariance), are used as additional input features for a model-based predictor (we separately test SemiNAS and NGBoost). We call this method OMNI, the omnipotent predictor. We give results in Figure 3 and pseudo-code as well as additional experiments in Section B.4. In contrast to all other predictors, the performance of OMNI is strong across almost all budget constraints and search spaces. In some settings, OMNI achieves a Kendall Tau value 30% higher than the next-best predictors.

The success of OMNI verifies that the information learned by different families of predictors are complementary: the information learned by extrapolating a learning curve, by computing a zero-cost proxy, and by encoding the architecture, all improve performance. We further confirm this by running an ablation study for OMNI in Section B.4. We can hypothesize that each predictor type measures distinct quantities: SOTL-E measures the training speed, zero-cost predictors measure the covariance between activations on different datapoints, and model-based predictors simply learn patterns between the architecture encodings and the validation accuracies. Finally, while we showed a proof-of-concept, there are several promising areas for future work such as creating ensembles of the model-based approaches, combining zero-cost methods with model-based methods in more sophisticated ways, and giving a full quantification of the correlation among different families of predictors.

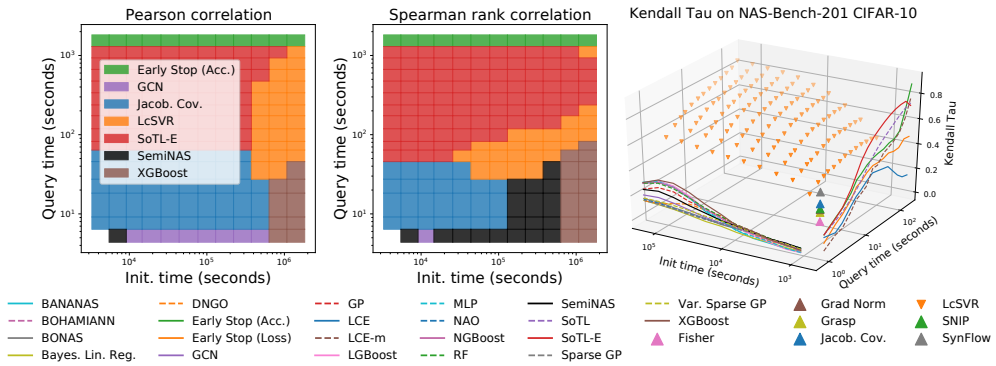

Figure 4: The best predictors on NAS-Bench-201 CIFAR-10 with respect to Pearson (left) and Spearman (middle). Kendall Tau values from a mutation-based training and test set (right).

NAS-Bench-101: a more complex search space. In Figure 2, the plot for NAS-Bench-101 looks significantly different than the plots for the other search spaces, for two reasons. The first reason is a technical one: the NAS-Bench-101 API only gives the validation accuracy at four epochs, and does not give the training loss for any epochs. Therefore, we could not implement SoTL or any learning curve extrapolation method. However, all sixteen of the model-based predictors were implemented on NAS-Bench-101. In this case, BANANAS significantly outperformed the next-best predictors (GCN and SemiNAS) across every initialization time. One explanation is due to the complexity of the NAS-Bench-101 search space: while all NAS-Bench-201 architectures have the same graph topology and DARTS architectures' nodes have exactly two incoming edges, the NAS-Bench-101 search space is much more diverse with architectures ranging from a single node and no edges, to five nodes with nine connecting edges. In fact, the architecture encoding used in BANANAS, the path encoding, was designed specifically to deal with the complexity of the NAS-Bench-101 search space (replacing the standard adjacency matrix encoding). To test this explanation, in Appendix B we run several of the simpler tree-based and GP-based predictors using the path encoding, and we see that these methods now surpass BANANAS in performance.

A mutation-based test set. The results from Figure 2 used a test set drawn uniformly at random from the search space (and the training set used by model-based predictors was also drawn uniformly at random). However, neighborhood-based NAS algorithms such as local search, regularized evolution, and some versions of Bayesian optimization consider architectures which are local perturbations of the architectures encountered so far. Therefore, the predictors used in these NAS algorithms must be able to distinguish architectures which are local mutations of a small set of seed architectures.

We run an experiment in which the test set is created by mutating architectures from an initial set of seed architectures. Specifically, we draw a set of 50 random architectures and choose the five with the highest validation accuracy as seed architectures. Then we create a set of 200 test architectures by randomly mutating up to three attributes of the seed architectures. Therefore, all architectures in the test set are at most an edit distance of three from a seed architecture, where two architectures are a single edit distance away if they differ by one operation or edge.

We create the training set by randomly choosing architectures from the test set and mutating one random attribute. As in all of our experiments, we ensure that the training set and test set are disjoint. In Figure 4 (right), we plot the correlation results for NAS-Bench-201 CIFAR-10. While the zero-cost and learning curve-based approaches have similar performance, the model-based approaches have significantly worse performance compared to the uniform random setting. This is because the average edit distance between architectures in the test set is low, making it significantly harder for model-based predictors to distinguish the performance of these architectures, even when using a training set that is based on mutations of the test set. In fact, interestingly, the performance of many model-based approaches starts to perform worse after 10^6 seconds. SemiNAS in particular performs much worse in this setting, and boosted trees have comparatively stronger performance in this setting.

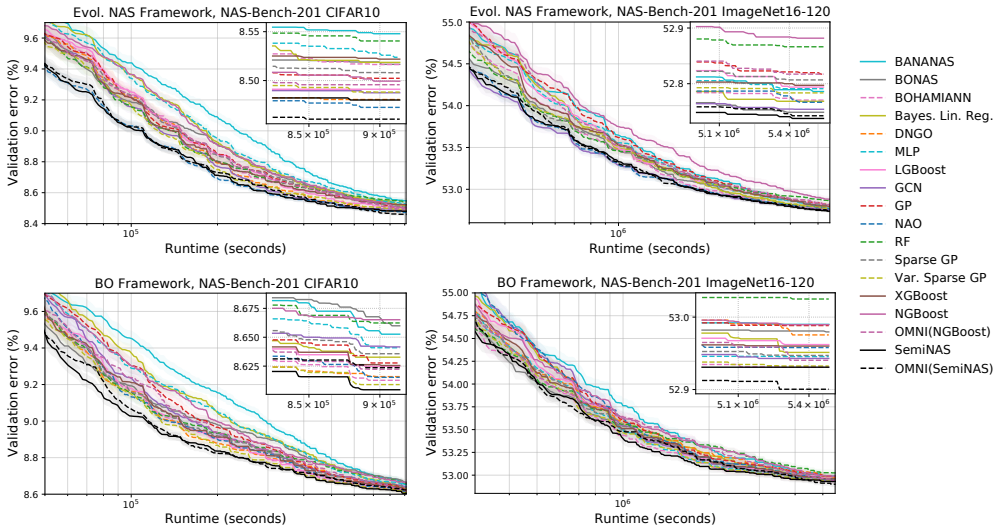

Figure 5: Validation error vs. runtime for the predictor-guided evolution framework and the Bayesian optimization + predictor framework using different predictors.

4.2 Predictor-Based NAS Experiments

Now we evaluate the ability of each model-based performance predictor to speed up NAS. We use two popular predictor-based NAS methods: the predictor-guided evolution framework [66, 59], and the Bayesian optimization + predictor framework [17, 36, 54]. The predictor-guided evolution framework is an iterative procedure in which the best architectures in the current population are mutated to create a set of candidate architectures. A predictor (trained on the entire population) chooses k architectures which are then evaluated. In our experiments, the candidate pool is created by mutating the top five architectures 40 times each, and we set $k = 20$. For each predictor, we run predictor-guided evolution for 25 iterations and average the results over 100 trials. The BO + predictor framework is similar to the evolution framework, but an ensemble of three performance predictors are used so that uncertainty estimates for each prediction can be computed. In each iteration, the candidate architectures whose predictions maximize an acquisition function are then evaluated. Similar to prior work, we use independent Thompson sampling [69], as the acquisition function, and an ensemble is created by using a different ordering of the training set and different random weight initializations (if applicable) of the same predictor. In each iteration, the top 20 architectures are chosen from a randomly sampled pool of 200 architectures.

In Figure 5, we present results for both NAS frameworks on NAS-Bench-201 CIFAR-10 and ImageNet16-120 for the 16 model-based predictors. We also test OMNI, using its lowest query time setting (consisting of NGBoost + Jacobian covariance), and another version of OMNI that replaces NGBoost with SemiNAS. Our results show that the model-based predictors with the top Kendall Tau rank correlations in the low query time region from Figure 2 also roughly achieve the best performance when applied for NAS: SemiNAS and NAO perform the best for shorter runtime, and boosted trees perform best for longer runtime. OMNI(NGBoost) consistently outperforms NGBoost, and OMNI(SemiNAS) often achieves top performance. This suggests that using zero-cost methods in conjunction with model-based methods is a promising direction for future study.

4.3 So, how powerful are performance predictors?

Throughout Section 4, we tested performance predictors in a variety of settings, by varying the search spaces, datasets, runtime budgets, and training/test distributions. We saw largely the same trends among all of our experiments. Interesting findings included the success of zero-cost predictors even when compared to model-based predictors and learning curve extrapolation predictors with longer runtime budgets, and the fact that information from different families of predictors are complementary. When choosing a performance predictor for new applications, we recommend deciding on a target initialization time and query time budget, consulting Figures 2 and 6, and then combining the best

predictors from the desired runtime setting, similar to OMNI. For example, if a performance predictor with medium initialization time and low runtime is desired for a search space similar to NAS-Bench-201 or DARTS, we recommend using NGBoost with Jacobian covariance and SynFlow as additional features.

5 Societal Impact

Our hope is that our work will have a positive impact on the AutoML community by making it quicker and easier to develop and fairly compare performance predictors. For example, AutoML practitioners can consult our experiments to more easily decide on the performance prediction methods best suited to their application, rather than conducting computationally intensive experiments of their own [43]. Furthermore, AutoML researchers can use our library to develop new performance prediction techniques and compare new methods to 31 other algorithms across four search spaces. Since the topic of this work is AutoML, it is a level of abstraction away from real applications. This work may be used to improve deep learning applications, both beneficial (e.g. reducing CO_2 emissions), or harmful (e.g. creating language models with heavy bias) to society.

6 Conclusions and Limitations

In this work, we gave the first large-scale study of performance predictors for neural architecture search. We compared 31 different performance predictors, including learning curve extrapolation methods, weight sharing methods, zero-cost methods, and model-based methods. We tested the performance of the predictors in a variety of settings and with respect to different metrics. Although we ran experiments on four different search spaces, it will be interesting to extend our experiments to even more machine learning tasks beyond image classification and language modeling.

Our new predictor, OMNI, is the first predictor to combine complementary information from three families of performance preditors, leading to substantially improved performance. While the simplicity of OMNI is appealing, it also opens up new directions for future work by combining different predictors in more sophisticated ways. To facilitate follow-up work, we release our code featuring a library of performance predictors. Our goal is for our repository to grow over time as it is used by the community, so that experiments in our library can be even more comprehensive.

Acknowledgments and Disclosure of Funding

This work was done while CW and YL were employed at Abacus.AI. AZ and FH acknowledge support by the European Research Council (ERC) under the European Union Horizon 2020 research and innovation programme through grant no. 716721, and by BMBF grant DeToL. BR was supported by the Clarendon Fund of University of Oxford.

References

[1] Mohamed S Abdelfattah, Abhinav Mehrotra, Łukasz Dudziak, and Nicholas Donald Lane. Zero-cost proxies for lightweight nas. In *Proceedings of the International Conference on Learning Representations (ICLR)*, 2021.

[2] Bowen Baker, Otkrist Gupta, Ramesh Raskar, and Nikhil Naik. Accelerating neural architecture search using performance prediction. *arXiv preprint arXiv:1705.10823*, 2017.

[3] Matthias Bauer, Mark van der Wilk, and Carl Edward Rasmussen. Understanding probabilistic sparse gaussian process approximations. *arXiv preprint arXiv:1606.04820*, 2016.

[4] Gabriel Bender, Pieter-Jan Kindermans, Barret Zoph, Vijay Vasudevan, and Quoc Le. Understanding and simplifying one-shot architecture search. In *Proceedings of the International Conference on Machine Learning (ICML)*, pages 550–559, 2018.

[5] Eli Bingham, Jonathan P Chen, Martin Jankowiak, Fritz Obermeyer, Neeraj Pradhan, Theofanis Karaletsos, Rohit Singh, Paul Szerlip, Paul Horsfall, and Noah D Goodman. Pyro: Deep universal probabilistic programming. *The Journal of Machine Learning Research*, 20(1):973–978, 2019.

[6] Christopher M Bishop. *Pattern recognition and machine learning*. springer, 2006.

[7] Han Cai, Ligeng Zhu, and Song Han. Proxylessnas: Direct neural architecture search on target task and hardware. *Proceedings of the International Conference on Learning Representations (ICLR)*, 2019.

[8] Akshay Chandrashekaran and Ian R Lane. Speeding up hyper-parameter optimization by extrapolation of learning curves using previous builds. In *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*, pages 477–492. Springer, 2017.

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

[10] Patryk Chrabaszcz, Ilya Loshchilov, and Frank Hutter. A downsampled variant of imagenet as an alternative to the CIFAR datasets. *CoRR*, abs/1707.08819, 2017.

[11] Tobias Domhan, Jost Tobias Springenberg, and Frank Hutter. Speeding up automatic hyperparameter optimization of deep neural networks by extrapolation of learning curves. In *Twenty-Fourth International Joint Conference on Artificial Intelligence*, 2015.

[12] Xuanyi Dong and Yi Yang. Searching for a robust neural architecture in four gpu hours. In *Proceedings of the IEEE Conference on computer vision and pattern recognition*, pages 1761–1770, 2019.

[13] Xuanyi Dong and Yi Yang. Nas-bench-201: Extending the scope of reproducible neural architecture search. In *Proceedings of the International Conference on Learning Representations (ICLR)*, 2020.

[14] Tony Duan, Avati Anand, Daisy Yi Ding, Khanh K Thai, Sanjay Basu, Andrew Ng, and Alejandro Schuler. Ngboost: Natural gradient boosting for probabilistic prediction. In *International Conference on Machine Learning*, pages 2690–2700. PMLR, 2020.

[15] Thomas Elsken, Jan Hendrik Metzen, and Frank Hutter. Neural architecture search: A survey. *arXiv preprint arXiv:1808.05377*, 2018.

[16] Stefan Falkner, Aaron Klein, and Frank Hutter. Bohb: Robust and efficient hyperparameter optimization at scale. In *Proceedings of the International Conference on Machine Learning (ICML)*, 2018.

[17] Kirthevasan Kandasamy, Willie Neiswanger, Jeff Schneider, Barnabas Poczos, and Eric P Xing. Neural architecture search with Bayesian optimisation and optimal transport. In *Advances in Neural Information Processing Systems*, pages 2016–2025, 2018.

[18] Guolin Ke, Qi Meng, Thomas Finley, Taifeng Wang, Wei Chen, Weidong Ma, Qiwei Ye, and Tie-Yan Liu. Lightgbm: A highly efficient gradient boosting decision tree. *Advances in neural information processing systems*, 30:3146–3154, 2017.

[19] Hiroaki Kitano. Designing neural networks using genetic algorithms with graph generation system. *Complex systems*, 4(4):461–476, 1990.

[20] Aaron Klein, Stefan Falkner, Jost Tobias Springenberg, and Frank Hutter. Learning curve prediction with Bayesian neural networks. *ICLR 2017*, 2017.

[21] Nikita Klyuchnikov, Ilya Trofimov, Ekaterina Artemova, Mikhail Salnikov, Maxim Fedorov, and Evgeny Burnaev. Nas-bench-nlp: neural architecture search benchmark for natural language processing. *arXiv preprint arXiv:2006.07116*, 2020.

[22] Alex Krizhevsky. Learning multiple layers of features from tiny images. Technical report, University of Toronto, 2009.

[23] Namhoon Lee, Thalaiyasingam Ajanthan, and Philip Torr. SNIP: Single-shot network pruning based on connection sensitivity. In *Proceedings of the International Conference on Learning Representations (ICLR)*, 2019.

[24] Liam Li, Kevin Jamieson, Afshin Rostamizadeh, Ekaterina Gonina, Moritz Hardt, Benjamin Recht, and Ameet Talwalkar. A system for massively parallel hyperparameter tuning. In *Proceedings of the Conference on Machine Learning Systems*, 2020.

[25] Liam Li, Mikhail Khodak, Maria-Florina Balcan, and Ameet Talwalkar. Geometry-aware gradient algorithms for neural architecture search. In *Proceedings of the International Conference on Learning Representations (ICLR)*, 2021.

[26] Liam Li and Ameet Talwalkar. Random search and reproducibility for neural architecture search. *arXiv preprint arXiv:1902.07638*, 2019.

[27] Lisha Li, Kevin Jamieson, Giulia DeSalvo, Afshin Rostamizadeh, and Ameet Talwalkar. Hyperband: A novel bandit-based approach to hyperparameter optimization. *arXiv preprint arXiv:1603.06560*, 2016.

[28] Hanwen Liang, Shifeng Zhang, Jiacheng Sun, Xingqiu He, Weiran Huang, Kechen Zhuang, and Zhenguo Li. Darts+: Improved differentiable architecture search with early stopping. *arXiv preprint arXiv:1909.06035*, 2019.

[29] Andy Liaw, Matthew Wiener, et al. Classification and regression by randomforest. *R news*, 2(3):18–22, 2002.

[30] Marius Lindauer, Katharina Eggensperger, Matthias Feurer, Stefan Falkner, André Biedenkapp, and Frank Hutter. Smac v3: Algorithm configuration in python. *URL https://github.com/automl/SMAC3*, 2017.

[31] Marius Lindauer and Frank Hutter. Best practices for scientific research on neural architecture search. *arXiv preprint arXiv:1909.02453*, 2019.

[32] Hanxiao Liu, Karen Simonyan, and Yiming Yang. Darts: Differentiable architecture search. *arXiv preprint arXiv:1806.09055*, 2018.

[33] Renqian Luo, Xu Tan, Rui Wang, Tao Qin, Enhong Chen, and Tie-Yan Liu. Neural architecture search with gbdt. *arXiv preprint arXiv:2007.04785*, 2020.

[34] Renqian Luo, Xu Tan, Rui Wang, Tao Qin, Enhong Chen, and Tie-Yan Liu. Semi-supervised neural architecture search. In *Proceedings of the Annual Conference on Neural Information Processing Systems (NeurIPS)*, 2020.

[35] Renqian Luo, Fei Tian, Tao Qin, Enhong Chen, and Tie-Yan Liu. Neural architecture optimization. In *Proceedings of the Annual Conference on Neural Information Processing Systems (NeurIPS)*, 2018.

[36] Lizheng Ma, Jiaxu Cui, and Bo Yang. Deep neural architecture search with deep graph Bayesian optimization. In *2019 IEEE/WIC/ACM International Conference on Web Intelligence (WI)*, pages 500–507. IEEE, 2019.

[37] Krzysztof Maziarz, Andrey Khorlin, Quentin de Laroussilhe, and Andrea Gesmundo. Evolutionary-neural hybrid agents for architecture search. *arXiv preprint arXiv:1811.09828*, 2018.

[38] Joseph Mellor, Jack Turner, Amos Storkey, and Elliot J Crowley. Neural architecture search without training. *arXiv preprint arXiv:2006.04647*, 2020.

[39] Szymon Jakub Mikler. Snip: Single-shot network pruning based on connection sensitivity. *GitHub repository gahaalt/SNIP-pruning*, 2019.

[40] Tomáš Mikolov, Martin Karafiát, Lukáš Burget, Jan Černockỳ, and Sanjeev Khudanpur. Recurrent neural network based language model. In *Annual conference of the international speech communication association*, 2010.

[41] Xuefei Ning, Wenshuo Li, Zixuan Zhou, Tianchen Zhao, Yin Zheng, Shuang Liang, Huazhong Yang, and Yu Wang. A surgery of the neural architecture evaluators. *arXiv preprint arXiv:2008.03064*, 2020.

[42] Xuefei Ning, Yin Zheng, Tianchen Zhao, Yu Wang, and Huazhong Yang. A generic graph-based neural architecture encoding scheme for predictor-based nas. *arXiv preprint arXiv:2004.01899*, 2020.

[43] David Patterson, Joseph Gonzalez, Quoc Le, Chen Liang, Lluis-Miquel Munguia, Daniel Rothchild, David So, Maud Texier, and Jeff Dean. Carbon emissions and large neural network training. *arXiv preprint arXiv:2104.10350*, 2021.

[44] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, et al. Scikit-learn: Machine learning in python. *the Journal of machine Learning research*, 12:2825–2830, 2011.

[45] Hieu Pham, Melody Y Guan, Barret Zoph, Quoc V Le, and Jeff Dean. Efficient neural architecture search via parameter sharing. *arXiv preprint arXiv:1802.03268*, 2018.

[46] Joaquin Quiñonero Candela and Carl Edward Rasmussen. A unifying view of sparse approximate gaussian process regression. *J. Mach. Learn. Res.*, 6:1939–1959, December 2005.

[47] Carl Edward Rasmussen. Gaussian processes in machine learning. In *Summer School on Machine Learning*, pages 63–71. Springer, 2003.

[48] Carl Edward Rasmussen. Gaussian processes in machine learning. In *Summer School on Machine Learning*, pages 63–71. Springer, 2003.

[49] Esteban Real, Alok Aggarwal, Yanping Huang, and Quoc V Le. Regularized evolution for image classifier architecture search. In *Proceedings of the aaai conference on artificial intelligence*, volume 33, pages 4780–4789, 2019.

[50] Binxin Ru, Clare Lyle, Lisa Schut, Mark van der Wilk, and Yarin Gal. Revisiting the train loss: an efficient performance estimator for neural architecture search. *arXiv preprint arXiv:2006.04492*, 2020.

[51] Binxin Ru, Xingchen Wan, Xiaowen Dong, and Michael Osborne. Neural architecture search using Bayesian optimisation with weisfeiler-lehman kernel. *arXiv preprint arXiv:2006.07556*, 2020.

[52] Michael Ruchte, Arber Zela, Julien Siems, Josif Grabocka, and Frank Hutter. Naslib: a modular and flexible neural architecture search library, 2020.

[53] Christian Sciuto, Kaicheng Yu, Martin Jaggi, Claudiu Musat, and Mathieu Salzmann. Evaluating the search phase of neural architecture search. *arXiv preprint arXiv:1902.08142*, 2019.

[54] Han Shi, Renjie Pi, Hang Xu, Zhenguo Li, James Kwok, and Tong Zhang. Bridging the gap between sample-based and one-shot neural architecture search with bonas. *Advances in Neural Information Processing Systems*, 33, 2020.

[55] Julien Siems, Lucas Zimmer, Arber Zela, Jovita Lukasik, Margret Keuper, and Frank Hutter. Nas-bench-301 and the case for surrogate benchmarks for neural architecture search. *arXiv preprint arXiv:2008.09777*, 2020.

[56] Jasper Snoek, Oren Rippel, Kevin Swersky, Ryan Kiros, Nadathur Satish, Narayanan Sundaram, Mostofa Patwary, Mr Prabhat, and Ryan Adams. Scalable Bayesian optimization using deep neural networks. In *International conference on machine learning*, pages 2171–2180, 2015.

[57] Jost Tobias Springenberg, Aaron Klein, Stefan Falkner, and Frank Hutter. Bayesian optimization with robust Bayesian neural networks. In *Advances in Neural Information Processing Systems*, pages 4134–4142, 2016.

[58] Kenneth O Stanley and Risto Miikkulainen. Evolving neural networks through augmenting topologies. *Evolutionary computation*, 10(2):99–127, 2002.

[59] Yanan Sun, Xian Sun, Yuhan Fang, and Gary Yen. A new training protocol for performance predictors of evolutionary neural architecture search algorithms. *arXiv preprint arXiv:2008.13187*, 2020.

[60] Kevin Swersky, David Duvenaud, Jasper Snoek, Frank Hutter, and Michael Osborne. Raiders of the lost architecture: Kernels for Bayesian optimization in conditional parameter spaces. In *NIPS workshop on Bayesian Optimization in Theory and Practice*, December 2013.

[61] Hidenori Tanaka, Daniel Kunin, Daniel LK Yamins, and Surya Ganguli. Pruning neural networks without any data by iteratively conserving synaptic flow. *arXiv preprint arXiv:2006.05467*, 2020.

[62] Lucas Theis, Iryna Korshunova, Alykhan Tejani, and Ferenc Huszár. Faster gaze prediction with dense networks and fisher pruning. *arXiv preprint arXiv:1801.05787*, 2018.

[63] Michalis Titsias. Variational learning of inducing variables in sparse gaussian processes. In *Artificial Intelligence and Statistics*, pages 567–574, 2009.

[64] Chaoqi Wang, Guodong Zhang, and Roger Grosse. Picking winning tickets before training by preserving gradient flow. In *Proceedings of the International Conference on Learning Representations (ICLR)*, 2019.

[65] Linnan Wang, Yiyang Zhao, Yuu Jinnai, and Rodrigo Fonseca. Alphax: exploring neural architectures with deep neural networks and monte carlo tree search. *arXiv preprint arXiv:1805.07440*, 2018.

[66] Chen Wei, Chuang Niu, Yiping Tang, and Jimin Liang. Npenas: Neural predictor guided evolution for neural architecture search. *arXiv preprint arXiv:2003.12857*, 2020.

[67] Wei Wen, Hanxiao Liu, Hai Li, Yiran Chen, Gabriel Bender, and Pieter-Jan Kindermans. Neural predictor for neural architecture search. *arXiv preprint arXiv:1912.00848*, 2019.

[68] Colin White, Willie Neiswanger, Sam Nolen, and Yash Savani. A study on encodings for neural architecture search. In *Proceedings of the Annual Conference on Neural Information Processing Systems (NeurIPS)*, 2020.

[69] Colin White, Willie Neiswanger, and Yash Savani. Bananas: Bayesian optimization with neural architectures for neural architecture search. In *Proceedings of the AAAI Conference on Artificial Intelligence*, 2021.

[70] Antoine Yang, Pedro M Esperança, and Fabio M Carlucci. Nas evaluation is frustratingly hard. In *Proceedings of the International Conference on Learning Representations (ICLR)*, 2020.

[71] Chris Ying, Aaron Klein, Esteban Real, Eric Christiansen, Kevin Murphy, and Frank Hutter. Nas-bench-101: Towards reproducible neural architecture search. *arXiv preprint arXiv:1902.09635*, 2019.

[72] Kaicheng Yu, Rene Ranftl, and Mathieu Salzmann. How to train your super-net: An analysis of training heuristics in weight-sharing nas. *arXiv preprint arXiv:2003.04276*, 2020.

[73] Arber Zela, Thomas Elsken, Tonmoy Saikia, Yassine Marrakchi, Thomas Brox, and Frank Hutter. Understanding and robustifying differentiable architecture search. In *Proceedings of the International Conference on Learning Representations (ICLR)*, 2020.

[74] Arber Zela, Julien Siems, and Frank Hutter. Nas-bench-1shot1: Benchmarking and dissecting one-shot neural architecture search. In *Proceedings of the International Conference on Learning Representations (ICLR)*, 2020.

[75] Yuge Zhang. Neural predictor for neural architecture search. *GitHub repository ultmaster/neuralpredictor.pytorch*, 2020.

[76] Yuge Zhang, Zejun Lin, Junyang Jiang, Quanlu Zhang, Yujing Wang, Hui Xue, Chen Zhang, and Yaming Yang. Deeper insights into weight sharing in neural architecture search. *arXiv preprint arXiv:2001.01431*, 2020.

[77] Dongzhan Zhou, Xinchi Zhou, Wenwei Zhang, Chen Change Loy, Shuai Yi, Xuesen Zhang, and Wanli Ouyang. Econas: Finding proxies for economical neural architecture search. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 11396–11404, 2020.

[78] Barret Zoph and Quoc V. Le. Neural architecture search with reinforcement learning. In *Proceedings of the International Conference on Learning Representations (ICLR)*, 2017.

[79] Barret Zoph, Vijay Vasudevan, Jonathon Shlens, and Quoc V Le. Learning transferable architectures for scalable image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 8697–8710, 2018.

Checklist

1. For all authors...
   (a) Do the main claims made in the abstract and introduction accurately reflect the paper's contributions and scope? [Yes] Our abstract and introduction accurately reflect our paper.
   (b) Did you describe the limitations of your work? [Yes] See Section 6.
   (c) Did you discuss any potential negative societal impacts of your work? [Yes] See Section 5.
   (d) Have you read the ethics review guidelines and ensured that your paper conforms to them? [Yes] We discuss the ethics guidelines in Section 5.
2. If you are including theoretical results...
   (a) Did you state the full set of assumptions of all theoretical results? [N/A] We did not include theoretical results.
   (b) Did you include complete proofs of all theoretical results? [N/A] We did not include theoretical results.
3. If you ran experiments...
   (a) Did you include the code, data, and instructions needed to reproduce the main experimental results (either in the supplemental material or as a URL)? [Yes] We included all code, data, and instructions in the supplementary material.
   (b) Did you specify all the training details (e.g., data splits, hyperparameters, how they were chosen)? [Yes] All training details are specified in the supplementary material.
   (c) Did you report error bars (e.g., with respect to the random seed after running experiments multiple times)? [Yes] We ran 100 trials for all experiments. Our plots that are performance vs. runtime (e.g. Figure 5) have error bars. For our Pareto-optimality plots (e.g. Figure 7), see the supplementary material for the error bars for all 31 predictors.
   (d) Did you include the total amount of compute and the type of resources used (e.g., type of GPUs, internal cluster, or cloud provider)? [Yes] We report the compute time and resources used in Section 4 and in the supplementary material.
4. If you are using existing assets (e.g., code, data, models) or curating/releasing new assets...
   (a) If your work uses existing assets, did you cite the creators? [Yes] We cite the creators of all code, data, and models used.
   (b) Did you mention the license of the assets? [Yes] We mention the licenses in the supplementary material.
   (c) Did you include any new assets either in the supplemental material or as a URL? [N/A] We do not include new assets.
   (d) Did you discuss whether and how consent was obtained from people whose data you're using/curating? [Yes] We did not use or release any datasets with personal data.
   (e) Did you discuss whether the data you are using/curating contains personally identifiable information or offensive content? [N/A] The data we are using does not contain personal information or offensive content.
5. If you used crowdsourcing or conducted research with human subjects...
   (a) Did you include the full text of instructions given to participants and screenshots, if applicable? [N/A] We did not run experiments with human subjects.
   (b) Did you describe any potential participant risks, with links to Institutional Review Board (IRB) approvals, if applicable? [N/A] We did not run experiments with human subjects.
   (c) Did you include the estimated hourly wage paid to participants and the total amount spent on participant compensation? [N/A] We did not run experiments with human subjects.

A NAS Research checklist

There have been a few recent works which have called for improving the reproducibility and fairness in experimental comparisons in NAS research [26, 71, 70]. This led to the release of a NAS best practices checklist [31]. We address each part of the checklist.

1. **Best Practices for Releasing Code**

   For all experiments you report:

   (a) Did you release code for the training pipeline used to evaluate the final architectures? [Yes] We used publicly available NAS benchmarks: NAS-Bench-101, NAS-Bench-201, DARTS/NAS-Bench-301, and NAS-Bench-NLP, so we did not train the architectures ourselves.

   (b) Did you release code for the search space [Yes] Since we used NAS benchmarks, this code is already publicly available.

   (c) Did you release the hyperparameters used for the final evaluation pipeline, as well as random seeds? [Yes] Since we used NAS benchmarks, the final training pipeline was fixed. We released our code, which includes the seeds used.

   (d) Did you release code for your NAS method? [Yes] All of our code is available at `https://github.com/automl/naslib`.

   (e) Did you release hyperparameters for your NAS method, as well as random seeds? [Yes] The hyperparameters are given in Appendix Table 2. We ran 100 trials (random seeds) for each experiment, and we released the code to launch these experiments is in our repository.

2. **Best practices for comparing NAS methods**

   (a) For all NAS methods you compare, did you use exactly the same NAS benchmark, including the same dataset (with the same training-test split), search space and code for training the architectures and hyperparameters for that code? [Yes] We only used NAS benchmarks, which means the training details are fixed.

   (b) Did you control for confounding factors (different hardware, versions of DL libraries, different runtimes for the different methods)? [Yes] We only used NAS Benchmarks, which keep these details fixed.

   (c) Did you run ablation studies? [Yes] Our abalation study for OMNI is in Appendix B.4.

   (d) Did you use the same evaluation protocol for the methods being compared? [Yes] We only used NAS Benchmarks, which keep this fixed.

   (e) Did you compare performance over time? [Yes] Our experiments compare performance over initialization time and query time.

   (f) Did you compare to random search? [No] We did not compare to random search, although we used other baselines for performance predictors such as random forests and simple MLPs.

   (g) Did you perform multiple runs of your experiments and report seeds? [Yes] We ran 100 trials of each experiment, and we released the code to launch these experiments is in our repository.

   (h) Did you use tabular or surrogate benchmarks for in-depth evaluations? [Yes] We only used tabular and surrogate benchmarks.

3. **Best practices for reporting important details**

   (a) Did you report how you tuned hyperparameters, and what time and resources this required? [Yes] We reported this information in Section 4.

   (b) Did you report the time for the entire end-to-end NAS method (rather than, e.g., only for the search phase)? [Yes] We present results along three axes: performance, initialization time, and query time.

   (c) Did you report all the details of your experimental setup? [Yes] We did include all details of the setup in Section 4 and Appendix B.

B Details from Section 4 (Experiments)

In this section, we give more details from Section 4, and we present more experiments. In Section B.1, we give a short description and implementation details for all 31 predictors. We also mention the licenses for the datasets we use. Next, in Section B.2, we give the details for hyperparameter tuning. Then in Section B.3, we give detailed experimental results for all search spaces. After that, in Section B.4, we give pseudo-code and an ablation study for OMNI.

B.1 Descriptions and Implementation Details

We describe all 31 methods that we used.

- **BANANAS.** The BANANAS [69] predictor is a model-based predictor consisting of an ensemble of three MLPs. Architectures are encoded using the *path encoding*, which encodes each possible path from the input to the output of the cell as a bit. We use the code from the original repository, but with PyTorch for the MLPs instead of Tensorflow.
- **Bayesian Linear Regression** Bayesian Linear Regression [6] is one of the simplest Bayesian methods, which assumes that the samples are normally distributed, and assumes that the output labels are a linear function of the input features (the architecture encoding in our case). Therefore, the predictions are sampled from a distribution computed in closed form given the observations, the posterior of the linear model parameters and the observation noise. We use the implementation from the `pybnn` [2] package and the one-hot adjacency matrix encoding.
- **BOHAMIANN.** BOHAMIANN [57] utilizes Bayesian inference via stochastic gradient Hamiltonian Monte Carlo (SGHMC) in order to sample from the estimated posterior of a Bayesian Neural Network. We use the original implementation from the `pybnn` package and the one-hot adjacency matrix encoding.
- **BONAS.** Bayesian optimization for NAS [54] is a NAS algorithm which makes use of a graph convolutional network (GCN) as a model-based predictor within an outer loop of Bayesian optimization. In this work, we refer to "BONAS" as the GCN predictor. The implementation from the original paper works on a constrained version of the DARTS search space where the normal cell and the reduction cell have the same architecture. We adapted BONAS for the three search spaces using the original encoding style. Specifically, given that the DARTS search space includes both normal and reduction cells, we encoded both in one adjacency matrix by arranging the two cells' adjacency matrices diagonally and zero-padding the rest.
- **DNGO.** Deep Networks for Global Optimization (DNGO) is an implementation of Bayesian optimization using adaptive basis regression using neural networks instead of Gaussian processes to avoid the cubic scaling [56]. We use the adaptive basis regressor as a model-based predictor, using the original code from the `pybnn` package and the one-hot adjacency matrix encoding.
- **Early Stopping with Val. Acc.** Early Stopping using validation accuracy has been considered in NAS many times (e.g. [77, 27, 16, 79]). It uses the validation accuracy from the most recent epoch trained so far as a proxy for architecture performance.
- **Early Stopping with Val. Loss** Early stopping with validation loss uses the validation loss instead of the validation accuracy [50].
- **Fisher.** Fisher [1] is a zero-cost predictor which computes the sum over all gradients of the activations in a neural network. Fisher builds off of prior work on channel pruning at initialization [62]. We used the implementation from [1].
- **GCN.** The GCN approach for NAS has also been studied by [67]. Although the code was never released, we used an unofficial implementation online from [75]. The encoding strategy follows that of BONAS.
- **GP.** Gaussian Process (GP) [48] is a simple model where every finite number of random variables have a joint Gaussian distribution. It is commonly used as the default model choice

[2] `https://github.com/automl/pybnn`

in Bayesian optimization, and it is fully specified by its mean and covariance functions. The runtime of GPs is cubic in the number of datapoints, because the covariance matrix must be inverted to compute the predictive distribution is computed. We use the `Pyro` implementation [5] and the one-hot adjacency matrix encoding.

- **Grad Norm.** Grad Norm [1] is a zero-cost predictor which sums the Euclidean norm of the gradients of one minibatch of training data. It was used by [1] as a baseline when comparing other zero-cost methods. We used the implementation from [1].
- **Grasp.** Grasp [64] was introduced as a technique to prune network weights based on a saliency metric at initialisation. It improved over SNIP [23] by approximating the change in gradient norm. Later, it was used as a zero-cost predictor in NAS by [1]. We used the implementation from [1].
- **Jacobian covariance.** Jacobian covariance [38] is a zero-cost method which measures the modelling flexibility of a network based on the covariance of its prediction Jacobians with respect to different image inputs. [38] claims that architectures with more flexible prediction at initialization tend to have better test performance after training. We use the original code.
- **LCE.** Learning curve extrapolation (LCE) [11] takes in a partial learning curve as input, and then extrapolates the curve to a chosen epoch. It works by fitting the curve to several parametric models, and choosing the best model using MCMC. We use the original code, but we used a subset of the original parametric models which we found to improve performance.
- **LCE-m.** LCE-m, introduced as a baseline by [20] is an extrapolation method similar to LCE but with a modified loss function that drops LCE's original mechanism of biasing the search to never underestimate the accuracy at the asymptote of the curve. The list of parametric models was also partially changed. Note that this is not to be confused with the learning curve *prediction* method from [20]. We used the original code, but we used a subset of the parametric models which we found to improve performance.
- **LcSVR.** This is a hybrid predictor, which extrapolates the learning curves using a trained ν-SVR (LcSVR) [2]. Specifically, the ν-SVR model takes in a partial learning curve as well as their first and second derivatives, and the training hyperparameters as the inputs. Like other model-based predictors, the ν-SVR model must be trained by using fully evaluated architectures as training data, and like other learning-curve-based predictors, each new query requires partially training the architecture. We use the original code.
- **LGBoost.** Light Gradient Boosting Machine (LightGBM or LGBoost) [18] was designed to be a more lightweight gradient boosting implementation. LGBoost has been used as a model-based predictor both in NAS algorithms [33] and in the creation of NAS-Bench-301 [55]. We followed the implementation of [33] and used the one-hot adjacency matrix encoding. We reduced the minimum number of data points in a leaf to 5 so that LGBoost could run with smaller training set sizes.
- **MLP.** The multilayer perceptron (MLP) predictor is a model-based predictor consisting of a fully-connected neural network, which has been used by prior work as a baseline for comparisons [69, 65]. We use the implementation in BANANAS, in which the default network has width 10 and depth 20. To encode the architecture, we used a one-hot encoding of the adjacecy matrix and list of operations, as in most prior work in NAS [71].
- **NAO.** Neural Architecture Optimization makes use of an encoder-decoder [35]. The original neural architecture is mapped to a continuous representation via an LSTM encoder network. Performance prediction is powered by a feedfoward neural network, and the decoder is built with an LSTM and attention mechanisms. We used the implementation from SemiNAS [34].
- **NGBoost.** Natural gradient boosting (NGBoost) [14] is another gradient-boosted method that uses natural gradients in order to enable uncertainty estimates of the predictions. It has been studied as a model-based predictor in the creation of NAS-Bench-301 [55]. We used the original code and the one-hot adjacency matrix encoding.
- **OneShot.** We derive the OneShot predictor following a similar procedure as in prior work [73]. We first train the weight-sharing model (supernetwork) with normal SGD training. The number of epochs and number of training examples are picked based on a grid search optimizing the Spearman rank correlation on a validation set. After the weight-sharing network is trained, we use it as a predictor for the performance of an architecture by

computing the performance on the validation set using only the subpath in the supernetwork corresponding to that architecture. The rest of the supernetwork is zeroed out.

- **Random Forest.** Random forests [29] consist of ensembles of decision trees. Random forests have been studied as model-based predictors in the creation of NAS-Bench-301 [55]. We use the Scikit-learn implementation [44] and the one-hot adjacency matrix encoding.
- **RSWS.** Random Search with Weight Sharing (RSWS) [26] is used exactly the same as the OneShot predictor when predicting the performance of an architecture. The only difference is the way RSWS optimizes the weight-sharing model. Instead of training it as a single network, at each mini-batch iteration, RSWS uniformly samples one architecture from the search space and updates the weights of *only* that operations corresponding to that architecture in the supernetwork.
- **SemiNAS.** Semi-supervised NAS (SemiNAS) uses semi-supervised learning with the NAO architecture [34]. Specifically, additional synthetic training data is generated and used to train the architecture. The performance of synthetic training data is predicted by the NAO predictor. We use the original implementation, reducing the ratio of real vs. synthetic data to 1:1 and decreasing the number of epochs to 100 to decrease its extreme training time. SemiNAS was originally implemented for NAS-Bench-101 and ProxylessNAS [7] search spaces. We adapted it for NAS-Bench-201 and DARTS using the original encoding style. Encoder/decoder lengths and vocabulary sizes of the NAO predictor were changed accordingly to fit the new search spaces.
- **SNIP.** Single-shot network pruning (SNIP) was first proposed in [23] as a technique to prune network weights at initialisation. SNIP was later adapted by [1] to become a zero-cost predictor for ranking architecture performance. We used the implementation from [39].
- **SoTL.** Sum of training losses (SoTL) [50] is a learning curve-based predictor, which estimates the generalization performance of architectures based on the sum of their training losses over the epochs trained so far. SoTL does not attempt to predict the final validation accuracy as in model-based predictors, but does output a score with high rank correlation with respect to the final validation accuracy. We use the original implementation of SoTL [50].
- **SoTL-E.** Sum of training losses at last epoch E (SoTL-E) [50] is very similar to SoTL, but it only considers the sum over the training mini-batches in the most recent epoch trained so far.
- **Sparse GP.** Compared to classical GPs, Sparse Gaussian Processes (Sparse GPs) [46] scale better to large amounts of data by introducing the so-called inducing variables to summarize the training data. To bypass the expensive marginal likelihood estimation, variational inference is used in order to approximate the posterior distribution. We use the `Pyro` implementation [5] and the one-hot adjacency matrix encoding.
- **SynFlow.** Synaptic Flow (SynFlow) [61] was introduced as a technique to prune network weights at initialisation based on a saliency metric. It improved over SNIP [23] and Grasp [64] by avoiding layer collapse when performing parameter pruning by taking a product of all parameters in the network. It was used as a zero-cost predictor in NAS by [1]. We used the implementation from [1].
- **Variational Sparse GP.** Variational Sparse Gaussian Process (Var. Sparse GP) [63] is similar to the Sparse GP, but it can handle non-Gaussian likelihoods. We use the `Pyro` implementation [5] and the one-hot adjacency matrix encoding.
- **XGBoost.** eXtreme Gradient Boosting (XGBoost) [9] is the first of three gradient-boosted decision tree implementations that we used. XGBoost has been used as a model-based predictor in the creation of NAS-Bench-301 [55]. We used the original code and the one-hot adjacency matrix encoding.

In Table 1, we discuss the licenses for the NAS datasets we used.

B.2 Hyperparameter Tuning

Now we give the details of hyperparameter tuning. Recall from Section 4 that we run cross-validation on all model-based predictors that we studied, for two reasons. First, we compared 16 model-based predictors directly from the original repositories when possible, but the published hyperparameters

Table 1: Licenses for the datasets that we use.

Dataset	License	URL
NAS-Bench-101	Apache 2.0	https://github.com/google-research/nasbench
NAS-Bench-201	MIT	https://github.com/D-X-Y/NAS-Bench-201
NAS-Bench-301	Apache 2.0	https://github.com/automl/nasbench301
NAS-Bench-NLP	None	https://github.com/fmsnew/nas-bench-nlp-release

from different methods have significantly different levels of hyperparameter tuning. Therefore, running more hyperparameter tuning for all predictors will help to level the playing field. Second, most predictor-based NAS algorithms can naturally utilize cross-validation during the search to improve performance. This is because the bottleneck for predictor-based NAS algorithms is typically the training of architectures, not fitting the predictor [56, 30, 16]. Therefore, adding cross-validation to model-based predictors is a more realistic setting. Note that the same is not true for other families of performance predictors. For example, LCE methods are typically used to replace fully training architectures during NAS, therefore we would not know the final validation accuracy of these architectures and would not be able to run cross-validation.

Overall, our goal was to run lightweight cross-validation to level the playing field. For each model-based predictor, we chose 3-5 hyperparameters and chose ranges based on their default values from their original repositories. See Table 2. Note that some methods such as the three GP-based methods and the three methods from `pybnn` (Bayes. Lin. Reg., BOHAMIANN, DNGO) already had cross-validation built in, so we excluded these. In all of our experiments, we ran random search on each performance predictor for 5000 iterations, with a maximum total runtime of 15 minutes.

B.3 Additional Experiments

In Figure 2, we plotted the predictors which are Pareto-optimal for each initialization and query time budget. In this section, we present the complete results, including 3D plots for all search spaces, as well as separate plots for initialization time and query time vs. Kendall Tau which include the standard deviation of the results from 100 trials. See Figure 6. We put in a reasonable effort to implement all 31 predictors for all search spaces, however, a few of these combinations are omitted. For example, running learning curve methods on NAS-Bench-101 would require training the architectures from scratch, since NAS-Bench-101 does not include full learning curve information.

In general, in Figure 6 we see that the relative performance of the predictors are largely the same across the three datasets from NAS-Bench-201. However, there are clear differences across NAS-Bench-201 CIFAR-10, NAS-Bench-101, and DARTS, even though all three use CIFAR-10 as the dataset. For example, for high initialization time, BANANAS exhibits the highest rank correlation on NAS-Bench-101 but the worst rank correlation on DARTS. As explained in Section 4, this is largely due to the strength of the path encoding specifically on NAS-Bench-101. However, the path encoding does not scale as well on larger search spaces such as DARTS.

In Figure 7, we plot the predictors which are Pareto-optimal for each initialization and query time budget, for three metrics: Pearson correlation, Spearman rank correlation, and sparse Kendall Tau. We also include Kendall Tau for completeness (which we had already presented in Figure 2).

Since Figures 6 and 7 take up one page each already, we plot the results for the final search space, NAS-Bench-NLP, in Figure 8. We see largely the same trends on this repository as well, with a few differences. Since the NAS-Bench-NLP code is written in a much earlier version of PyTorch (1.1), we were unable to implement the zero-cost predictors in NAS-Bench-NLP. Furthermore, the SOTL and SOTL-E predictors do not perform as well as Early Stop (Acc.) for NAS-Bench-NLP. We see that LcSVR performs particularly well on NAS-Bench-NLP (as was also the case with DARTS). This may be because the large size of the search space gives a bigger benefit for hybrid methods which include LCE components, over model-based methods alone.

Finally, recall that in Section 4, we discussed the differences between NAS-Bench-101 and the other search spaces. We mentioned one technical reason (the NAS-Bench-101 API only gives validation accuracies at four epochs, and does not give the training loss for any epochs) and one reason based on the differences in the search space itself: the NAS-Bench-101 search space is more complex than

Table 2: Hyperparameters of the model-based methods and their default values from their original repositories.

Model	Hyperparameter	Range	Log-transform	Default Value
BANANAS	Num. layers	[5, 25]	false	20
	Layer width	[5, 25]	false	20
	Learning rate	[0.0001, 0.1]	true	0.001
BONAS	Num. layers	[16, 128]	true	64
	Batch size	[32, 256]	true	128
	Learning rate	[0.00001, 0.1]	true	0.0001
GCN	Num. layers	[64, 200]	true	144
	Batch size	[5, 32]	true	7
	Learning rate	[0.00001, 0.1]	true	0.0001
	Weight decay	[0.00001, 0.1]	true	0.0003
LCSVR	Penalty param.	[0.00001, 10]	true	-
	Kernel coefficient	[0.00001, 10]	false	-
	Frac. support vectors	[0, 1]	false	-
LGBoost	Num. leaves	[10, 100]	false	31
	Learning rate	[0.001, 0.1]	true	0.05
	Feature fraction	[0.1, 1]	false	0.9
MLP	Num. layers	[5, 25]	false	20
	Layer width	[5, 25]	false	20
	Learning rate	[0.0001, 0.1]	true	0.001
NAO	Num. layers	[16, 128]	true	64
	Batch size	[32, 256]	true	100
	Learning rate	[0.00001, 0.1]	true	0.001
NGBoost	Num. estimators	[128, 512]	true	505
	Learning rate	[0.001, 0.1]	true	0.081
	Max depth	[1, 25]	false	6
	Max features	[0.1, 1]	false	0.79
RF	Num. estimators	[16, 128]	true	116
	Max features	[0.1, 0.9]	true	0.17
	Min samples (leaf)	[1, 20]	false	2
	Min samples (split)	[2, 20]	true	2
SemiNAS	Num. layers	[16, 128]	true	64
	Batch size	[32, 256]	true	100
	Learning rate	[0.00001, 0.1]	true	0.001
XGBoost	Max depth	[1, 15]	false	6
	Min child weight	[1, 10]	false	1
	Col sample (tree)	[0, 1]	false	1
	Learning rate	[0.001, 0.5]	true	0.3
	Col sample (level)	[0, 1]	false	1

NAS-Bench-201 and DARTS because it allows any graph topology. Therefore, the path encoding is particularly well-suited for NAS-Bench-101. To test this explanation, in Figure 9 we run several of the simpler tree-based and GP-based predictors using the path encoding, and we see that these methods now surpass BANANAS.

B.4 OMNI Details and Ablation

In this section, we present more details and experiments for OMNI. Recall that OMNI combines strong predictors from three different families: SoTL-E, Jacobian covariance, and either NGBoost or SemiNAS, from the families of learning curve methods, zero-cost methods, and model-based methods, respectively. See Algorithm 1 for pseudo-code. Recall from Section 3 that performance

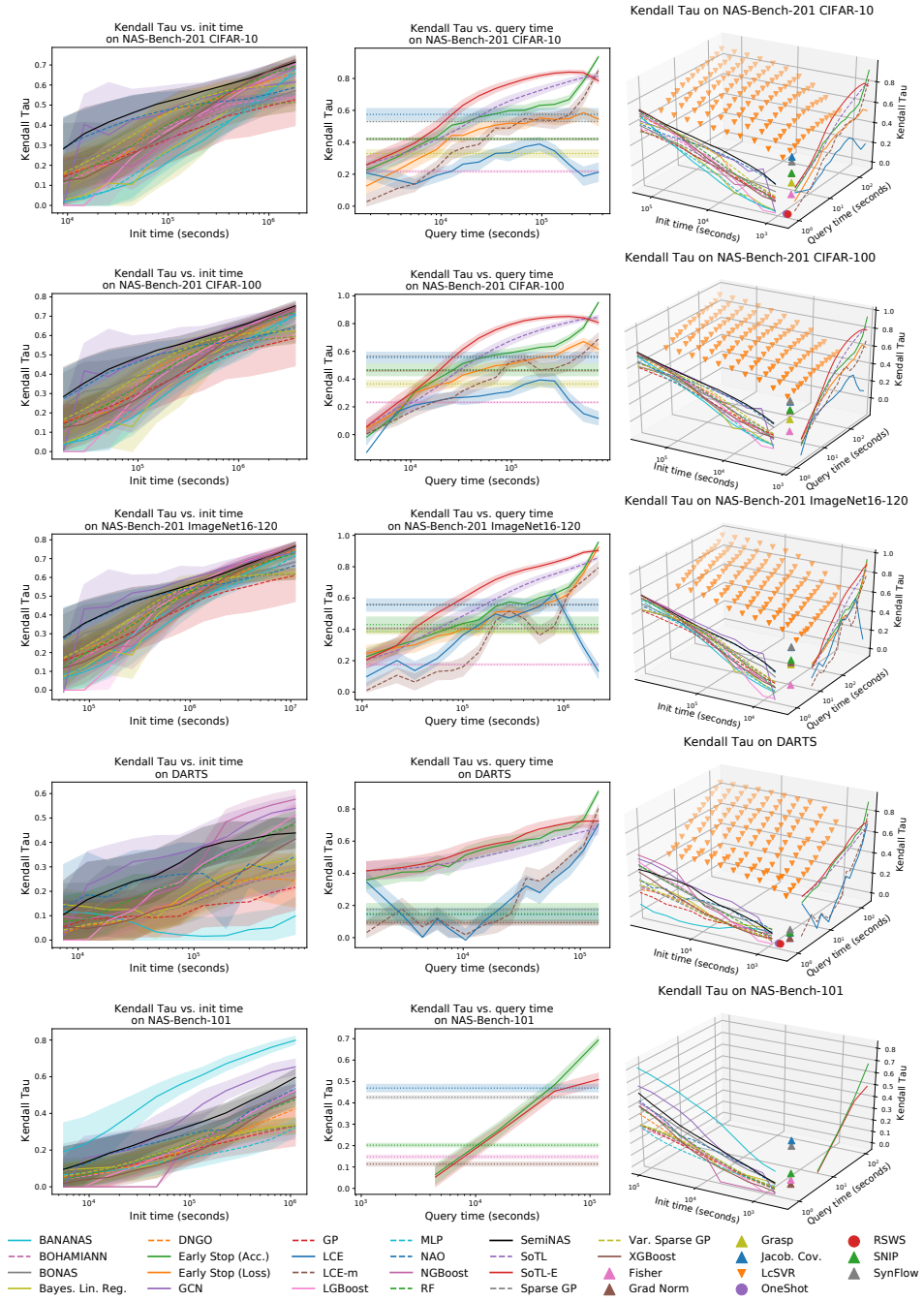

Figure 6: Full results for Kendall Tau with standard deviations shaded.

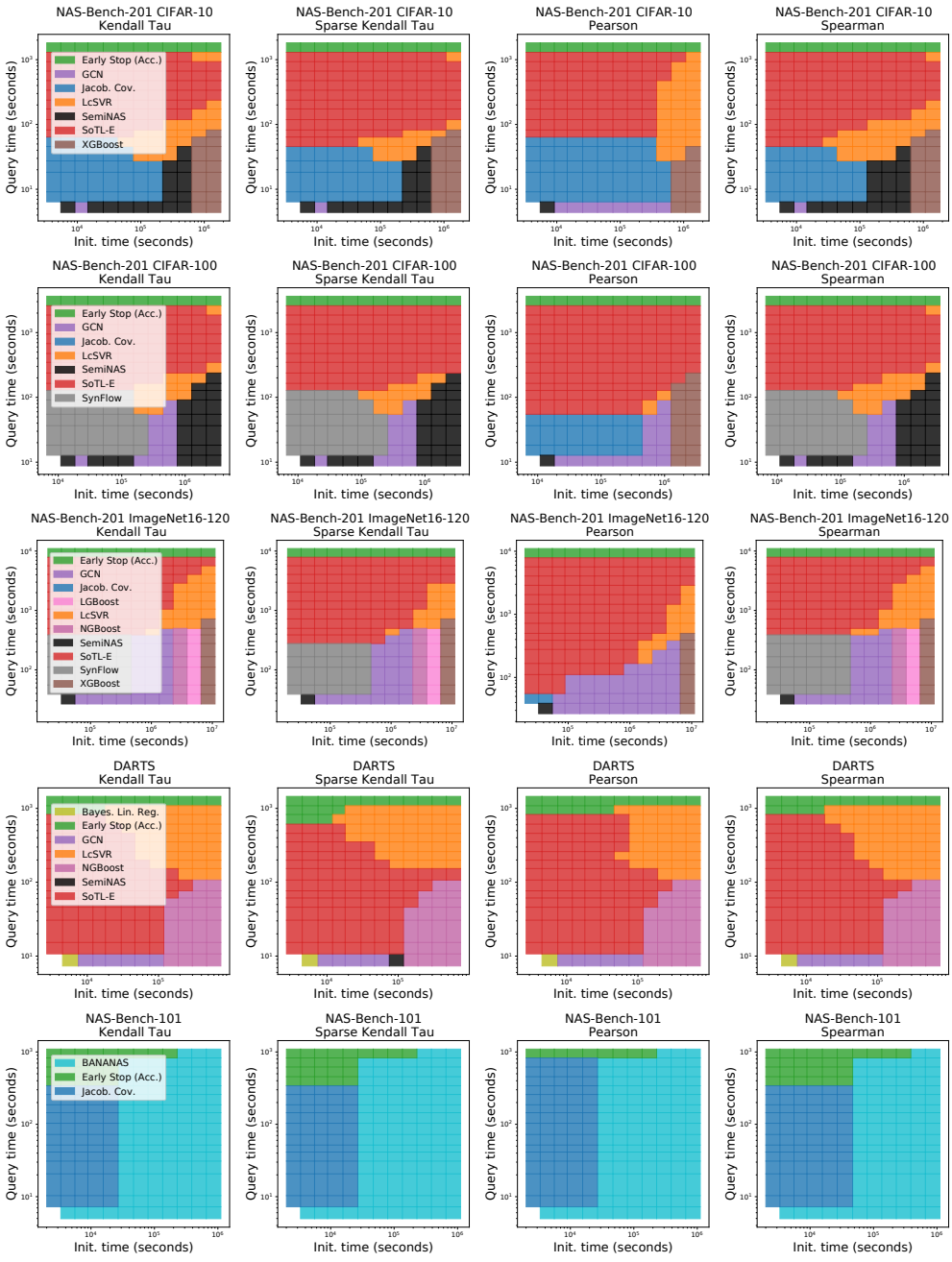

Figure 7: The performance predictors with the highest metrics for all initialization time and query time budgets and search spaces. The first column is repeated from Figure 2.

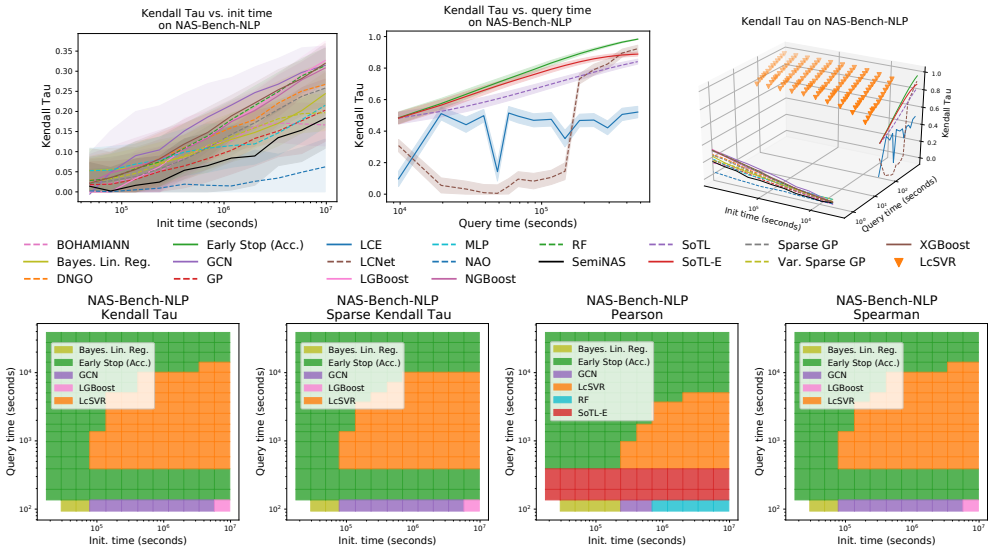

Figure 8: Full results for NAS-Bench-NLP with standard deviations shaded (top row). The performance predictors with the highest metrics for all initialization time and query time budgets on NAS-Bench-NLP (bottom row).

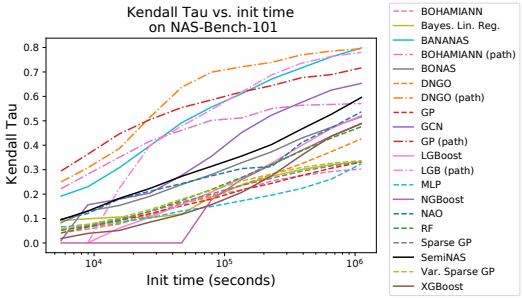

Figure 9: Kendall Tau rank correlation for predictors on NAS-Bench-101. The GP-based and tree-based methods are not competitive until used with the path encoding.

predictors have an initialization stage that is called once, and a query stage that is called many times throughout the NAS algorithm.

Now we give an ablation study for OMNI. We consider four different versions of OMNI: (NGBoost + Jacob. Cov.), (SoTL-E + NGBoost), (Jacob. Cov. + SoTL-E), and (NGBoost + Jacob. Cov. + SoTL-E). Note that Jacob. Cov. + SoTL-E is created by using Jacob. Cov. and SoTL-E as features in NGBoost without the architecture encoding as features. All other OMNI variants use NGBoost with the architecture encodings as features.

In Figure 10, we plot the percentage of Kendall Tau compared to the best predictor in {Jacob. Cov. SoTL-E, and NGBoost} for each initialization time and query time budget, for each OMNI variant. We see that the variants (NGBoost + Jacob. Cov.) and (SoTL-E + NGBoost) both give 20% Kendall Tau improvements for some budget constraints, but do not perform well for other budget constraints. On the other hand, (Jacob. Cov. + SoTL-E) has fairly consistent performance across all budget constraints. Finally, (NGBoost + Jacob. Cov. + SoTL-E) has consistent performance everywhere and peaks at 30% improvement. This ablation study shows that predictors from all three families are needed to achieve maximum performance.

Next, we give additional experiments for OMNI in other settings. Note that Figure 3 used the version of OMNI with NGBoost. See Figure 11 for the version of OMNI with SemiNAS, which performs worse. Finally, see Figure 12 for the performance of OMNI in the mutation-based setting described in Section 4. Note that it performs comparatively worse than in the standard uniformly random setting.

Algorithm 1 OMNI predictor

Input: Search space A, dataset D, initialization time budget B_{init}, query time budget B_{query}.
Initialization():

- $\mathcal{D}_{\text{train}} \leftarrow \emptyset$
- While $t < B_{\text{init}}$
  - Draw an architecture a randomly from A
  - Train a to completion to compute val. accuracy y_a
  - $\mathcal{D}_{\text{train}} \leftarrow \mathcal{D}_{\text{train}} \cup \{(a, y_a)\}$
- Train an NGBoost model m to predict the final val. accuracy of architectures from $\mathcal{D}_{\text{train}}$, using the architecture encoding, SoTL-E, and Jacob. cov. as input features.

Query(architecture a_{test}):

- While $t < B_{\text{query}}$, train a_{test}
- Compute SoTL-E using the partial learning curve, and compute Jacob. cov., and the arch. encoding of a_{test}
- Predict val. acc. of a_{test} using m and the above features.

Table 3: Comparison of hybrid learning curve + model-based methods on NAS-Bench-201 CIFAR-10, reporting the mean Kendall Tau rank correlation along with the standard deviation.

Init. time (s)	Query time (s)	LC-prev-builds	LCNet	LcSVR
$1.4e4$	64	$\mathbf{0.512 \pm 0.043}$	0.322 ± 0.128	0.427 ± 0.181
$1.4e4$	238	$\mathbf{0.676 \pm 0.029}$	0.204 ± 0.209	0.484 ± 0.172
$1.4e4$	932	$\mathbf{0.780 \pm 0.019}$	0.317 ± 0.265	0.497 ± 0.174
$7.6e4$	64	0.516 ± 0.042	0.388 ± 0.104	$\mathbf{0.596 \pm 0.065}$
$7.6e4$	238	0.688 ± 0.026	0.471 ± 0.084	$\mathbf{0.700 \pm 0.052}$
$7.6e4$	932	$\mathbf{0.797 \pm 0.016}$	0.583 ± 0.090	0.740 ± 0.048
$2.2e5$	64	0.523 ± 0.041	0.418 ± 0.074	$\mathbf{0.632 \pm 0.044}$
$2.2e5$	238	0.692 ± 0.023	0.472 ± 0.097	$\mathbf{0.736 \pm 0.034}$
$2.2e5$	932	$\mathbf{0.797 \pm 0.014}$	0.613 ± 0.077	0.795 ± 0.023
$1.1e6$	64	0.528 ± 0.039	0.452 ± 0.081	$\mathbf{0.667 \pm 0.045}$
$1.1e6$	238	0.698 ± 0.025	0.472 ± 0.101	$\mathbf{0.759 \pm 0.032}$
$1.1e6$	932	0.798 ± 0.014	0.637 ± 0.076	$\mathbf{0.823 \pm 0.023}$

C Additional Experiments on NAS-Bench-201

In this section, we give additional experiments carried out on NAS-Bench-201 CIFAR-10. These experiments include implementing additional hybrid predictors, running zero-cost proxy baselines, computing the variance in random seeds when the training and testing datasets are fixed, and running the predictors with longer HPO budgets.

C.1 Additional hybrid predictors

In Section 4, we implemented one predictor that was a hybrid learning curve + model-based method: LcSVR [2]. In this section, we implement two other hybrid predictors: LCNet [20] and "LCE using previous builds" [8]. For both techniques, we used the original implementation.

We compute the Kendall Tau rank correlation of LcSVR, LCNet, and LC-prev-builds on a representative set of four different initialization times and three different query times, for twelve total settings on NAS-Bench-201 CIFAR-10. See Table 3. We find that LC-prev-builds outperforms LcSVR when the initialization time is small, and when the query time is high. LCNet never outperformed LcSVR.

C.2 Zero-cost proxy baselines

In this section, we implement flops and params as baselines for zero-cost predictors. We compare the Kendall Tau rank correlation and Spearman rank correlation on NAS-Bench-201 CIFAR-10 for flops, params, and all of the zero-cost proxies implemented in Section 4. Note that zero-cost predictors

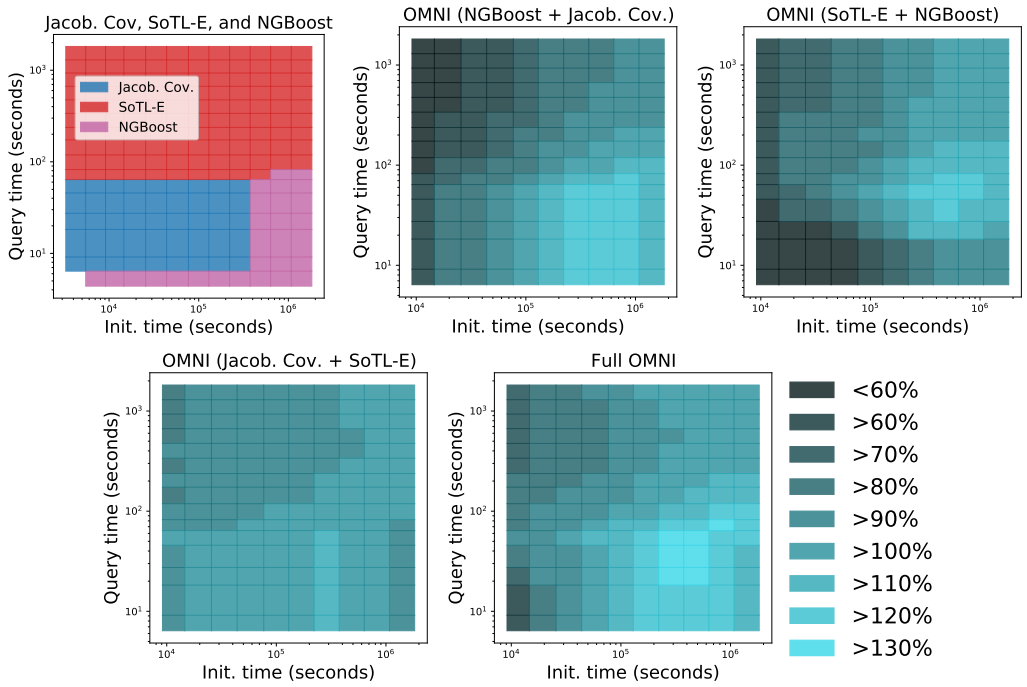

Figure 10: Ablation for OMNI. The first plot shows the best predictor in {Jacob. Cov. SoTL-E, and NGBoost} for each initialization time and query time budget. In the next four plots, we compute the percentage of the Kendall Tau value of the given OMNI variant, compared to the first plot. For example, in the bottom-right corner of the first plot, NGBoost achieves the highest Kendall Tau value from the set {Jacob. Cov. SoTL-E, and NGBoost}; In the bottom-right corner of the second plot, (NGBoost + Jacob. Cov.) achieves a Kendall Tau value that is 10% higher than the Kendall Tau value achieved by NGBoost. Therefore, combining Jacob. Cov. with NGBoost achieved a higher Kendall Tau value than the best individual predictor.

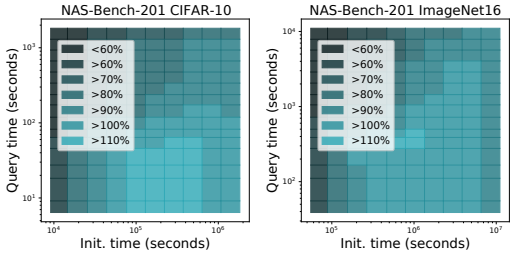

Figure 11: Percentage of OMNI (SemiNAS)'s Kendall Tau value compared to the next-best predictors for each budget constraint.

have just one setting: no initialization time and a query time of 5 seconds. See Table 4. Surprisingly, flops and params tie for the second highest Kendall Tau value out of all six zero-cost predictors on NAS-Bench-201 CIFAR-10, behind Jacobian covariance. For Spearman values, flops and params tie for third behind Jacobian covariance and SynFlow. Note that for the case of NAS-Bench-201, since the graph structure of the cell is fixed, flops has a one-to-one relationship with params (meaning they get the same rank correlation results).

C.3 Random seed experiments

Our plots in Figure 6 give the standard deviation across 100 trials for each predictor across different random seeds, which varies the tran and test sets as well as any stochasticity of the predictor. Now, we perform another experiment where we keep the train and test sets fixed and measure the standard

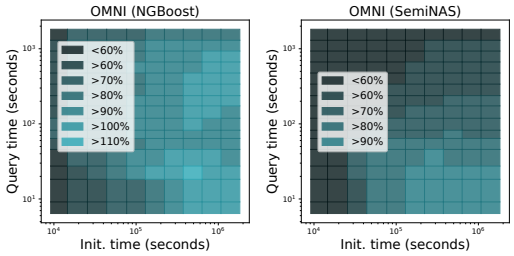

Figure 12: Percentage of OMNI's Kendall Tau value compared to the next-best predictors for each budget constraint, in the mutation-based setting on NAS-Bench-201 CIFAR-10.

Table 4: Comparison of baseline methods to zero-cost proxies on NAS-Bench-201 CIFAR-10.

Method	Kendall Tau	Spearman
Flops	0.539	0.713
Params	0.539	0.713
Fisher	0.218	0.299
Grad Norm	0.420	0.587
Grasp	0.330	0.481
Jacob. Cov.	**0.575**	**0.743**
SNIP	0.422	0.590
SynFlow	0.530	0.724

deviation across only the stochasticity of the predictor. For a representative set of eight predictors on NAS-Bench-201 CIFAR-10, we computed the standard deviation of 10 trials on a fixed train and test set, averaged over 50 trials of choosing new train and test sets (500 trials total per predictor). We used the median settings of initialization time and query time and reported the Kendall Tau rank correlation. See Table 5. We find that Bayesian Linear Regression has the highest stochasticity.

C.4 Longer HPO budgets

In Section 4, all model-based predictors were run using 15 minutes of hyperparameter tuning. In this section, we test whether there is further improvement when increasing the hyperparameter tuning budget from 15 minutes to 1 hour. We run this experiment on a representative set of ten model-based predictors on NAS-Bench-201 CIFAR-10, across all eleven initialization time settings from Section 4 (the query time for all model-based predictors is fixed at one second). For each predictor, after computing the improvement to Kendall Tau rank correlation when increasing the hyperparameter tuning budget, we report the average improvement over all initialization time settings, as well as the average of the top two and average of the worst two initialization time settings. See Table 6.

There were no predictors which achieved non-negligible improvement across all initialization time settings. Similarly, none of the predictors saw non-negligible decreases in performance, although BONAS had the lowest worst two average, at -0.020, likely due to additional overfitting that happens in the 1 hour hyperparameter tuning budget. For the top two average improvements, BANANAS saw the biggest improvements, and MLP, BONAS, and GCN also saw non-negligible improvements. The rest of the performance predictors saw little to no improvement. Interestingly, the largest improvements were all from deep learning based predictors.

D Reproducibility Table

When giving a large-scale comparison of performance predictors, it is important that all of the methods are accurately implemented and optimized. We took a number of reasonable steps to ensure this, including *(1)* using the original implementations whenever possible, and *(2)* applying light hyperparameter tuning to all methods, to help control for the fact that different techniques received different amounts of hyperparameter tuning in their original release. The best way to ensure that all

Table 5: Standard deviation of predictors when train and test sets are fixed.

Method	Mean	Std. Dev.	Std. Dev. w. fixed datasets
BANANAS	0.254	0.050	0.032
Bayes. Lin. Reg.	0.291	**0.065**	**0.115**
Jacob. Cov	0.539	0.000	0.006
NGBoost	0.355	0.059	0.032
SoTL-E	**0.623**	0.032	0.000
SynFlow	0.529	0.000	0.001
Var. Sparse GP	0.486	0.054	0.000
XGBoost	0.390	0.052	0.031

Table 6: Standard deviation of predictors when train and test sets are fixed.

Method	Avg.	Worst Two Avg.	Top Two Avg.
BANANAS	0.012	-0.010	**0.059**
Bayes. Lin. Reg.	0.002	-0.006	0.014
BOHAMIANN	0.001	-0.003	0.006
BONAS	0.008	**-0.020**	0.036
DNGO	0.000	-0.008	0.007
GCN	0.008	-0.016	0.035
GP	0.003	-0.004	0.009
MLP	0.009	-0.009	0.039
Sparse GP	-0.003	-0.008	0.006
Var. Sparse GP	**0.017**	-0.002	0.032

methods are properly implemented is to reproduce the results reported by the original authors of a given technique.

In this section, we present a table to clarify for which of the 31 predictors we have succeeded in reproducing published results, and for which predictors it is not possible. For each predictor, we mark whether the original paper (or first paper to run on a NAS benchmark) had *(1)* at least one search space out of the ones we used, *(2)* at least one initialization and query time that matches one of our settings, and *(3)* at least one metric from our set of metrics. If yes to *(1)-(3)*, then we check whether we *(4)* achieved nearly the same numbers as the original paper. See Table 7.

Of the 31 predictors, 14 were released before any of the NAS-Bench search spaces had come out. Two more did not give experiments in the setting of our paper. We are able to fairly compare the remaining 15 performance predictors (up to smaller changes in the experimental setting, such as different test set sizes) with either the original paper or the first paper to give results on a NAS benchmark. Our results are close to the original results, or in some cases stronger (due to our use of HPO). All 15 are within 0.04 of the reported rank correlation value or higher.

Table 7: Reproducibility table, clarifying which predictors we have reproduced with respect to existing published results.
[1] Compared with first follow-up paper to give correlation results on a NAS benchmark. [2] Larger test set. [3] Approximated from a plot.

Predictor	Paper	Search space	Setting	Metric	Value	Ours	Diff.
BANANAS	[69]	NAS-Bench-101	1000 train, 100 test	Pearson	0.699	0.904	+0.205
Bayes. Lin. Reg.	[6]	Released before NAS-Bench search spaces					
BOHAMIANN	[57]	Released before NAS-Bench search spaces					
BONAS	[54]	NAS-Bench-101	Only used size 360k train set.				
DNGO	[56]	Released before NAS-Bench search spaces					
Early Stop. ValAcc	[50][1]	NAS-Bench-201	100 epochs, 1000 test[2]	Spearman	0.85[3]	0.850	+0.0
Early Stop. ValLoss	[50][1]	NAS-Bench-201	100 epochs, 1000 test[2]	Spearman	0.83[3]	0.839	+0.009
Fisher	[1][1]	NAS-Bench-201	15k test[2]	Spearman	0.36	0.328	-0.032
GCN	[67]	NAS-Bench101[1]	1000 train, 100 test	Pearson	0.607	0.793	+0.186
GP	[48]	Released before NAS-Bench search spaces					
Grad Norm	[1]	NAS-Bench-201	15k test[2]	Spearman	0.58	0.587	+0.007
Grasp	[1][1]	NAS-Bench-201	15k test[2]	Spearman	0.48	0.481	+0.001
Jacob. Cov.	[1]	NAS-Bench-201	1000 test[2]	Pearson	0.574	0.575	+0.001
LCE	[11]	Released before NAS-Bench search spaces					
LCE-m	[20]	Released before NAS-Bench search spaces					
LcSVR	[50]	NAS-Bench-201[1]	100 epochs, 100 train, 1000 test[2]	Spearman	0.93[3]	0.931	+0.001
LGBoost	[18]	NAS-Bench101	1000 train, 100 test	Kendall Tau	0.640	0.705	+0.065
MLP	[69][1]	NAS-Bench101	1000 train, 100 test	Pearson	0.400	0.447	+0.047
NAO	[35]	Does not have NAS-Bench search spaces					
NGBoost	[14]	Released before NAS-Bench search spaces					
OneShot	[73]	Released before NAS-Bench search spaces					
Rand. Forest	[29]	Released before NAS-Bench search spaces					
RSWS	[26]	Released before NAS-Bench search spaces					
SemiNAS	[34]	NAS-Bench-101	No predictor experiments				
SNIP	[1][1]	NAS-Bench-201	15k test[2]	Spearman	0.58	0.590	+0.01
SoTL	[50]	NAS-Bench-201	100 epochs, 1000 test[2]	Spearman	0.96[3]	0.963	+0.003
SoTL-E	[50]	NAS-Bench-201	100 epochs, 1000 test[2]	Spearman	0.93[3]	0.932	+0.002
Sparse GP	[46]	Released before NAS-Bench search spaces					
SynFlow	[1]	NAS-Bench-201	15k test[2]	Spearman	0.74	0.738	-0.002
Var. Sparse GP	[63]	Released before NAS-Bench search spaces					
XGBoost	[9]	Released before NAS-Bench search spaces					