NAS-Bench-101: Towards Reproducible Neural Architecture Search

Chris Ying [*1] **Aaron Klein** [*2] **Esteban Real** [1] **Eric Christiansen** [1] **Kevin Murphy** [1] **Frank Hutter** [2]

Abstract

Recent advances in neural architecture search (NAS) demand tremendous computational resources, which makes it difficult to reproduce experiments and imposes a barrier-to-entry to researchers without access to large-scale computation. We aim to ameliorate these problems by introducing NAS-Bench-101, the first public architecture dataset for NAS research. To build NAS-Bench-101, we carefully constructed a compact, yet expressive, search space, exploiting graph isomorphisms to identify 423k unique convolutional architectures. We trained and evaluated all of these architectures multiple times on CIFAR-10 and compiled the results into a large dataset of over 5 million trained models. This allows researchers to evaluate the quality of a diverse range of models in milliseconds by querying the precomputed dataset. We demonstrate its utility by analyzing the dataset as a whole and by benchmarking a range of architecture optimization algorithms.

1. Introduction

Many successes in deep learning (Krizhevsky et al., 2012; Goodfellow et al., 2014; Sutskever et al., 2014) have resulted from novel neural network architecture designs. For example, in the field of image classification, research has produced numerous ways of combining neural network layers into unique architectures, such as Inception modules (Szegedy et al., 2015), residual connections (He et al., 2016), or dense connections (Huang et al., 2017). This proliferation of choices has fueled research into *neural architecture search (NAS)*, which casts the discovery of new architectures as an optimization problem (Baker et al., 2017; Zoph & Le, 2016; Real et al., 2017; Elsken et al., 2019). This has resulted in state of the art performance in the domain of image classification (Zoph et al., 2018; Real et al., 2018; Huang et al., 2018), and has shown promising results in other domains, such as sequence modeling (Zoph & Le, 2016; So et al., 2019).

Unfortunately, NAS research is notoriously hard to reproduce (Li & Talwalkar, 2019; Sciuto et al., 2019). First, some methods require months of compute time (*e.g.*, Zoph et al., 2018), making these methods inaccessible to most researchers. Second, while recent improvements (Liu et al., 2018a; Pham et al., 2018; Liu et al., 2018b) have yielded more efficient methods, different methods are not comparable to each other due to different training procedures and different search spaces, which make it difficult to attribute the success of each method to the search algorithm itself.

To address the issues above, this paper introduces NAS-Bench-101, the first architecture-dataset for NAS. To build it, we trained and evaluated a large number of different convolutional neural network (CNN) architectures on CIFAR-10 (Krizhevsky & Hinton, 2009), utilizing over 100 TPU years of computation time. We compiled the results into a large table which maps 423k unique architectures to metrics including run time and accuracy. This enables NAS experiments to be run via querying a table instead of performing the usual costly train and evaluate procedure. Moreover, the data, search space, and training code is fully public [1], to foster reproducibility in the NAS community.

Because NAS-Bench-101 exhaustively evaluates a search space, it permits, for the first time, a comprehensive analysis of a NAS search space as a whole. We illustrate such potential by measuring search space properties relevant to architecture search. Finally, we demonstrate its application to the analysis of algorithms by benchmarking a wide range of open source architecture/hyperparameter search methods, including evolutionary approaches, random search, and Bayesian optimization.

In summary, our contributions are the following:

- We introduce NAS-Bench-101, the first large-scale, open-

[*]Equal contribution [1]Google Brain, Mountain View, California, USA [2]Department of Computer Science, University of Freiburg, Germany. Correspondence to: Chris Ying <contact@chrisying.net>, Aaron Klein <kleinaa@cs.uni-freiburg.de>, Esteban Real <ereal@google.com>.

Proceedings of the 36^{th} International Conference on Machine Learning, Long Beach, California, PMLR 97, 2019.

[1] Data and code for NAS-Bench-101 available at https://github.com/google-research/nasbench.

source architecture dataset for NAS (Section 2);
- We illustrate how to use the dataset to analyze the nature of the search space, revealing insights which may guide the design of NAS algorithms (Section 3);
- We illustrate how to use the dataset to perform fast benchmarking of various open-source NAS optimization algorithms (Section 4).

2. The NASBench Dataset

The NAS-Bench-101 dataset is a table which maps neural network architectures to their training and evaluation metrics. Most NAS approaches to date have trained models on the CIFAR-10 classification set because its small images allow relatively fast neural network training. Furthermore, models which perform well on CIFAR-10 tend to perform well on harder benchmarks, such as ImageNet (Krizhevsky et al., 2012) when scaled up (Zoph et al., 2018)). For these reasons, we also use CNN training on CIFAR-10 as the basis of NAS-Bench-101.

2.1. Architectures

Similar to other NAS approaches, we restrict our search for neural net topologies to the space of small feedforward structures, usually called *cells*, which we describe below. We stack each cell 3 times, followed by a downsampling layer, in which the image height and width are halved via max-pooling and the channel count is doubled. We repeat this pattern 3 times, followed by global average pooling and a final dense softmax layer. The initial layer of the model is a *stem* consisting of one 3×3 convolution with 128 output channels. See Figure 1, top-left, for an illustration of the overall network structure. Note that having a stem followed by stacks of cells is a common pattern both in hand-designed image classifiers (He et al., 2016; Huang et al., 2017; Hu et al., 2018) and in NAS search spaces for image classification. Thus, the variation in the architectures arises from variation in the cells.

The space of cell architectures consists of all possible directed acyclic graphs on V nodes, where each possible node has one of L labels, representing the corresponding operation. Two of the vertices are specially labeled as operation `IN` and `OUT`, representing the input and output tensors to the cell, respectively. Unfortunately, this space of labeled DAGs grows exponentially in both V and L. In order to limit the size of the space to allow exhaustive enumeration, we impose the following constraints:
- We set $L = 3$, using only the following operations:
  - 3×3 convolution
  - 1×1 convolution
  - 3×3 max-pool
- We limit $V \leq 7$.
- We limit the maximum number of edges to 9.

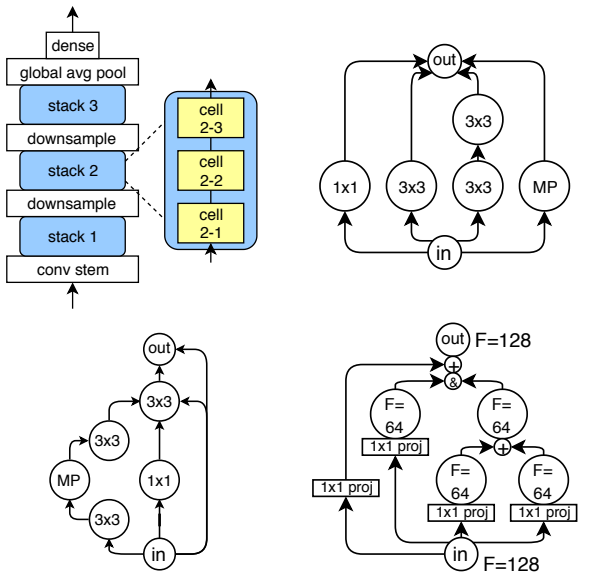

Figure 1: (top-left) The outer skeleton of each model. (top-right) An Inception-like cell with the original 5x5 convolution approximated by two 3x3 convolutions (concatenation and projection operations omitted). (bottom-left) The cell that attained the lowest mean test error (projection layers omitted). (bottom-right) An example cell that demonstrates how channel counts are automatically determined ("+" denotes addition and "&" denotes concatenation; 1×1 projections are used to scale channel counts).

All convolutions utilize batch normalization followed by ReLU. These constraints were chosen to ensure that the search space still contains ResNet-like and Inception-like cells (He et al., 2016; Szegedy et al., 2016). An example of an Inception-like cell is illustrated in Figure 1, top-right. We intentionally use convolutions instead of separable convolutions to match the original designs of ResNet and Inception, although this comes as the cost of being more parameter-heavy than some of the more recent state-of-the-art architectures like AmoebaNet (Real et al., 2018).

2.2. Cell encoding

There are multiple ways to encode a cell and different encodings may favor certain algorithms by biasing the search space. For most of our experiments, we chose to use a very general encoding: a 7-vertex directed acyclic graph, represented by a 7×7 upper-triangular binary matrix, and a list of 5 labels, one for each of the 5 intermediate vertices (recall that the input and output vertices are fixed) Since there are 21 possible edges in the matrix and 3 possible operations for each label, there are $2^{21} * 3^5 \approx 510M$ total unique models in this encoding. In Supplement S3, we also

discuss an alternative encoding.

However, a large number of models in this space are invalid (i.e., there is no path from the input vertex, or the number of total edges exceeds 9). Furthermore, different graphs in this encoding may not be computationally unique. The method which we used to identify and enumerate unique graphs is described in Supplement S1. After de-duplication, there are approximately 423k unique graphs in the search space.

2.3. Combine semantics

Translating from the graph to the corresponding neural network is straightforward, with one exception. When multiple edges point to the same vertex, the incoming tensors must be combined. Adding them or concatenating them are both standard techniques. To support both ResNet and Inception-like cells and to keep the space tractable, we adopted the following fixed rule: tensors going to the output vertex are concatenated and those going into other vertices are summed. The output tensors from the input vertex are projected in order to match the expected input channel counts of the subsequent operations. This is illustrated in Figure 1, bottom-right.

2.4. Training

The training procedure forms an important part of an architecture search benchmark, since different training procedures can lead to very substantial performance differences. To counter this issue and allow comparisons of NAS algorithms on equal grounds, we designed and open-sourced a single general training pipeline for all models in the dataset.

Choice of hyperparameters. We utilize a single, fixed set of hyperparameters for all NAS-Bench-101 models. This set of hyperparameters was chosen to be robust across different architectures by performing a coarse grid search optimizing the average accuracy of a set of 50 randomly-sampled architectures from the space. This is similar to standard practice in the literature (Zoph et al., 2018; Liu et al., 2018a; Real et al., 2018) and is further justified by our experimental analysis in Section 5.1.

Implementation details. All models are trained and evaluated on CIFAR-10 (40k training examples, 10k validation examples, 10k testing examples), using standard data augmentation techniques (He et al., 2016). The learning rate is annealed via cosine decay (Loshchilov & Hutter, 2017) to 0 in order to reduce the variance between multiple independent training runs. Training is performed via RMSProp (Tieleman & Hinton, 2012) on the cross-entropy loss with L2 weight decay. All models were trained on the TPU v2 accelerator. The code, implemented in TensorFlow, along with all chosen hyperparameters, is publicly available at `https://github.com/google-research/nasbench`.

3 repeats and 4 epoch budgets. We repeat the training and evaluation of all architectures 3 times to obtain a measure of variance. Also, in order to allow the evaluation of multi-fidelity optimization methods, *e.g.*, Hyperband (Li et al., 2018)), we trained all our architectures with four increasing epoch budgets: $E_{\text{stop}} \in \{E_{\max}/3^3, E_{\max}/3^2, E_{\max}/3, E_{\max}\} = \{4, 12, 36, 108\}$ epochs. In each case, the learning rate is annealed to 0 by epoch E_{stop}.[2] We thus trained $3 \times 423k \sim 1.27M$ models for each value of E_{stop}, and thus $4 \times 1.27M \sim 5M$ models overall.

2.5. Metrics

We evaluated each architecture A after training three times with different random initializations, and did this for each of the 4 budgets E_{stop} above. As a result, the dataset is a mapping from the $(A, E_{\text{stop}}, \texttt{trial\#})$ to the following quantities:

- training accuracy;
- validation accuracy;
- testing accuracy;
- training time in seconds; and
- number of trainable model parameters.

Only metrics on the training and validation set should be used to search models within a single NAS algorithm, and testing accuracy should only be used for an offline evaluation. The training time metric allows benchmarking algorithms that optimize for accuracy while operating under a time limit (Section 4) and also allows the evaluation of multi-objective optimization methods. Other metrics that do not require retraining can be computed using the released code.

2.6. Benchmarking methods

One of the central purposes of the dataset is to facilitate benchmarking of NAS algorithms. This section establishes recommended best practices for using NAS-Bench-101 which we followed in our subsequent analysis; we also refer to Supplement S6 for a full set of best practices in benchmarking with NAS-Bench-101.

The goal of NAS algorithms is to find architectures that have high testing accuracy at epoch $E_{\max}$. To do this, we repeatedly query the dataset at (A, E_{stop}) pairs, where A is an architecture in the search space and E_{stop} is an allowed number of epochs ($E_{\text{stop}} \in \{4, 12, 36, 108\}$). Each query does a look-up using a random trial index, drawn uniformly

[2] Instead of 4 epoch budgets, we could have trained single long runs and used the performance at intermediate checkpoints as benchmarking data for early stopping algorithms. However, because of the learning rate schedule, such checkpoints would have occurred when the learning rates are still high, leading to noisy accuracies that do not correlate well with the final performance.

at random from $\{1, 2, 3\}$, to simulate the stochasticity of SGD training.

While searching, we keep track of the best architecture $\hat{A}_i$ the algorithm has found after each function evaluation i, as ranked by its *validation accuracy*. To best simulate real world computational constratints, we stop the search run when the total "training time" exceeds a fixed limit. After each complete search rollout, we query the corresponding mean *test accuracy* $f(\hat{A}_i)$ for that model (test accuracy should never be used to guide the search itself). Then we compute the immediate test regret: $r(\hat{A}_i) = f(\hat{A}_i) - f(A^*)$, where A^* denotes the model with the highest mean test accuracy in the entire dataset. This regret becomes the score for the search run. To measure the robustness of different search algorithms, a large number of independent search rollouts should be conducted.

3. NASBench as a Dataset

In this section, we analyze the NAS-Bench-101 dataset as a whole to gain some insight into the role of neural network operations and cell topology in the performance of convolutional neural networks. In doing so, we hope to shed light on the loss landscape that is traversed by NAS algorithms.

3.1. Dataset statistics

First we study the empirical cumulative distribution (ECDF) of various metrics across all architectures in Figure 2. Most of the architectures converge and reach 100% training accuracy. The validation accuracy and test accuracy are both above 90% for a majority of models. The best architecture in our dataset (Figure 1) achieved a mean test accuracy of 94.32%. For comparison, the ResNet-like and Inception-like cells attained 93.12% and 92.95%, respectively, which is roughly in-line with the performance of the original ResNet-56 (93.03%) on CIFAR-10 (He et al., 2016). We observed that the correlation between validation and test accuracy is extremely high ($r = 0.999$) at 108 epochs which suggests that strong optimizers are unlikely to overfit on the validation error. Due to the stochastic nature of the training process, training and evaluating the same architecture will generally lead to a small amount of noise in the accuracy. We also observe, as expected, that the noise between runs is lower at longer training epochs.

Figure 3 investigates the relationship between the number of parameters, training time, and validation accuracy of models in the dataset. The left plot suggests that there is positive correlation between all of these quantities. However parameter count and training time are not the only factors since the best cell in the dataset is not the most computationally intensive one. Hand-designed cells, such as ResNet and Inception, perform near the Pareto frontier of accuracy over cost, which suggests that topology and operation selection are critical for finding both high-accuracy and low-cost models.

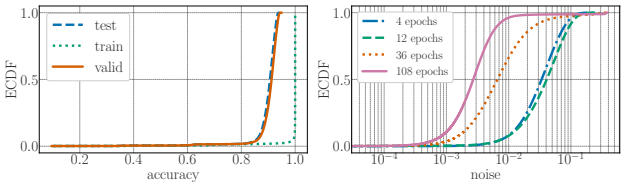

Figure 2: The empirical cumulative distribution (ECDF) of all valid configurations for: (left) the train/valid/test accuracy after training for 108 epochs and (right) the noise, defined as the standard deviation of the test accuracy between the three trials, after training for 12, 36 and 108 epochs.

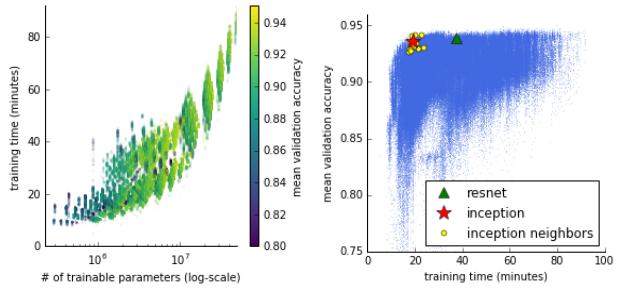

Figure 3: (left) Training time vs. trainable parameters, color-coded by validation accuracy. (right) Validation accuracy vs. training time with select cell architectures highlighted. Inception neighbors are the graphs which are 1-edit distance away from the Inception cell.

3.2. Architectural design

NAS-Bench-101 presents us with the unique opportunity to investigate the impact of various architectural choices on the performance of the network. In Figure 4, we study the effect of replacing each of the operations in a cell with a different operation. Not surprisingly, replacing a 3×3 convolution with a 1×1 convolution or 3×3 max-pooling operation generally leads to a drop in absolute final validation accuracy by 1.16% and 1.99%, respectively. This is also reflected in the relative change in training time, which decreases by 14.11% and 9.84%. Even though 3×3 max-pooling is parameter-free, it appears to be on average 5.04% more expensive in training time than 1×1 convolution and also has an average absolute validation accuracy 0.81% lower. However, some of the top cells in the space (ranked by mean test accuracy, i.e., Figure 1) contain max-pool operations, so other factors must also be at play and replacing all 3×3 max-

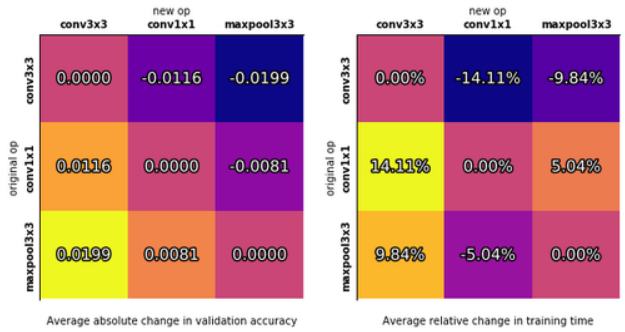

Figure 4: Measuring the aggregated impact of replacing one operation with another on (left) absolute validation accuracy and (right) relative training time.

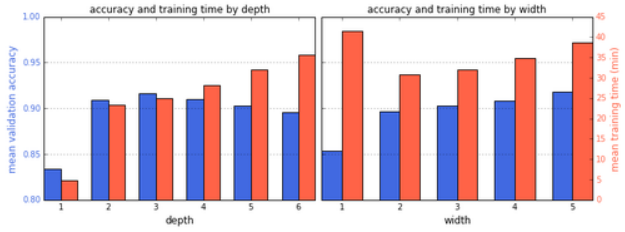

Figure 5: Comparing mean validation accuracy and training time for cells by (left) depth, measured by length of longest path from inpu to output, and (right) width, measured by maximum directed cut on the graph.

pooling operations with 1×1 convolutions is not necessarily a globally optimal choice.

In Figure 5, we also investigate the role of depth vs. width. In terms of average validation accuracy, it appears that a depth of 3 is optimal whereas increasing width seems to increase the validation accuracy up to 5, the maximum width of networks in the dataset. The training time of networks increases as networks get deeper and wider with one exception: width 1 networks are the most expensive. This is a consequence of the combine semantics (see Section 2.3), which skews the training time distributions because all width 1 networks are simple feed-forward networks with no branching, and thus the activation maps are never split via their channel dimension.

3.3. Locality

NASBench exhibits *locality*, a property by which architectures that are "close by" tend to have similar performance metrics. This property is exploited by many search algorithms. We define "closeness" in terms of *edit-distance*: the smallest number of changes required to turn one architecture into another; one *change* entails flipping the operation at a vertex or the presence/absence of an edge. A popular measure of locality is the random-walk autocorrelation (RWA), defined as the autocorrelation of the accuracies of points visited as we perform a long walk of random changes through the space (Weinberger, 1990; Stadler, 1996). The RWA (Figure 6, left) shows high correlations for lower distances, indicating locality. The correlations become indistinguishable from noise beyond a distance of about 6.

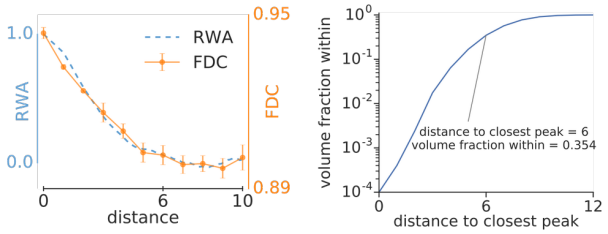

Figure 6: (left) RWA for the full space and the FDC relative to the global maximum. To plot both curves on a common horizontal axis, the autocorrelation curve is drawn as a function of the square root of the autocorrelation shift, to account for the fact that a random walk reaches a mean distance $\sqrt{N}$ after N steps. (right) Fraction of the search space volume that lies within a given distance to the closest high peak.

While the RWA aggregates across the whole space, we can also consider regions of particular interest. For example, Figure 3 (right) displays the neighbors of the Inception-like cell, indicating a degree of locality too, especially in terms of accuracy. Another interesting region is that around a global accuracy maximum. To measure locality within this neighborhood, we used the fitness-distance correlation metric (FDC, Jones et al. (1995)). Figure 6 (left) shows that there is locality around the global maximum as well and the peak also has a coarse-grained width of about 6.

More broadly, we can consider how rare it is to be near a global maximum. In the cell encoding described in Section 2.2, the best architecture (*i.e.*, the one with the highest mean testing accuracy) has 4 graph isomorphisms, producing 4 distinct peaks in our encoded search space. Moreover there are 11 other architectures whose mean test accuracy is within 2 times standard error of the mean of the best graph. Including the isomorphisms of these, too, there are 11 570 points in the 510M-point search space corresponding to these *top graphs*, meaning that the chance of hitting one of them with a random sample is about 1 to 50000. Figure 6 (right) shows how much volume of the search space lies near these graphs; in particular, 35.4% of the search space is within a distance of 6 from the closest top graph. Since the basin of attraction for local search appears to have a width of about 6, this suggests that locality-based search may be a good choice for this space.

4. NASBench as a Benchmark

4.1. Comparing NAS algorithms

In this section we establish baselines for future work by using our dataset to compare some popular algorithms for which open source code is available. Note that the intention is not to answer the question "*Which methods work best on this benchmark?*", but rather to demonstrate the utility of a reproducible baseline.

We benchmarked a small set of NAS and hyperparameter optimization (HPO) algorithms with publicly available implementations: random search (RS) (Bergstra & Bengio, 2012), regularized evolution (RE) (Real et al., 2018), SMAC (Hutter et al., 2011), TPE (Bergstra et al., 2011), Hyperband (HB) (Li et al., 2018), and BOHB (Falkner et al., 2018). We follow the guidelines established in Section 2.6. Due to its recent success for NAS (Zoph & Le, 2016), we also include our own implementation of reinforcement learning (RL) as an additional baseline, since an official implementation is not available. However, instead of using an LSTM controller, which we found to perform worse, we used a categorical distribution for each parameter and optimized the probability values directly with REINFORCE. Supplement S2 has additional implementation details for all methods.

NAS algorithms based on weight sharing (Pham et al., 2018; Liu et al., 2018b) or network morphisms (Cai et al., 2018; Elsken et al., 2018) cannot be directly evaluated on the dataset, so we did not include them. We also do not include Gaussian process–based HPO methods (Shahriari et al., 2016), such as Spearmint (Snoek et al., 2012), since they tend to have problems in high-dimensional discrete optimization tasks (Eggensperger et al., 2013). While Bayesian optimization methods based on Bayesian neural networks (Snoek et al., 2015; Springenberg et al., 2016) are generally applicable to this benchmark, we found their computational overhead compared to the other methods to be prohibitively expensive for an exhaustive empirical evaluation. The benchmarking scripts we used are publicly available[3]. For all optimizers we investigate their own main meta-parameters in Supplement S2.2 (except for TPE where the open-source implementation does not allow to change the meta-parameters) and report here the performance based on the best found settings.

Figure 7 (left) shows the mean performance of each of these NAS/HPO algorithms across 500 independent trials. The x-axis shows *estimated wall-clock time*, counting the evaluation of each architecture with the time that the corresponding training run took. Note that the evaluation of 500 trials of each NAS algorithm (for up to 10M simulated TPU seconds, i.e., 115 TPU days each) was only made possible by virtue of our tabular benchmark; without it, they would have amounted to over 900 TPU years of computation.

We make the following observations:

- RE, BOHB, and SMAC perform best and start to outperform RS after roughly 50 000 TPU seconds (the equivalent of roughly 25 evaluated architectures); they achieved the final performance of RS about 5 times faster and continued to improve beyond this point.
- SMAC, as a Bayesian optimization method, performs this well despite the issue of invalid architectures; we believe that this is due to its robust random forest model. SMAC is slightly slower in the beginning of the search; we assume that this is due to its internal incumbent estimation procedure (which evaluates the same architecture multiple times).
- The other Bayesian optimization method, TPE, struggles with this benchmark, with its performance falling back to random search.
- The multi-fidelity optimization algorithms HB and BOHB do not yield the speedups frequently observed compared to RS or Bayesian optimization. We attribute this to the relatively low rank-correlation between the performance obtained with different budgets (see Figure 7 in Supplement S2).
- BOHB achieves the same test regret as SMAC and RE after recovering from misleading early evaluations; we attribute this to the fact, that, compared to TPE, it uses a multivariate instead of a univariate kernel density estimator.
- Even though RL starts outperforming RS at roughly the same time as the other methods, it converges much slower towards the global optimum.

Besides achieving good performance, we argue that robustness, *i.e.*, how sensitive an optimizer is to the randomness in both the search algorithm and the training process, plays an important role in practice for HPO and NAS methods. This aspect has been neglected in the NAS literature due to the extreme cost of performing many repeated runs of NAS experiments, but with NAS-Bench-101 performing many repeats becomes trivial. Figure 7 (right) shows the empirical cumulative distribution of the regret after 10M seconds across all 500 runs of each method. For all methods, the final test regrets ranged over roughly an order of magnitude, with RE, BOHB, and SMAC showing the most robust performance.

4.2. Generalization bootstrap

To test the generalization of our findings on the dataset, we ideally would need to run the benchmarked algorithms on a larger space of architectures. However, due to computational limitations, it is infeasible for us to run a large number of NAS trials on a meaningfully larger space. Instead, to provide some preliminary evidence of generalization, we

[3] https://github.com/automl/nas_benchmarks

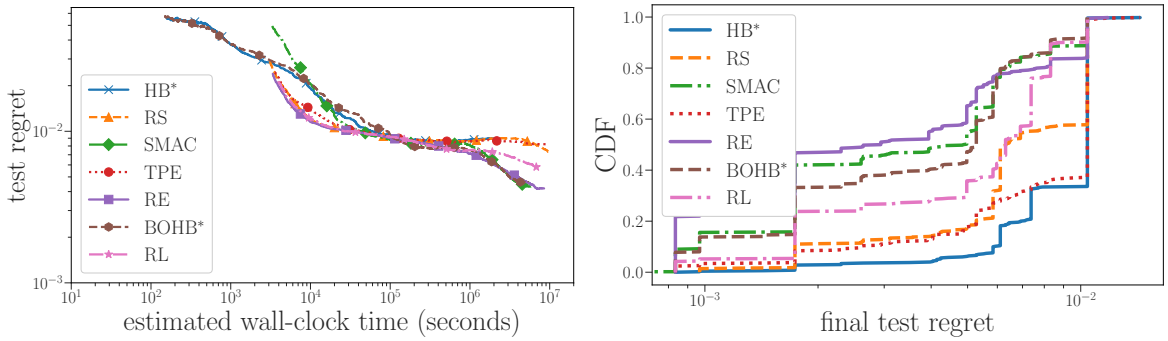

Figure 7: (left) Comparison of the performance of various search algorithms. The plot shows the mean performance of 500 independent runs as a function of the estimated training time. (right) Robustness of different optimization methods with respect to the seed for the random number generator. *HB and BO-HB are budget-aware algorithms which query the dataset a shorter epoch lengths. The remaining methods only query the dataset at the longest length (108 epochs).

perform a bootstrapped experiment: we set aside a subset of NAS-Bench-101, dubbed NAS-Bench-Mini, and compare the outcomes of algorithms run on NAS-Bench-Mini compared to the full NAS-Bench-101. NAS-Bench-Mini contains all cells within the search space that utilize 6 or fewer vertices ($64.5k$ unique cells), compared to the full NAS-Bench-101 that uses up to 7 vertices ($423k$ unique cells).

We compare two very similar algorithms (regularized evolution, RE, and non-regularized evolution, NRE) to a baseline (random search, RS). RE and NRE are identical except that RE removes the *oldest* individual in a population to maintain the population size whereas NRE removes the *lowest fitness* individual. Figure 8 (top) shows the comparison on NAS-Bench-Mini and NAS-Bench-101 on 100 trials of each algorithm to a fixed time budget. The plots show that the rankings of the three algorithms (RS $<$ NRE $<$ RE) are consistent across the smaller dataset and the larger one. Furthermore, we demonstrate that NAS-Bench-Mini can generalize to NAS-Bench-101 for different hyperparameter settings of a single algorithm (regularized evolution) in Figure 8 (middle, bottom). This suggests that conclusions drawn from NAS-Bench-101 may generalize to larger search spaces.

5. Discussion

In this section, we discuss some of the choices we made when designing NAS-Bench-101.

5.1. Relationship to hyperparameter optimization

All models in NAS-Bench-101 were trained with a fixed set of hyperparameters. In this section, we justify that choice. The problem of hyperparameter optimization (HPO) is closely intertwined with NAS. NAS aims to discover good neural network architectures while HPO involves finding the best set of training hyperparameters for a given architecture. HPO operates by tuning various numerical neural network training parameters (*e.g.*, learning rate) as well as categorical choices (*e.g.*, optimizer type) to optimize the training process. Formally, given an architecture A, the task of HPO is to find its optimal hyperparameter configuration H^*:

$$H^*(A) = \arg\max_H f(A, H),$$

where f is a performance metric, such as validation accuracy and the $\arg\max$ is over all possible hyperparameter configurations. The "pure" NAS problem can be formulated as finding an architecture A^* when all architectures are evaluated under optimal hyperparameter choices:

$$A^* = \arg\max_A f(A, H^*(A)),$$

In practice, this would involve running an inner HPO search for each architecture, which is computationally intractable. We therefore approximate A^* with $A^\dagger$:

$$A^* \approx A^\dagger = \arg\max_A f(A, H^\dagger),$$

where $H^\dagger$ is a set of hyperparameters that has been estimated by maximizing the average accuracy on a small subset S of the architectures:

$$H^\dagger(S) = \arg\max_H \overline{\{f(A, H) : A \in S\}}.$$

For example, in Section 2.4, S was a random sample of 50 architectures.

To justify the approximation above, we performed a study on a different set of NAS-HPO-Bench (Klein & Hutter, 2019) datasets (described in detail in Supplement S5) These are smaller datasets of architecture–hyperparameter pairs

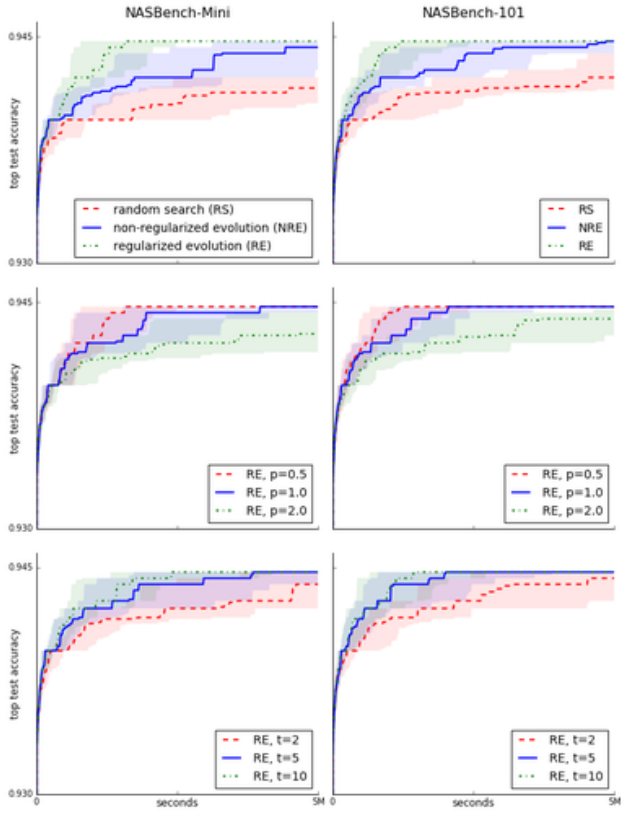

Figure 8: Generalization bootstrap experiments. Each line marks the median of 100 runs and the shaded region is the 25% to 75% interquartile range. (top) Comparing random search (RS), non-regularized evolution (NRE), and regularized evolution (RE) against NAS-Bench-Mini and NAS-Bench-101. (middle) Comparing RE runs with different mutation rates. (bottom) Comparing RE runs with different tournament sizes.

(A, H), where we computed $f(A, H)$ for all settings of A and H. This let us compute the *exact* hyperparameter-optimized accuracy, $f^*(A) = \max_H f(A, H)$. We can also measure how well this correlates with the approximation we use in NAS-Bench-101. To do this, we chose a set of hyperparameters $H^\dagger$ by optimizing the mean accuracy across all of the architectures for a given dataset. This allows us to map each architecture A to its *approximate* hyperparameter-optimized accuracy, $f^\dagger(A) = f(A, H^\dagger)$. (This approximate accuracy is analogous to the one computed in the NAS-Bench-101 metrics, except there the average was over 50 random architectures, not all of them.)

We find that $f^\dagger$ and f^* are quite strongly correlated across models, with a Spearman rank correlation of 0.9155; Figure 9 provides a scatter plot of f^* against $f^\dagger$ for the architectures. The ranking is especially consistent for the best architectures (points near the origin).

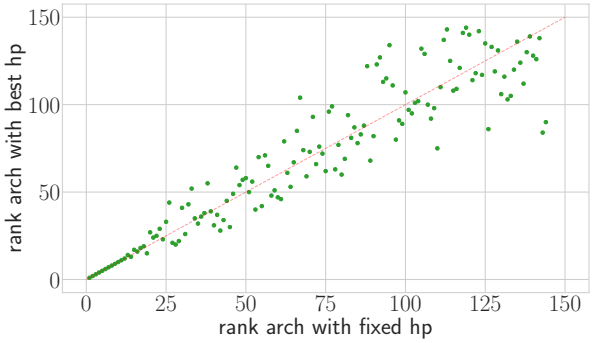

Figure 9: Scatter plot between ranks of f^* (vertical axis) and $f^\dagger$ (horizontal axis) on the NAS-HPO-Bench-Protein dataset. Ideally, the points should be close to the diagonal. The high correlation at low-rank means the best architectures are ranked identically when using H^* and $H^\dagger$.

5.2. Absolute accuracy of models

The choice of search space, hyperparameters, and training techniques were designed to ensure that NAS-Bench-101 would be feasible to compute with our resources. Unfortunately, this means that the models we evaluate do not reach current state-of-the-art performance on CIFAR-10. This is primarily because: (1) the search space is constrained in both size and selection of operations and does not contain more complex architectures, such as those used by NASNet (Zoph et al., 2018); (2) We do not apply the expensive "augmentation trick" (Zoph et al., 2018) by which models' depth and width are increased by a large amount and the training lengthened to hundreds of epochs; and (3) we do not utilize more advanced regularization like Cutout (DeVries & Taylor, 2017), ScheduledDropPath (Zoph et al., 2018) and decoupled weight decay (Loshchilov & Hutter, 2019) in order to keep our training pipeline similar to previous standardized models like ResNet.

6. Conclusion

We introduced NAS-Bench-101, a new tabular benchmark for neural architecture search that is inexpensive to evaluate but still preserves the original NAS optimization problem, enabling us to rigorously compare various algorithms quickly and without the enormous computational budgets often used by projects in the field. Based on the data we generated for this dataset, we were able to analyze the properties of an exhaustively evaluated set of convolutional neural architectures at unprecedented scale. In open-sourcing the NAS-Bench-101 data and generating code, we hope to make NAS research more accessible and reproducible. We also hope that NAS-Bench-101 will be the first of a continually improving sequence of rigorous benchmarks for the emerging NAS field.

Acknowledgements

Aaron and Frank gratefully acknowledge support by the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme under grant no. 716721, by BMBF grant DeToL, by the state of Baden-Württemberg through bwHPC and the German Research Foundation (DFG) through grant no INST 39/963-1 FUGG, and by a Google Faculty Research Award. Chris, Esteban, Eric, and Kevin would like to thank Quoc Le, Samy Bengio, Alok Aggarwal, Barret Zoph, Jon Shlens, Christian Szegedy, Jascha Sohl-Dickstein; and the larger Google Brain team.

References

Baker, B., Gupta, O., Naik, N., and Raskar, R. Designing neural network architectures using reinforcement learning. In *ICLR*, 2017.

Bergstra, J. and Bengio, Y. Random search for hyper-parameter optimization. *JMLR*, 2012.

Bergstra, J. S., Bardenet, R., Bengio, Y., and Kégl, B. Algorithms for hyper-parameter optimization. In *NIPS*, 2011.

Cai, H., Chen, T., Zhang, W., Yu, Y., and Wang, J. Efficient architecture search by network transformation. In *AAAI*, 2018.

DeVries, T. and Taylor, G. W. Improved regularization of convolutional neural networks with cutout. *arXiv*, 2017.

Eggensperger, K., Feurer, M., Hutter, F., Bergstra, J., Snoek, J., Hoos, H., and Leyton-Brown, K. Towards an empirical foundation for assessing bayesian optimization of hyperparameters. In *NIPS workshop on Bayesian Optimization in Theory and Practice*, December 2013.

Elsken, T., Metzen, J. H., and Hutter, F. Multi-objective architecture search for cnns. *arXiv preprint arXiv:1804.09081*, 2018.

Elsken, T., Metzen, J. H., and Hutter, F. Neural architecture search: A survey. *Journal of Machine Learning Research*, 20(55):1–21, April 2019.

Falkner, S., Klein, A., and Hutter, F. Bohb: Robust and efficient hyperparameter optimization at scale. *ICML*, 2018.

Goodfellow, I., Pouget-Abadie, J., Mirza, M., Xu, B., Warde-Farley, D., Ozair, S., Courville, A., and Bengio, Y. Generative adversarial nets. In *NIPS*, 2014.

He, K., Zhang, X., Ren, S., and Sun, J. Deep residual learning for image recognition. In *CVPR*, 2016.

Hu, J., Shen, L., and Sun, G. Squeeze-and-excitation networks. *CVPR*, 2018.

Huang, G., Liu, Z., Weinberger, K. Q., and van der Maaten, L. Densely connected convolutional networks. In *CVPR*, 2017.

Huang, Y., Cheng, Y., Chen, D., Lee, H., Ngiam, J., Le, Q. V., and Chen, Z. Gpipe: Efficient training of giant neural networks using pipeline parallelism. *arXiv preprint arXiv:1811.06965*, 2018.

Hutter, F., Hoos, H. H., and Leyton-Brown, K. Sequential model-based optimization for general algorithm configuration. In *International Conference on Learning and Intelligent Optimization*, 2011.

Jones, T. et al. *Evolutionary algorithms, fitness landscapes and search*. PhD thesis, Citeseer, 1995.

Klein, A. and Hutter, F. Tabular benchmarks for joint architecture and hyperparameter optimization. 2019.

Krizhevsky, A. and Hinton, G. Learning multiple layers of features from tiny images. *Master's thesis, Dept. of Computer Science, U. of Toronto*, 2009.

Krizhevsky, A., Sutskever, I., and Hinton, G. E. Imagenet classification with deep convolutional neural networks. In *NIPS*, 2012.

Li, L. and Talwalkar, A. Random Search and Reproducibility for Neural Architecture Search. *arXiv e-prints*, art. arXiv:1902.07638, Feb 2019.

Li, L., Jamieson, K., DeSalvo, G., Rostamizadeh, A., and Talwalkar, A. Hyperband: A novel bandit-based approach to hyperparameter optimization. *JMLR*, 2018.

Liu, C., Zoph, B., Shlens, J., Hua, W., Li, L.-J., Fei-Fei, L., Yuille, A., Huang, J., and Murphy, K. Progressive neural architecture search. *ECCV*, 2018a.

Liu, H., Simonyan, K., and Yang, Y. Darts: Differentiable architecture search. *ICLR*, 2018b.

Loshchilov, I. and Hutter, F. Sgdr: Stochastic gradient descent with warm restarts. *ICLR*, 2017.

Loshchilov, I. and Hutter, F. Decoupled weight decay regularization. In *International Conference on Learning Representations*, 2019. URL https://openreview.net/forum?id=Bkg6RiCqY7.

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

Real, E., Moore, S., Selle, A., Saxena, S., Suematsu, Y. L., Le, Q., and Kurakin, A. Large-scale evolution of image classifiers. In *ICML*, 2017.

Real, E., Aggarwal, A., Huang, Y., and Le, Q. V. Regularized evolution for image classifier architecture search. *arXiv preprint arXiv:1802.01548*, 2018.

Sciuto, C., Yu, K., Jaggi, M., Musat, C., and Salzmann, M. Evaluating the search phase of neural architecture search. *CoRR*, abs/1902.08142, 2019. URL http://arxiv.org/abs/1902.08142.

Shahriari, B., Swersky, K., Wang, Z., Adams, R. P., and de Freitas, N. Taking the human out of the loop: A review of bayesian optimization. *Proceedings of the IEEE*, 104 (1):148–175, 2016.

Snoek, J., Larochelle, H., and Adams, R. P. Practical bayesian optimization of machine learning algorithms. In *NIPS*, 2012.

Snoek, J., Rippel, O., Swersky, K., Kiros, R., Satish, N., Sundaram, N., Patwary, M., Prabhat, and Adams, R. Scalable Bayesian optimization using deep neural networks. In *Proceedings of the 32nd International Conference on Machine Learning (ICML'15)*, 2015.

So, D. R., Liang, C., and Le, Q. V. The evolved transformer. *CoRR*, abs/1901.11117, 2019.

Springenberg, J. T., Klein, A., Falkner, S., and Hutter, F. Bayesian optimization with robust bayesian neural networks. In *Proceedings of the 29th International Conference on Advances in Neural Information Processing Systems (NIPS'16)*, 2016.

Stadler, P. F. Landscapes and their correlation functions. *Journal of Mathematical chemistry*, 20(1):1–45, 1996.

Sutskever, I., Vinyals, O., and Le, Q. V. Sequence to sequence learning with neural networks. In *NIPS*, 2014.

Szegedy, C., Liu, W., Jia, Y., Sermanet, P., Reed, S., Anguelov, D., Erhan, D., Vanhoucke, V., and Rabinovich, A. Going deeper with convolutions. In *CVPR*, 2015.

Szegedy, C., Vanhoucke, V., Ioffe, S., Shlens, J., and Wojna, Z. Rethinking the inception architecture for computer vision. In *CVPR*, 2016.

Tieleman, T. and Hinton, G. Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude. *COURSERA: Neural networks for machine learning*, 2012.

Weinberger, E. Correlated and uncorrelated fitness landscapes and how to tell the difference. *Biological cybernetics*, 63(5):325–336, 1990.

Williams, R. J. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine Learning*, 8:229–256, 1992. doi: 10.1007/BF00992696. URL https://doi.org/10.1007/BF00992696.

Ying, C. Enumerating unique computational graphs via an iterative graph invariant. *CoRR*, abs/1902.06192, 2019.

Zoph, B. and Le, Q. V. Neural architecture search with reinforcement learning. In *ICLR*, 2016.

Zoph, B., Vasudevan, V., Shlens, J., and Le, Q. V. Learning transferable architectures for scalable image recognition. In *CVPR*, 2018.

NAS-Bench-101: Towards Reproducible Neural Architecture Search

Supplementary Material

S1. Identifying Isomorphic Cells

Within the NAS-Bench-101 search space of models, there are models which have different adjacency matrices or have different labels but are computationally equivalent (e.g., Figure 1). We call such cells *isomorphic*. Furthermore, vertices not on a path from the input vertex to the output vertex do not contribute to the computation of the cell. Cells with such vertices can be pruned to smaller cell without changing the effective behavior of the cell in the network. Due to the size of the search space, it would be computationally intractable (and wasteful) to evaluate each possible graph representation without considering isomorphism.

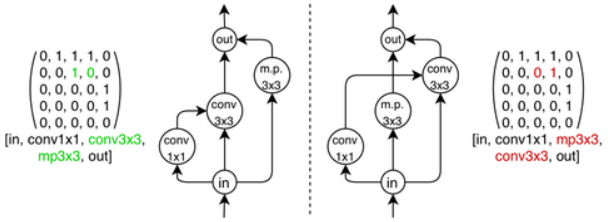

Figure 1: Two cells which are represented differently according to their adjacency matrix and labels but encode the same computation.

Thus, we utilize an iterative graph hashing algorithm, described in (Ying, 2019), which quickly determines whether two cells are isomorphic. To summarize the algorithm, we iteratively perform isomorphism-invariant operations on the vertices of the graph which incorporates information from both the adjacent vertices as well as the vertex label. The algorithm outputs a fixed-length hash which uniquely identifies isomorphic cells (i.e., computationally identical graphs cells to the same value and computationally different cells hash to different values).

Using such an algorithm allows us to enumerate all unique cells within the space and choose a single canonical cell to represent each equivalence class of cells and perform the expensive train and evaluation procedure on the canonical cell only. When querying the dataset for a valid model, we first hash the proposed cell then use the hash to return the data associated with the evaluated canonical graph.

batch size	256
initial convolution filters	128
learning rate schedule	cosine decay
initial learning rate	0.2
ending learning rate	0.0
optimizer	RMSProp
momentum	0.9
L2 weight decay	0.0001
batch normalization momentum	0.997
batch normalization epsilon	0.00001
accelerator	TPU v2 chip

Table 1: Important training hyperparamters.

S2. Implementation Details

S2.1. Generating the dataset

Table 1 shows the training hyperparameters used for all models in the space. These values were tuned to be optimal for the average of 50 randomly sampled cells in the search space. In practice, we find that these hyperparameters do not significantly affect the ranking of cells as long as they are set within reasonable ranges.

S2.2. Benchmarked algorithms

All methods employ the same encoding structure as defined in Section 2.2. For each method except random search, which is parameterfree, we identified the method's key hyperparameters and found a well-performing setting by a simple grid search which follows the same experimental protocol as described in the main text. Scripts to reproduce our experiments can be found at https://github.com/automl/nas_benchmarks.

Random search (RS) We used our own implementation of random search which samples architectures simply from a uniform distribution over all possible configurations in the configuration space.

Regularized evolution (RE) We used a publicly available re-implementation for RE (Real et al., 2018). To mutate an architecture, we first sample uniformly at random an

edge or an operator. If we sampled an edge we simply flip it and for operators, we sample a new operator for the set of all possible operations excluding the current one. RE kills the oldest member of the population at each iteration after reaching the population size. We evaluated different values for the population size (PS) and the tournament size (TS) (see Figure 4) and set them to PS=100 and TS=10 for the final evaluation.

Tree-structured Parzen estimator (TPE) We used the Hyperopt implementation from `https://github.com/hyperopt/hyperopt` for TPE. All hyperparameters were left to their defaults, since the open-source implementation does not expose them and, hence, we could not adapt them for the comparison.

Hyperband For Hyperband we used the publicly available implementation from `https://github.com/automl/HpBandSter`. We set η to 3 which is also used in Li et al. (2018) and Falkner et al. (2018). Note that, changing η will lead to different budgets, which are not included in NAS-Bench-101.

BOHB For BOHB we also used the implementation from `https://github.com/automl/HpBandSter`. Figure 3 shows the performance of different values for the fraction of random configurations, the number of samples to optimize the acquisition function, the minimum allowed bandwidth for the kernel density estimator and the factor which is multiplied to the bandwidth. Interestingly, while the minimum bandwidth and the bandwidth-factor do not seem to have an influence, the other parameters help to improve BOHB's performance, especially at the end of the optimization, if they are set to quite aggressive values. For the final evaluation we set the random fraction to 0%, the number of samples to 4, the minimum-bandwidth to 0.3 (default) and the bandwidth factor to 3 (default).

Sequential model-based algorithm configuration (SMAC) We used the implementation from `https://github.com/automl/SMAC3` for SMAC.

As meta-parameters we exposed the fraction of random architecture that are evaluated, the maximum number of function evaluations per architecture and the number of trees of the random forest (see Figure 2). Since the fraction of random configurations does not seem to have an influence on the final performance of SMAC we kept it as its default (33%). Interestingly, a smaller number of trees seems to help and we set it to 5 for the final evaluation. Allowing to evaluate the same configuration multiple times slows SMAC down in the beginning of the search, hence, we keep it at 1.

Reinforcement Learning Figure 5 right shows the effect of the learning rate for our reinforcement learning agent described in Section S4. For the final evaluation we used a learning rate of 0.5.

S3. Encoding

Besides the encoding described in Section 4, we also tried another encoding of the architecture space, which implicitly contains the constraint of a maximum of 9 edges. Instead of having a binary vector for all the 21 possible edges in our graph, we defined for each edge i a numerical parameter in $p_i \in [0, 1]$. Additionally, we defined an integer parameter $N \in 0, ..., 9$. Now, in order to generate an architecture, we pick the N edges with the highest values. The encoding for the operators stays the same.

The advantage of this encoding is that by design no architecture violates the maximum number of edges constraint. The major disadvantage is that some methods, such as regularized evolution or reinforcement learning, are not easily applicable without major changes due to the continuous nature of the search space.

Figure 6 shows the comparison of all the methods that can be trivially applied to this encoding. We used the same setup as described in Section 4. Additionally, we also include Vizier, which is not applicable to the default encoding. All hyperparameters are the same as described in Section S2.2. Interestingly the ranking of algorithms changed compared to the results in Figure 7. TPE achieves a much better performance now than on the default encoding and outperforms SMAC and BOHB. We assume that, since we used the hyperparameters of SMAC and BOHB that were optimized for the default encoding in Section S2.2, they do not translate to this new encoding.

S4. REINFORCE Baseline Approach

We attempted to benchmark a reinforcement learning (RL) approach using a 1-layer LSTM controller trained with PPO, as proposed by Zoph et al. (2018). With no additional hyperparameter tuning, the controller seems to fail to learn to traverse the space and tends to converge quickly to a far-from-optimal configuration. We suspect that one reason for this is the highly conditional nature of the space (i.e., cells with more than 9 edges are "invalid"). Further tuning may be required to get RL techniques to work on NAS-Bench-101, and this constitutes an interesting direction for future work.

We did, however, successfully train a naive REINFORCE-based (Williams, 1992) controller which simply outputs a multinomial probability distribution at each of the 21 possible edges and 5 operations and samples the distribution

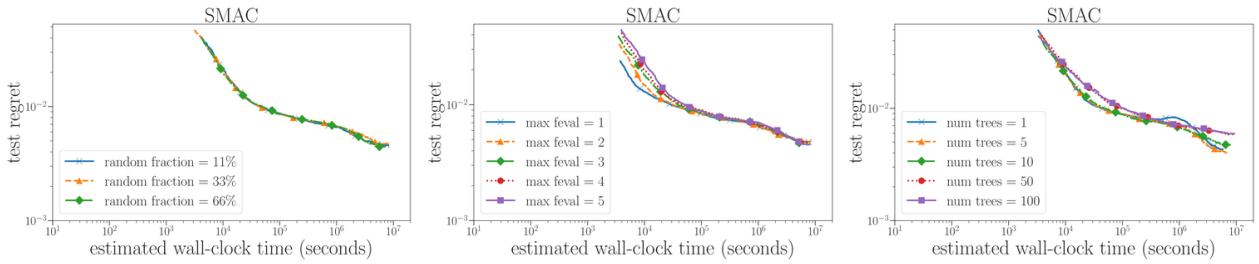

Figure 2: Performance of different meta parameters of SMAC. Left: fraction of random architectures; Middle: maximum number of function evaluations per architecture; Right: Number of trees in the random forest model.

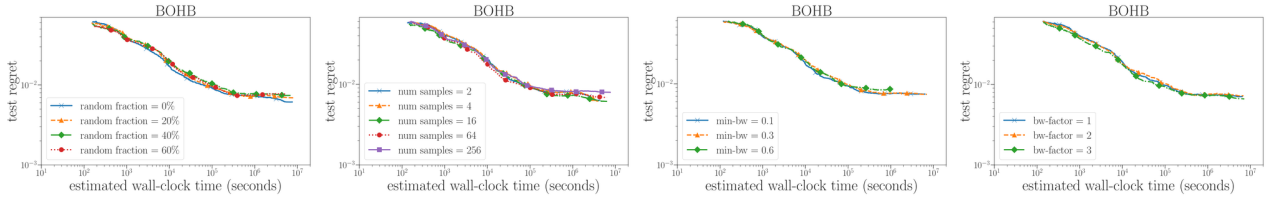

Figure 3: Performance of different meta parameters of BOHB. Left: fraction of random architectures; Middle Left: number of samples to optimize the acquisition function; Middle Right: minimum allowed bandwidth of the kernel density estimator; Right: Factor that is multiplied on the bandwidth for exploration.

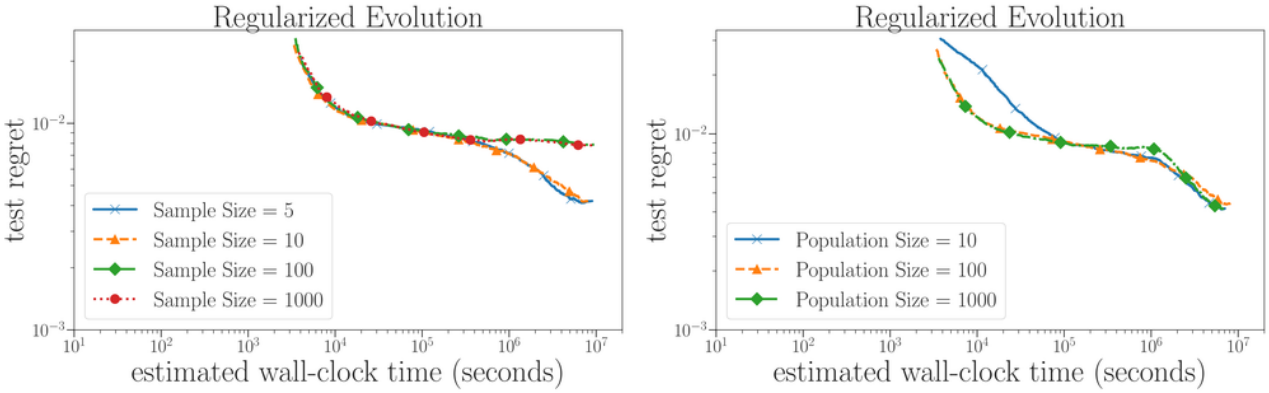

Figure 4: Meta parameters of RE. Left: Tournament Size; Right: Population Size.

to get a new model. We believe that this sampling behavior allows it to find more diverse models than the LSTM-PPO method. The results, when run in the same context as Section 4.2, are shown in Figure 8. REINFORCE appears to perform around as strongly as non-regularized evolution (NRE) but both NRE and REINFORCE tends to be weaker than regularized evolution (RE). All methods beat the baseline random search.

S5. The NAS-HPO-Bench Datasets

The NAS-HPO-Bench datasets consists of 62208 hyperparameter configurations of a 2-layer feedforward networks on four different non-image regression domains, making them complementary to NAS-Bench-101. We varied the number of hidden units, activation types and dropout in each layer as well as the learning rate, batch size and learning rate schedule. While the graph space is much smaller than NAS-Bench-101, it has the important advantage of including hyperparameter choices in the search space, allowing us to measure their interaction and relative importance. For a full description of these datasets, we refer to Klein & Hutter (2019).

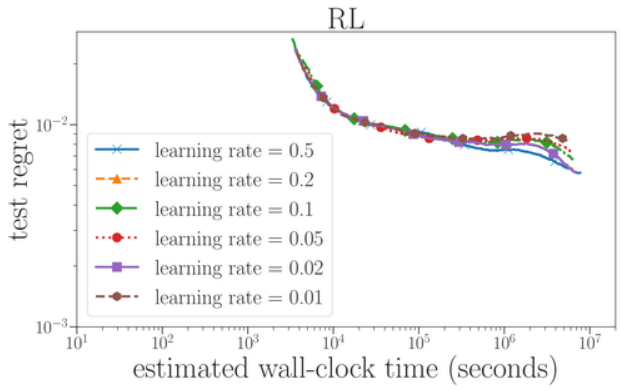

Figure 5: Right: Learning rate of our reinforcement learning agent.

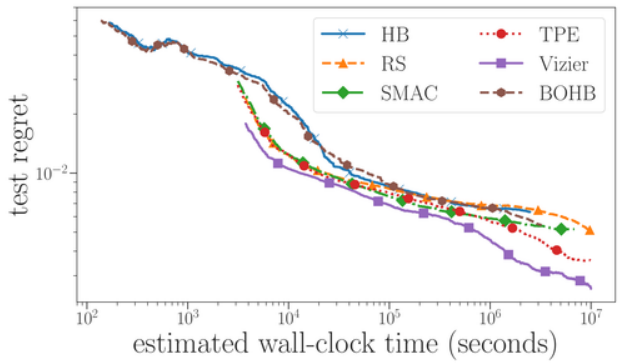

Figure 6: Comparison with a different encoding of architectures (see Section S3 for details). The experimental setup is the same as for Figure 7 in the main text, but note that the hyperparameters of BOHB and SMAC were determined based on the main encoding and are not optimal for this encoding.

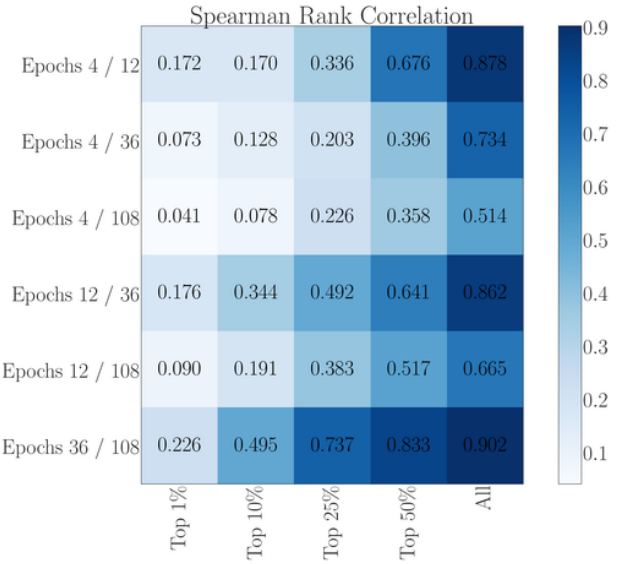

Figure 7: The Spearman rank correlation between accuracy at different number of epoch pairs (rows) for different percentiles of the top architectures (columns) in NAS-Bench-101. *E.g.*, the accuracies between 36 and 108 epochs across the top-10% of architectures have a 0.365 correlation.

S6. Guidelines for Future Benchmarking of Experiments on NAS-Bench-101

To facilitate a standardized use of NAS-Bench-101 in the future benchmarking of algorithms, we recommend the following practices:

1. Perform many runs of the various NAS algorithms (in our experiments, we ran 500).
2. Plot performance as a function of estimated wall clock time and/or number of function evaluations (as in our Figure 7, left). This allows judging the performance of algorithms under different resource constraints. To allow this, every benchmarked algorithm needs to keep track of the best architecture found up to each time step.
3. Do not use test set error during the architecture search process. In particular, the choice of the best architecture found up to each time step can only be based on the training and validation sets. The test error can only be used for offline evaluation once the search runs are complete.
4. To assess robustness of the algorithms with respect to the seed of the random number generator, plot the empirical cumulative distribution of the many runs performed; see our Figure 7 (right) for an example.
5. Compare algorithms using the same hyperparameter settings for NAS-Bench-101 as for other benchmarks. Even though tabular benchmarks like NAS-Bench-101 allow for cheap comprehensive evaluations of different hyperparameter settings (see the next point), in practice NAS algorithms need to come with a set of defaults that the authors propose to use for new NAS benchmarks (or an automated/adaptive method for setting the hyperparameters online); the performance of these defaults should be evaluated.
6. Report performance with different hyperparameter settings to produce a quantitative sensitivity analysis (as in Figures 2-5 of this appendix).
7. If applicable, also study performance for alternative encodings, such as the continuous encoding discussed in Appendix S3.

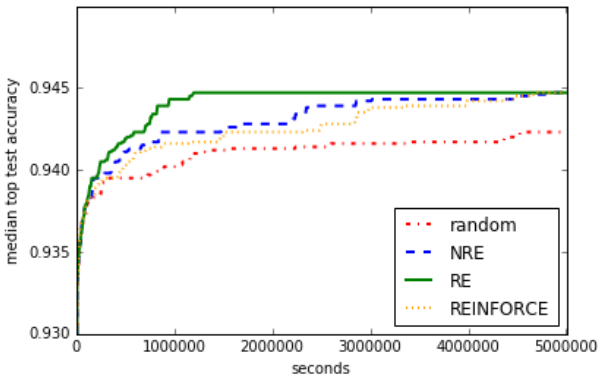

Figure 8: Comparing REINFORCE against regularized evolution (RE), non-regularized evolution (NRE), and a random search baseline (RS).