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.

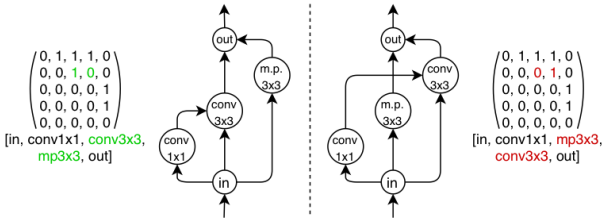

Figure 1: Two cells which are represented differently according to their adjacency matrix and labels but encode the same computation.

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

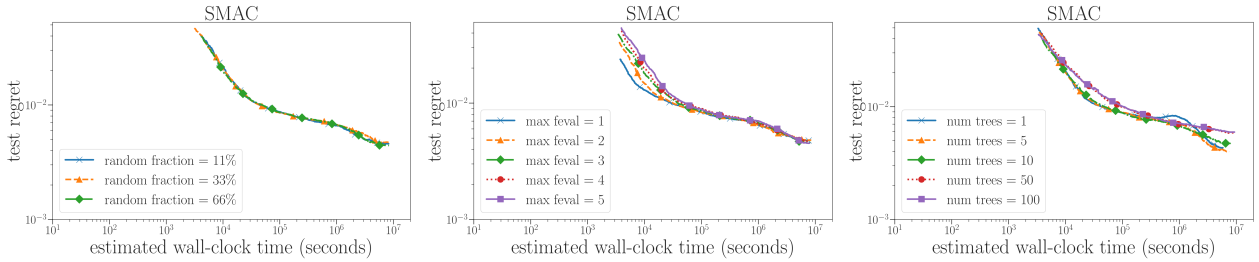

Figure 2: Performance of different meta parameters of SMAC. Left: fraction of random architectures; Middle: maximum number of function evaluations per architecture; Right: Number of trees in the random forest model.

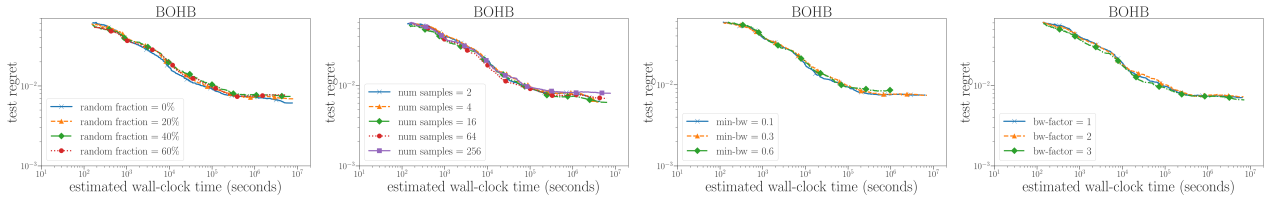

Figure 3: Performance of different meta parameters of BOHB. Left: fraction of random architectures; Middle Left: number of samples to optimize the acquisition function; Middle Right: minimum allowed bandwidth of the kernel density estimator; Right: Factor that is multiplied on the bandwidth for exploration.

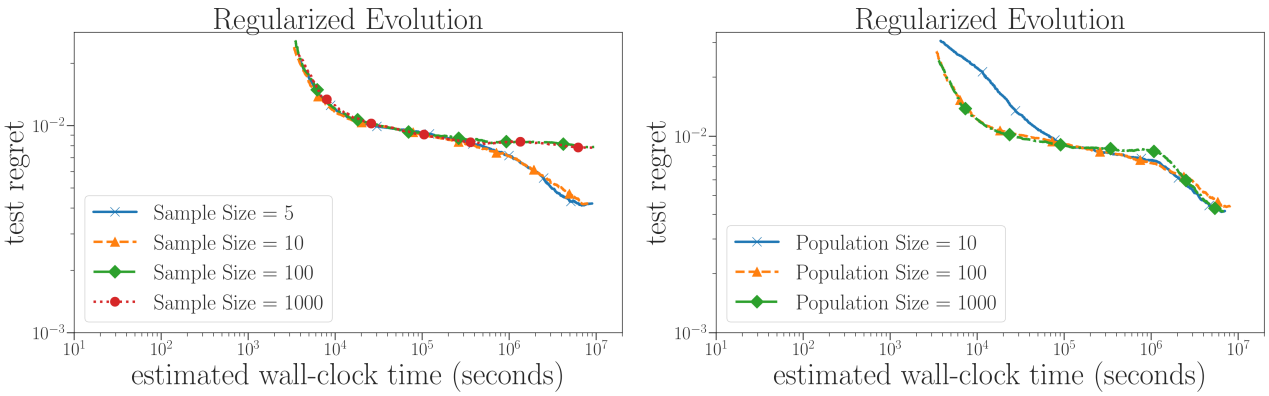

Figure 4: Meta parameters of RE. Left: Tournament Size; Right: Population Size.

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).

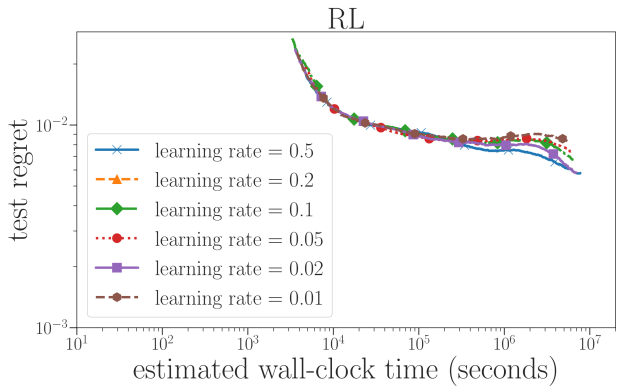

Figure 5: Right: Learning rate of our reinforcement learning agent.

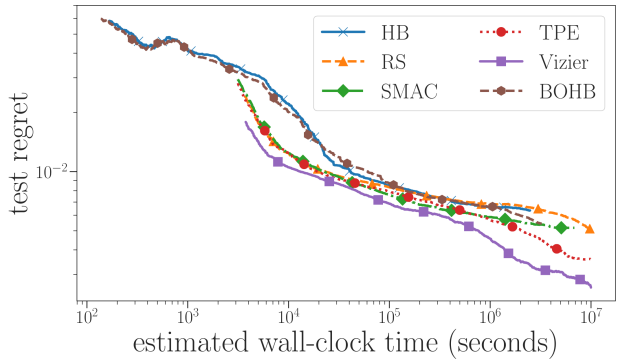

Figure 6: Comparison with a different encoding of architectures (see Section S3 for details). The experimental setup is the same as for Figure 7 in the main text, but note that the hyperparameters of BOHB and SMAC were determined based on the main encoding and are not optimal for this encoding.

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.

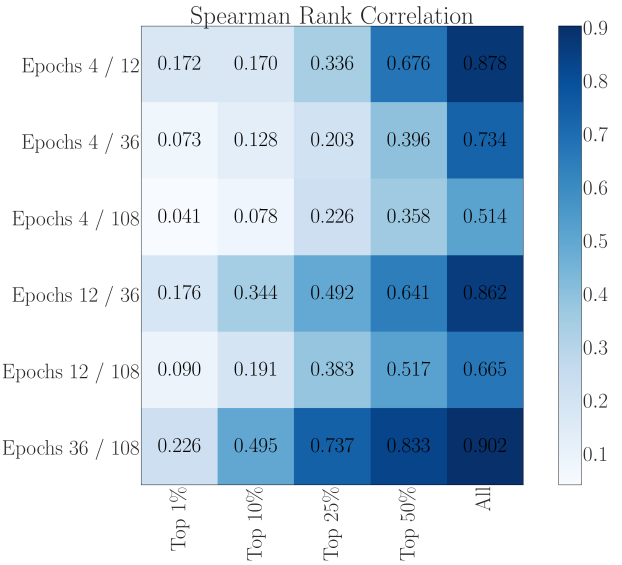

Figure 7: The Spearman rank correlation between accuracy at different number of epoch pairs (rows) for different percentiles of the top architectures (columns) in NAS-Bench-101. *E.g.*, the accuracies between 36 and 108 epochs across the top-10% of architectures have a 0.365 correlation.

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.

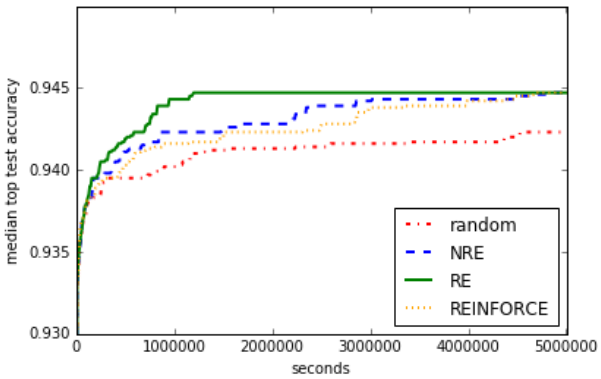

Figure 8: Comparing REINFORCE against regularized evolution (RE), non-regularized evolution (NRE), and a random search baseline (RS).