Evaluating the Search Phase of Neural Architecture Search

Christian Sciuto[*]
AI Lab, Swisscom

Kaicheng Yu[*†]
CV Lab, EPFL

Martin Jaggi
MLO Lab, EPFL

Claudiu Musat
AI Lab, Swisscom

Mathieu Salzmann
CV Lab, EPFL

Abstract

Neural Architecture Search (NAS) aims to facilitate the design of deep networks for new tasks. Existing techniques rely on two stages: searching over the architecture space and validating the best architecture. NAS algorithms are currently evaluated solely by comparing their results on the downstream task. While intuitive, this fails to explicitly evaluate the effectiveness of their search strategies. In this paper, we present a NAS evaluation framework that includes the search phase. To this end, we compare the quality of the solutions obtained by NAS search policies with that of random architecture selection. We find that: (i) On average, the random policy outperforms state-of-the-art NAS algorithms; (ii) The results and candidate rankings of NAS algorithms do not reflect the true performance of the candidate architectures; and (iii) The widely used weight sharing strategy negatively impacts the training of good architectures, thus reducing the effectiveness of the search process. We believe that following our evaluation framework will be key to designing NAS strategies that truly discover superior architectures.

1 Introduction

By automating the design of a neural network for the task at hand, Neural Architecture Search (NAS) has tremendous potential to impact the practicality of deep learning [36, 17, 14, 29, 2]. This is part of a larger trend – automated machine learning (AutoML) – that promises to solve or at least alleviate the scarcity of ML experts needed to design custom architectures. A typical NAS technique [36, 27, 14] has two stages: the search phase, which aims to find a good architecture, and the evaluation one, where the best architecture is trained from scratch and validated on the test data.

Since the search space contains billions of architectures, the fact that NAS algorithms produce networks that achieve state-of-the-art results on the downstream tasks is typically attributed to the search algorithms these methods rely on. While this might seem intuitive, this assumption has never been verified; the effectiveness of the search procedure is never *directly* evaluated.

In this paper, we introduce a framework to evaluate the search phase of NAS algorithms. Our investigation aims to improve the understanding of what causes the state-of-the-art results obtained by NAS discovered networks. In particular, to facilitate the search procedure, state-of-the-art NAS techniques rely on two approximations [27, 18, 20]: a well-designed search space and weight sharing across different architectures. The impact of these schemes cannot be solely analyzed from the final performance on the test data; doing so requires evaluating the search process.

[*]Equal contribution
[†]Correspondence to: kaicheng.yu@epfl.ch

Preprint. Under review.

To this end, we therefore propose to compare the quality of the NAS solutions with a random search policy. The latter uniformly randomly samples an architecture from the same search space as the NAS algorithms, then trains it using the same hyper-parameters as the NAS solutions. To reduce randomness, the search using each policy, i.e., the random and NAS ones, is repeated several times, always changing the initial random seed for different initialization.

We performed experiments on the Penn Tree Bank (PTB) dataset [21], and compared the state-of-the-art NAS algorithms DARTS [18], NAO [20] and ENAS [27] to our random policy according to the standard *perplexity* (PPL) metric [4].

The results of this comparison are surprising:

1. As shown in Table 1, on average, none of 3 NAS algorithms outperforms the random sampling. This shows that the search space of these algorithms has been sufficiently constrained during its construction so that even a random architecture in this space provides good results.

2. The ranking by quality of candidate architectures produced by the NAS algorithms does not reflect the true performance of these architectures. Further investigation of this observation led to our third finding below.

3. Weight sharing, pervasively used to reduce the amount of required resources from thousands of GPU days to a single one, harms the individual networks' performance, and consequently, shuffles the ranking of different architectures, compared to the true ranking obtained by independent training of each architecture. More precisely, using a reduced search space, we show that the architecture rankings obtained with and without weight sharing are entirely uncorrelated, as indicated by a Kendall Tau value of -0.004 on average over 10 different runs.

Table 1: **Typical evaluation of NAS algorithms vs evaluation in this paper.** Existing works typically do not perform extensive comparisons to randomly sampled architectures and, in most cases, do not report the results of their algorithms with different random seeds. Here, we show that a fair and thorough comparison to random architecture selection, using multiple random seeds, is crucial to analyze the effectiveness of search policies. As shown in the last column, where we report the mean validation perplexity (the lower, the better) on the Penn Tree Bank dataset, our evaluation reveals that the state-of-the-art NAS algorithms tend to perform at best on par with random search, emphasizing the need for new search policies.

	Comp. to random	Multiple seeds	Mean valid PPL
ENAS	✗	✗	61.74± 0.78
DARTS	✗	✓	63.35 ± 2.09
NAO	✗	✗	64.18 ± 1.81
Random (Ours)	✓	✓	62.22 ± 0.91

In other words, we disprove the common belief that the quality of architectures trained with weight sharing is similar to that of architectures trained without. We show that the difference in ranking negatively impacts the search phase of NAS algorithms, which, in addition to sensitivity to different seeds, seriously impedes NAS robustness and performance.

In short, evaluating the search phase of NAS, which is typically ignored, allowed us to identify two key characteristics of state-of-the-art NAS algorithms: The importance of the search space and the negative impact of weight sharing. We believe that our evaluation framework will be instrumental in designing NAS search strategies that are superior to the random one.

2 Related Work

Since its introduction in [36], neural architecture search demonstrated great potential to surpass the human design of deep networks for both visual recognition [17, 1, 7, 26, 15] and natural language processing [36, 27, 20, 37, 17, 5]. Existing search strategies include reinforcement learning (RL) samplers [36, 37, 27], evolutionary algorithms [31, 28, 24, 17, 19], gradient-descent [18], and performance predictors [14, 20]. Here, our goal is not to introduce a new search policy, but rather to provide the means to analyze existing ones. Below, we briefly discuss existing NAS methods and focus on how they are typically evaluated.

Neural architecture search with weight sharing. The potential of vanilla NAS comes with the drawback of requiring thousands of GPU hours even for small datasets, such as PTB [22] and CIFAR-10 [12]. Furthermore, even when using such heavy computational resources, vanilla NAS had to restrict the number of trained architectures from a total of 10^9 to 10^4, and increasing the sampler accuracy can only be achieved by increasing the resources.

ENAS [27] was the first to propose a training scheme with sharing parameters, reducing the resources from thousands of GPU days to one. Instead of training from random initialization to convergence,

each sampled model, inherits the parameters from previously-trained ones. Since then, NAS research has mainly focused on two directions: 1) Replacing the RL sampler with a better search algorithm, such as gradient descent [18] and performance predictor based on variational auto-encoder [20]; 2) Exploit NAS for other applications, e.g., object detection [9, 8], compact network [6] and semantic segmentation [16].

Evaluation of NAS algorithms. Typically the quality of NAS algorithms is judged based on the results of the final architecture they produce on the downstream task. In other words, the search and the robustness of these algorithms are generally not studied. In the case of robustness, [18] constitutes the only exception, where results obtained with different random seeds were reported. Here, we aim to further the understanding of the mechanisms behind the search phase of NAS algorithms. Specifically, we propose doing so by comparing them with a simple random search policy, which uniformly randomly samples one architecture per run in the same search space as the NAS techniques.

While some works have provided partial comparisons to random search, these comparisons unfortunately did not give a fair chance to the random policy. Specifically, [27] reports the results of only a single random architecture, and [17] those of an architecture selected among 8 randomly sampled ones as the most promising one after training for 300 epochs only. Here, we show that a fair comparison to the random policy, obtained by training all architectures, i.e., random and NAS ones, for 1000 epochs and averaging over multiple random seeds for robustness, yields a different picture; the state-of-the-art search policies are no better than the random one.

The motivation behind this comparison was our observation of only a weak correlation between the performance of the searched architectures and the ones trained from scratch during the evaluation phase. This phenomenon was already noticed by [35], and concurrently to our work by Li and Talwalkar [13], Xie et al. [32], Ying et al. [34], but the analysis of its impact or its causes went no further. Here, by contrast, we link this difference in performance between the search and evaluation phases to the use of weight sharing.

This may seem to contradict the findings of [3], which, on CIFAR-10, observed a strong correlation between architectures trained with and without weight sharing when searching a CNN cell. However, our work differs from [3] in two fundamental ways: 1) The training scheme in [3], in which the *entire model* with shared parameters is trained via random path dropping, is fundamentally different from those used by state-of-the-arts weight sharing NAS strategies [27, 18, 20]; 2) While the correlation in [3] was approximated using a small subset of sampled architectures, we make use of a reduced search space where we can perform a complete evaluation of *all* architectures, thus providing an exact correlation measure in this space.

Concurrently to our work, [34] introduced a dataset that contains the ground-truth performance of CNN cells, and [30] evaluated some traditional search algorithms on it. While these works support our claim that evaluation of NAS algorithms is crucial, they do not directly evaluate the state-of-the-arts NAS algorithms on the full search space as we do here.

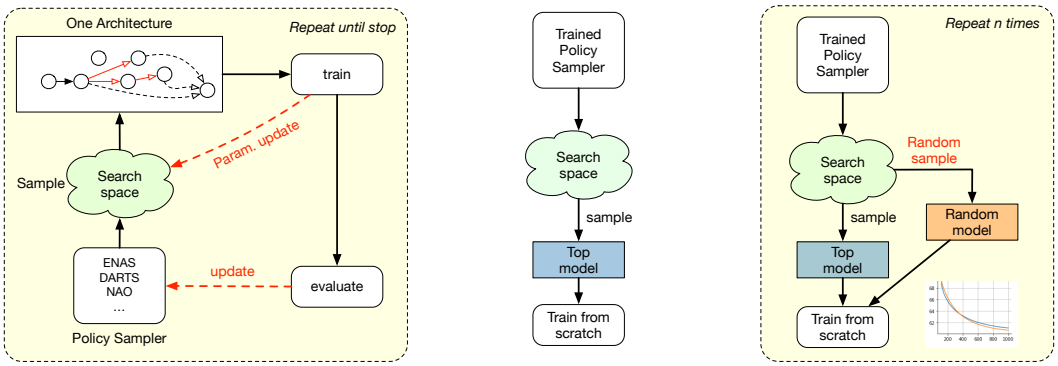
3 Evaluation Framework

In this section, we detail our evaluation framework for the NAS search phase. As depicted in Fig. 1(a,b), typical NAS algorithms consist of two phases:

- **Search:** The goal of this phase is to find the best candidate architecture from the search space[3]. This is where existing algorithms, such as ENAS, DARTS and NAO, differ; they each correspond to a different search algorithm. Nevertheless, for all the algorithms, the search depends heavily on initialization. In all the studied policies, initialization is random and the outcome thus depends on the chosen random seed.

- **Evaluation:** In this phase, all the studied algorithms retrain the best model found in the search phase. The retrained model is then evaluated on the test data.

[3]Details about search space are provided in Appendix A.



(a) Search Phase (b) Standard Evaluation Phase (c) Our Evaluation Phase

Figure 1: **Evaluating NAS.** Existing frameworks consist of two phases: **(a)** The search phase, where a sampler is trained to convergence or a pre-defined stopping criterion; **(b)** The evaluation phase that trains the best model from scratch and evaluates it on the test data. Here, we argue that one should evaluate the search itself. To this end, as shown in **(c)**, we compare the best architecture found by the NAS policy with *a single* uniformly randomly sampled architecture. For this comparison to be meaningful, we repeat it with different random seeds for both training the NAS sampler and our random search policy. We then report the mean and standard deviations over the different seeds.

3.1 Evaluating the NAS Search

As argued above, the standard evaluation of NAS techniques focuses solely on the final results on the test data. Here, by contrast, we aim to evaluate the search phase itself, which is the phase that truly differentiates existing algorithms.

To do this, as illustrated in Fig. 1(c), we establish a baseline; we compare the search phase of existing algorithms with a random search policy. An effective search algorithm should yield a solution that clearly outperforms the random policy. Below, we introduce our framework to compare NAS search algorithms with random search. The three NAS algorithms that we evaluated, DARTS [18], NAO [20] and ENAS [27], are representative of the state of the art for different search algorithms: reinforcement learning, gradient-descent and performance prediction, and are disucssed in more detail in the Appendix B.

3.1.1 Comparing to Random Search

We implement our random search policy by simply assigning uniform probabilities to all operations. Then, for each node in the Directed Acyclic Graph (DAG) that is typically used to represent an architecture [18], we randomly sample a connection to *one* previous node from the resulting distributions.

An effective search policy should outperform the random one. We therefore compare it with each studied NAS algorithm. To this end, as illustrated in Fig. 1(c), we compute the validation results of the best architecture found by the NAS algorithm trained from scratch, as well as those of *a single* randomly sampled architecture.

Comparing these values for a single random seed would of course not provide a reliable measure. Therefore, we repeat this process for multiple random seeds used both during the search phase of the NAS algorithm and to sample one random architecture as described above. We then report the means and standard deviations of these results over the different seeds. Note that while we use different seeds for the search and random sampling, we always use the same seed when training the models from scratch during the evaluation phase.

Our use of multiple random seeds and of the same number of epochs for the NAS algorithm and for our random search policy makes the comparison fair. This contrasts with the comparisons performed in [27], where the results of only *a single random* architecture were reported, and in [18], which selected a single best random architecture among an initial set of 8 after training for 300 epochs only. As shown in the Appendix C, some models that perform well in the early training stages may yield

worse performance than others after convergence. Therefore, choosing the best random architecture after only 300 epochs, and doing so for a single random seed, might not be representative of the general behavior.

3.2 Experimental Setup

In the next section, we compare DARTS [18], NAO [20] and ENAS [27] with our random search policy, as discussed in Section 3.1.1. To this end, following common practice in NAS, we make use of the word-level language modeling Penn Tree Bank (PTB) dataset [22]. The goal then becomes finding a recurrent cell that correctly predicts the next word given the input sequence. The quality of a candidate cell is then evaluated using the standard *perplexity* metric [4].

Note that, for our comparisons, we follow the procedure used in [18, 27, 20] for the final evaluation, consisting of keeping the connections found for the best architecture in the search phase but increasing the hidden state size (to 850 in practice), so as to increase capacity. Furthermore, when training an architecture from scratch, we follow [33, 23] and first make the use of standard SGD to speed up training, and then change to average SGD to improve convergence. For the comparison to be fair, we use the same hidden state size and optimizer in our random search policy.

4 Experimental Results

We analyze the search phase of the three state-of-the-art NAS algorithms discussed above on the task of language modeling. We first compare these algorithms to our random policy when using a search space of 4 activation functions, commonly used in the literature. The surprising findings in this typical NAS use case prompted us to study the behavior of the search strategies in a simplified case, where the search space is reduced to a minimum: 2 nodes. This allows us to identify a factor that has a significant impact on the observed results: Weight sharing. We then quantify this impact on the ranking of the NAS candidates, evidencing that it dramatically affects the effectiveness of the search.

4.1 NAS Comparison in a Standard Search Space

We begin with the standard search space where the best found recurrent cell achieves state-of-the-art performance on the PTB dataset. For the sake of computational cost, as the experiments are repeated multiple times, we follow Liu et al. [18] and define a search space consisting of 8 nodes, instead of the 12 ones [20, 27], which still yields around *2 billion solutions* ($n! * |\mathcal{O}|^n$ solutions, where $n = 8$ nodes and $|\mathcal{O}| = 4$ operations).

Table 2: **Final perplexities in the 8-node search space.** We report the mean and best perplexity on the validation and test sets at the end of training the architectures found using DARTS, NAO, ENAS, as well as for our random policy.

TYPE	MEAN VAL	MEAN TEST	BEST VAL	BEST TEST
DARTS	63.35 ± 2.09	61.10 ± 2.83	**59.84**	57.66
NAO	64.18 ± 1.81	61.99 ± 1.95	62.16	59.77
ENAS	$\mathbf{61.74 \pm 0.78}$	$\mathbf{59.25 \pm 0.68}$	60.38	57.93
RANDOM	62.22 ± 0.91	59.88 ± 0.98	60.18	**57.60**

For each of the four search policies, we run 10 experiments with a different seed. During the search phase, we used the authors-provided hyper-parameters and code for each policy. Once a best architecture is identified by the search phase, it is used for evaluation, i.e., we train the chosen architecture from scratch for 1000 epochs. In this phase, we use a fixed seed for all the tests.

Results. In Figure 2, we plot, on the left, the mean perplexity evolution over the 1000 epochs, obtained by averaging the results of the best architectures found using the 10 consecutive seeds.[4] On the right, we show the perplexity evolution for the best cell of each strategy among the 10 different runs. In each figure, we also show the curve obtained for the random policy. The final results, after 1000 epochs, are further summarized in Table 2.

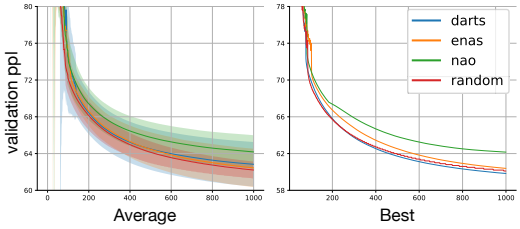


Figure 2: Validation perplexity evolution in the 8-node search space. (Best viewed in color)

[4] Starting from 1268, which is right after 1267, the seed released by Liu et al. [18]. Note that, using this seed, we can reproduce the DARTS search and obtain a validation PPL of 55.7 as in [18].

Observations:

- Random sampling is robust and consistently competitive. As shown in Table 2, it outperforms on average the DARTS and NAO policies, and yields the overall best cell for these experiments. Further training our best random cell for 4000 epochs, as in [18], yields a perplexity of 55.93. typically rely on these spaces, albeit using 12 nodes instead of 8. The influence of such manually-built spaces has never been analyzed before.
- The ENAS policy sampler has the lowest variance among the three tested ones. This shows that ENAS is more robust to the variance caused by the random seed of the search phase. Such a comparison of search policies would not have been possible without our framework.

Encouraged by these surprising results, we then dig deeper into their causes. To this end, below, we make use of a smaller search space that we can explore exhaustively.

4.2 Searching a Reduced Space

The results in the previous section highlight the inability of the studied methods to surpass the random search. To further understand the reason behind this, we repeat the analysis on the smallest possible space of a similar nature, consisting of only two nodes. Given that each node is identified by two values, the ID of the incoming node and the activation function, the space has a cardinality $|S| = n! * |\mathcal{O}|^n$, where $n = 2$ nodes and $|\mathcal{O}| = 4$ operations, thus only having 32 possible solutions.

We benchmark the same three policies against random sampling. Adapting each policy to the reduced space requires the following changes:

- For DARTS, no changes are needed except modifying the number of nodes in the search space.
- For NAO, to mimic the behavior of the algorithm in the space of 8 nodes, we randomly sample 20% of the possible architectures to define the initial candidate pool. We train the encoder-predictor-decoder network for 250 iterations every 50 epochs using the top-4 architectures in the NAO ranking. At each search iteration, we sample at most 3 new architectures to be added to the pool. The rest of the search logic remains unchanged.
- For ENAS, we reduce the number of architectures sampled in one epoch to 20 and increase the number of batches to 10 for each architecture. All other hyper-parameters are unchanged.

Results. In Table 3, we provide the results of searching the 2-node space. Its smaller size allows us to exhaustively compute the validation and test results of all possible solutions, thus determining the upper bound for this case. In Figure 3, we plot the top 1 architecture discovered by the three NAS algorithms for each of the 10 different runs.

Table 3: **Final perplexities in the 2-node search space.** We report the mean and best perplexity on the validation and test sets at the end of training the architectures found using DARTS, NAO, ENAS, as well as for our random policy. We also report the real best results obtained by exhaustive search.

TYPE	MEAN VALID	MEAN TEST	BEST VALID	BEST TEST
DARTS	71.29 ± 2.45	68.74 ± 2.42	68.05	65.55
NAO	$\mathbf{68.66 \pm 2.50}$	$\mathbf{66.03 \pm 2.40}$	66.22	63.59
ENAS	69.99 ± 0.0	66.61 ± 0.0	69.99	66.61
RANDOM	69.69 ± 2.44	67.21 ± 2.52	**66.16**	**63.27**
REAL-BEST	65.67 ± 0.21	62.95 ± 0.18	65.38	62.63

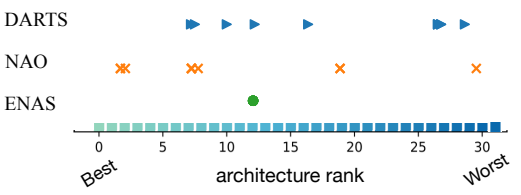


Figure 3: **Architectures discovered by NAS algorithms.** We rank all 32 architectures in the reduced search space based on their performance of individual training, from left (best) to right (worst), and plot the best cell found by the three NAS algorithms across the 10 random seeds. Note that ENAS always converges to the same architecture. The search of DARTS and NAO are sensitive to random initialization. DARTS never discovered a top-5 architecture.

Observations:

- All the policies failed to find the architecture that actually performs best, as in Figure 3.
- The best architecture sampled by any search policy was found by the random search, as in the 8-node case.

(a) Architecture performance with and without WS b) Rank difference due to weight sharing c) Example

Figure 4: **Rank disorder due to weight sharing. (a)** We report the average and std over 10 different runs. Note that the rankings significantly differ between using (line plot) and not using WS (bar plot), showing the negative impact of this strategy. **(b)** We visualize from left to right, the best, worst and average cases, and show the corresponding Kendall Tau value. A change in ranking, indicated by the colors and numbers, is measured as the absolute position change between the WS ranking and the true one. For conciseness, we only show the top 10 architectures. **(c)** For example, in the average scenario, the 6-th best architecture is wrongly placed as the best one, as indicated by the red arrow.

- Even in a space this small, on average, the random policy still yields comparable results to the other search strategies.
- The ENAS policy always converged to the same architecture, as shown in Figure 3. This further evidences the robustness of ENAS to the random seed.
- NAO performs better than random sampling on average because it keeps the ranking of sampled architectures. Nevertheless, the final architecture chosen by NAO is *always* one of the architectures from the initial pool, which were sampled uniformly randomly. This indicates that the ranking of NAO is not updated in an appropriate manner throughout the search.

For our random policy, the probability that it cannot find the best architecture in 10 runs is $(1 - 1/32)^{10} \approx 72.79\%$ and thus our first observation should not be surprising in this case. This observation, however, is surprising for the three NAS methods which implement different search algorithms. There are only two aspects shared between them, the search space and the weight sharing scheme, and we study their impact in detail below.

4.3 Impact of Search Space and Weight Sharing

We see two potential reasons to explain the failure of NAS algorithms: 1) All models in the reduced search space performs similarly, thus finding any of them is a success; and 2) Weight sharing reduces the correlation between the ranking in the search phase and the ranking of the evaluation phase. To analyze this, we perform the following experiments:

Without WS: We train all the 32 possible architectures of the search space individually. Each architecture is trained 10 times with a different seed, which therefore yields a mean and standard deviation of its performance. The mean value is used as a ground truth – the actual potential of the given architecture.

With WS: We train the 32 architectures in parallel, using the weight sharing strategy employed in NAO and ENAS. As DARTS does not have discrete representations of the solutions during the search, the idea of solution ranking does not apply at that point. During training, each mini-batch is given to an architecture uniformly sampled from the search space. We repeat the process 10 times, with 10 random seeds and train the shared weights for 1000 epochs.

We then compute the correlation between the architecture rankings found with WS and the ground truth (i.e., the architectures trained independently). As a correlation measure, we make use of the *Kendall Tau (τ)* metric [11]: a number in the range [-1, 1] with the following properties:

- $\tau = -1$: Maximum disagreement. One ranking is the opposite of the other.
- $\tau = 1$: Maximum agreement. The two rankings are identical.
- τ close to 0: A value close to zero indicates the absence of correlation.

For each of the 10 runs of the weight sharing strategy, we evaluate the Kendall Tau metric of the final rankings with respect to the real averaged ranking.

Results. In Figure 4(a), we depict the architecture performance obtained without WS (sorted in ascending order of average validation perplexity), and the corresponding performance with WS. In Figure 4(b), we show the rank difference, where the best and worst were found using the Kendall Tau metric, and show a concrete rank change example in Figure 4(c).

Observations:

- The difference of architecture performance is *not* related to the use of different random seeds, as indicated by the error bars in Figure 4(a).
- The behavior of the WS rankings is greatly affected by the changing of the seed. In particular, the Kendall Tau values for the three plots in Figure 4(b) are $0.282, -0.004, -0.116$ for Best, Average and Worst.
- For all of the 10 runs, the Kendall Tau metrics are close to zero, which suggests a lack of correlation between the WS rankings and the true one.

These results were obtained using the evaluation hyper-parameters to train all models, including the ones with WS. Nevertheless, we observed the same behavior when using the standard search-phase hyper-parameters, which confirms that our findings are not due to a poor hyper-parameter choice.

4.3.1 Influence of the Amount of Sharing

Depending on the active connections in the DAG, different architectures are subject to different amounts of weight sharing. In Figure 5, let us consider the 3-node case, with node 1 and node 2 fixed and node 3 having node 1 as incoming node. In this scenario, the input to node 3 can be either directly node 0 (i.e., the input), or node 1, or node 2. In the first case, the only network parameters that the output of node 3 depends on are the weights of its own operation. In the second and third cases, however, the output further depends on the parameters of node 1, and of nodes 1 and 2, respectively.

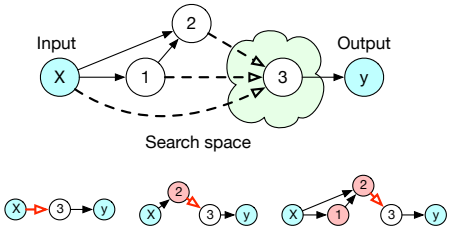


Figure 5: **Top**: Reduced search space. **Bottom**: The amount of sharing depends on the activated path.

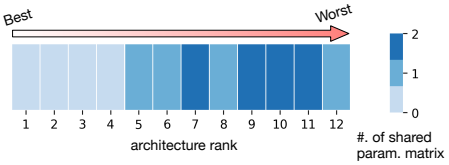


Figure 6: Impact of weight sharing.

To study the influence of the amount of sharing on the architecture ranking, we performed an experiment where we fixed the first two nodes and only searched for the third one. This represents a space of 12 architectures (3 possible connections to node 3 $\times$ 4 operations). We train them using the same setting in Section 4.3. The ranking of the 12 architectures is shown in Figure 6, where color indicates the number of shared weight matrices, that is, matrices of nodes 1 and 2 also used in the search for node 3. Note that the top-performing architectures do not share any weights and that the more weights are shared, the worse the architecture performs.

Together with the previous observations, we believe that these results evidence the negative impact of WS; it dramatically affects the performance of the sampled architectures, thus complicating the overall search process and leading to search policies that are no better than the random one.

5 Conclusion

In this paper, we have provided a thorough analysis of the effectiveness of the search phase of NAS algorithms, including proper and fair comparisons to random search. We have observed that, surprisingly, the search policies of state-of-the-art NAS techniques are no better than random, and have traced the reason for this to the use of (i) a constrained search space and (ii) weight sharing, which shuffles the architecture ranking during the search, thus negatively impacting it.

In essence, our gained insights highlight two key properties of state-of-the-art NAS strategies, which had been overlooked in the past due to the single-minded focus of NAS evaluation on the results on

the target tasks. We believe that this will be key to the development of novel search policies for NAS. This will be the focus of our future research.

References

[1] K. Ahmed and L. Torresani. Maskconnect: Connectivity learning by gradient descent. In *Proceedings of the European Conference on Computer Vision (ECCV)*, pages 349–365, 2018.

[2] B. Baker, O. Gupta, N. Naik, and R. Raskar. Designing neural network architectures using reinforcement learning. *arXiv preprint arXiv:1611.02167*, 2016.

[3] G. Bender, P.-J. Kindermans, B. Zoph, V. Vasudevan, and Q. Le. Understanding and simplifying one-shot architecture search. In *International Conference on Machine Learning*, pages 549–558, 2018.

[4] P. F. Brown, V. J. D. Pietra, R. L. Mercer, S. A. D. Pietra, and J. C. Lai. An estimate of an upper bound for the entropy of english. *Computational Linguistics*, 18(1):31–40, 1992.

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

[6] H. Cai, L. Zhu, and S. Han. ProxylessNAS: Direct Neural Architecture Search on Target Task and Hardware. *arXiv:1812.00332 [cs, stat]*, Dec. 2018. URL `http://arxiv.org/abs/1812.00332`. arXiv: 1812.00332.

[7] L.-C. Chen, M. Collins, Y. Zhu, G. Papandreou, B. Zoph, F. Schroff, H. Adam, and J. Shlens. Searching for efficient multi-scale architectures for dense image prediction. In *Advances in Neural Information Processing Systems*, pages 8713–8724, 2018.

[8] Y. Chen, T. Yang, X. Zhang, G. Meng, C. Pan, and J. Sun. DetNAS: Neural Architecture Search on Object Detection. *arXiv:1903.10979 [cs]*, Mar. 2019. URL `http://arxiv.org/abs/1903.10979`. arXiv: 1903.10979.

[9] G. Ghiasi, T.-Y. Lin, R. Pang, and Q. V. Le. NAS-FPN: Learning Scalable Feature Pyramid Architecture for Object Detection. *arXiv:1904.07392 [cs]*, Apr. 2019. URL `http://arxiv.org/abs/1904.07392`. arXiv: 1904.07392.

[10] S. Hochreiter and J. Schmidhuber. Long short-term memory. *Neural computation*, 9(8): 1735–1780, 1997.

[11] M. G. Kendall. A new measure of rank correlation. *Biometrika*, 30(1/2):81–93, 1938.

[12] A. Krizhevsky, V. Nair, and G. Hinton. Cifar-10 (canadian institute for advanced research). 2009.

[13] L. Li and A. Talwalkar. Random search and reproducibility for neural architecture search. *arXiv preprint arXiv:1902.07638*, 2019.

[14] C. Liu, B. Zoph, M. Neumann, J. Shlens, W. Hua, L.-J. Li, L. Fei-Fei, A. Yuille, J. Huang, and K. Murphy. Progressive neural architecture search. In *Proceedings of the European Conference on Computer Vision (ECCV)*, pages 19–34, 2018.

[15] C. Liu, L.-C. Chen, F. Schroff, H. Adam, W. Hua, A. Yuille, and L. Fei-Fei. Auto-deeplab: Hierarchical neural architecture search for semantic image segmentation. *arXiv preprint arXiv:1901.02985*, 2019.

[16] C. Liu, L.-C. Chen, F. Schroff, H. Adam, W. Hua, A. Yuille, and L. Fei-Fei. Auto-DeepLab: Hierarchical Neural Architecture Search for Semantic Image Segmentation. *arXiv:1901.02985 [cs]*, 2019. URL `http://arxiv.org/abs/1901.02985`.

[17] H. Liu, K. Simonyan, O. Vinyals, C. Fernando, and K. Kavukcuoglu. Hierarchical representations for efficient architecture search. *International Conference on Learning Representations (ICLR), Conference track*, 2018.

[18] H. Liu, K. Simonyan, and Y. Yang. Darts: Differentiable architecture search. *ICLR*, 2019.

[19] Z. Lu, I. Whalen, V. Boddeti, Y. Dhebar, K. Deb, E. Goodman, and W. Banzhaf. Nsga-net: A multi-objective genetic algorithm for neural architecture search. *arXiv preprint arXiv:1810.03522*, 2018.

[20] R. Luo, F. Tian, T. Qin, E.-H. Chen, and T.-Y. Liu. Neural architecture optimization. In *Advances in neural information processing systems*, 2018.

[21] M. Marcus, G. Kim, M. A. Marcinkiewicz, R. MacIntyre, A. Bies, M. Ferguson, K. Katz, and B. Schasberger. The penn treebank: Annotating predicate argument structure. In *Proceedings of the Workshop on Human Language Technology*, HLT '94, pages 114–119, Stroudsburg, PA, USA, 1994. Association for Computational Linguistics. ISBN 1-55860-357-3. doi: 10.3115/1075812.1075835. URL `https://doi.org/10.3115/1075812.1075835`.

[22] M. Marcus, G. Kim, M. A. Marcinkiewicz, R. MacIntyre, A. Bies, M. Ferguson, K. Katz, and B. Schasberger. The penn treebank: Annotating predicate argument structure. In *Proceedings of the Workshop on Human Language Technology*, pages 114–119. Association for Computational Linguistics, 1994.

[23] S. Merity, N. S. Keskar, and R. Socher. Regularizing and optimizing lstm language models. *arXiv preprint arXiv:1708.02182*, 2017.

[24] R. Miikkulainen, J. Liang, E. Meyerson, A. Rawal, D. Fink, O. Francon, B. Raju, H. Shahrzad, A. Navruzyan, N. Duffy, et al. Evolving deep neural networks. In *Artificial Intelligence in the Age of Neural Networks and Brain Computing*, pages 293–312. Elsevier, 2019.

[25] V. Nair and G. E. Hinton. Rectified linear units improve restricted boltzmann machines. In *Proceedings of the 27th international conference on machine learning (ICML-10)*, pages 807–814, 2010.

[26] J.-M. Pérez-Rúa, M. Baccouche, and S. Pateux. Efficient progressive neural architecture search. *arXiv preprint arXiv:1808.00391*, 2018.

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

[28] E. Real, S. Moore, A. Selle, S. Saxena, Y. L. Suematsu, J. Tan, Q. Le, and A. Kurakin. Large-scale evolution of image classifiers. *arXiv preprint arXiv:1703.01041*, 2017.

[29] M. Tan, B. Chen, R. Pang, V. Vasudevan, and Q. V. Le. Mnasnet: Platform-aware neural architecture search for mobile. *arXiv preprint arXiv:1807.11626*, 2018.

[30] L. Wang, Y. Zhao, Y. Jinnai, Y. Tian, and R. Fonseca. AlphaX: eXploring Neural Architectures with Deep Neural Networks and Monte Carlo Tree Search. *arXiv preprint arXiv:1903.11059*, 2019.

[31] L. Xie and A. Yuille. Genetic cnn. *IEEE International Conference on Computer Vision (ICCV)*, 2017.

[32] S. Xie, A. Kirillov, R. Girshick, and K. He. Exploring randomly wired neural networks for image recognition. *arXiv preprint arXiv:1904.01569*, 2019.

[33] Z. Yang, Z. Dai, R. Salakhutdinov, and W. W. Cohen. Breaking the softmax bottleneck: A high-rank rnn language model. *arXiv preprint arXiv:1711.03953*, 2017.

[34] C. Ying, A. Klein, E. Real, E. Christiansen, K. Murphy, and F. Hutter. Nas-bench-101: Towards reproducible neural architecture search. *arXiv preprint arXiv:1902.09635*, 2019.

[35] A. Zela, A. Klein, S. Falkner, and F. Hutter. Towards automated deep learning: Efficient joint neural architecture and hyperparameter search. *ICML AutoML Workshop*, 2018.

[36] B. Zoph and Q. V. Le. Neural Architecture Search with Reinforcement Learning. *International Conference on Learning Representations (ICLR), Conference track*, 2017.

[37] B. Zoph, V. Vasudevan, J. Shlens, and Q. 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.

A NAS Search Space Representation

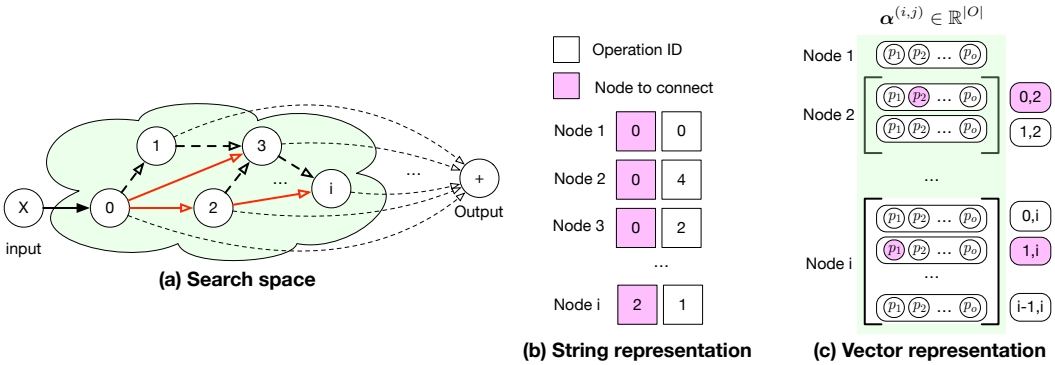


Figure 7: **Search space of NAS algorithms.** Typically, the search space is encoded as **(a)** a directed acyclic graph, and an architecture can be represented as **(b)** a string listing the node ID that each node is connected to, or the operation ID employed by each node. **(c)** An alternatively representation is a list of vectors α of size $\frac{n(n+1)}{2}|\mathcal{O}|$, where n is the number of nodes and $\mathcal{O}$ is the set of all operations. Each vector, $\alpha^{(i,j)}$, captures, via a softmax, the probability p_o that operation o is employed between node i and j. Note that any node only takes one incoming edge, thus (b) and (c) represent the same search space and only differs in its formality.

As discussed in the main paper, our starting point is a neural search space for a recurrent architecture, as illustrated in Figure 7. A convolutional cell can be represented with a similar topological structures. Following common practice in NAS [36], a candidate architecture sampled from this space connects the input and the output nodes through a sequence of intermediary ones. Each node is connected to others and has an operation attached to it.

A way of representing this search space [27, 20], depicted in Figure 7(b), is by using strings. Each character in the string indicates either the node ID that the current node is connected to, or the operation selected for the current node. Operations include the identity, sigmoid, tanh and ReLU [25].

Following the alternative way introduced in [18], we make use of a vectorized representation of these strings. More specifically, as illustrated by Figure 7(c), a node ID, resp. an operation, is encoded as a vector of probabilities over all node IDs, resp. all operations. For instance, the connection between nodes i and j is represented as $y^{(i,j)}(x) = \sum_{o \in \mathcal{O}} p_o o(x)$, with $\mathcal{O}$ the set of all operations, and $p_o = \texttt{softmax}(\alpha_o) = \exp(\alpha_o) / \sum_{o' \in \mathcal{O}} \exp(\alpha_{o'})$ the probability of each operation.

B NAS Algorithms

Here, we discuss the three state-of-the-art NAS algorithms used in our experiments in detail.

ENAS adopts a reinforcement learning sampling strategy that is updated with the REINFORCE algorithm. The sampler is implemented as a two-layer LSTM [10] and generates a sequence of strings. In the training process, each candidate sampled by the ENAS controller is trained on an individual mini-batch. At the end of each epoch, the controller samples new architectures that are evaluated on a single batch of the validation dataset. After this, the controller is updated accordingly using these validation metrics.

DARTS vectorizes the aforementioned strings as discussed in Section **??** and shown in Fig. 7(c). The sampling process is then parameterized by the vector α, which is optimized via gradient-descent in a dual optimization scheme: The architecture is first trained while fixing α, and α is then updated

while the network is fixed. This process is repeated in an alternating manner. In the evaluation phase, DARTS samples the top-performing architecture by using the trained α vector as probability prior, i.e., the final model is not a soft average of all paths but one path in the DAG, which makes its evaluation identical to that of the other NAS algorithms.

NAO implements a gradient-descent algorithm, but instead of vectorizing the strings as in DARTS, it makes use of a variational auto-encoder (VAE) to learn a latent representation of the candidate architectures. Furthermore, it uses a performance predictor, which takes a latent vector as input to predict the corresponding architecture performance. In short, the search phase of NAO consists of first randomly sampling an initial pool of architectures and training them so as to obtain a ranking. This ranking is then used to train the encoder-predictor-decoder network, from which new candidates are sampled, and the process is repeated in an iterative manner. The best architecture is then taken as the top-1 in the NAO ranking.

C Random Sampling Comparison

As discussed before, the random policy in [18] samples 8 architectures, and picks the best after training them for 300 epochs independently. It might seem contradictory that DARTS outperforms this random policy, but cannot surpass the much simpler one designed in our paper, which only randomly samples 10 architectures (1 per random seed), trains them to convergence and picks the best. However, the random policy in DARTS relies on the assumption that a model that performs well in the early training stage will remain effective until the end of training. While this may sound intuitive, we observed a different picture with our reduced search space.

Since we obtained the ground-truth performance ranking, as discussed in Section 4.2 of the main paper, in Figure 8, we plot the evolution of models' rank while training proceeds, based on the average validation perplexity over 10 runs. Clearly, there are significant variations during training: Good models in early stages drop lower in the ranking towards the end. As such, there is a non-negligible chance that the random policy in DARTS picks a model whose performance will be sub-optimal. We therefore believe that our policy that simply samples one model and trains it until convergence yields a more fair baseline. Furthermore, the fact that we perform our comparison using 10 random seeds, for both our approach and the NAS algorithms, vs a single one in[18] makes our conclusions more reliable.

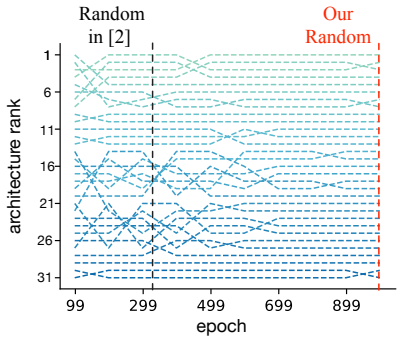


Figure 8: **Rank changes while training**. Each line represents the evolution of the rank of a single architecture. The models are sorted based on their test performance after 1000 epochs, with the best-performing one at the top. The curves were averaged over 10 runs. They correspond to the experiment in Section 4.2. The vertical dashed lines indicate the epoch number where random sampling was performed, either by the random policy in [18], or by ours.