Random Search and Reproducibility for Neural Architecture Search

Liam Li ME@LIAMCLI.COM
Carnegie Mellon University

Ameet Talwalkar TALWALKAR@CMU.EDU
Carnegie Mellon University

Abstract

Neural architecture search (NAS) is a promising research direction that has the potential to replace expert-designed networks with learned, task-specific architectures. In order to help ground the empirical results in this field, we propose new NAS baselines that build off the following observations: (i) NAS is a specialized hyperparameter optimization problem; and (ii) random search is a competitive baseline for hyperparameter optimization. Leveraging these observations, we evaluate both random search with early-stopping and a novel random search with weight-sharing algorithm on two standard NAS benchmarks—PTB and CIFAR-10. Our results show that random search with early-stopping is a competitive NAS baseline, e.g., it performs at least as well as ENAS, a leading NAS method, on both benchmarks. Additionally, random search with weight-sharing outperforms random search with early-stopping, achieving a state-of-the-art NAS result on PTB and a highly competitive result on CIFAR-10. Finally, we explore the existing reproducibility issues of published NAS results.

1. Introduction

Deep learning offers the promise of bypassing the process of manual feature engineering by learning representations in conjunction with statistical models in an end-to-end fashion. However, neural network architectures themselves are typically designed by experts in a painstaking, ad-hoc fashion. Neural architecture search (NAS) presents a promising path for alleviating this pain by automatically identifying architectures that are superior to hand-designed ones. Since the work by Zoph and Le [50], there has been explosion of research activity on this problem [30; 31; 38; 11; 42; 1; 23; 5; 40; 49; 33; 46; 7]. Notably, there has been great industry interest in NAS, as evidenced by the vast computational [50; 51; 42] and marketing resources [17] committed to industry-driven NAS research. However, despite a steady stream of promising empirical results [50; 51; 42; 33; 34; 7], we see three fundamental issues with the current state of NAS research:

Inadequate Baselines. Leading NAS methods exploit many of the strategies that were initially explored in the context of traditional hyperparameter optimization tasks, e.g., evolutionary search [39; 21], Bayesian optimization [43; 3; 20], and gradient-based approaches [2; 35]. Moreover, the NAS problem is in fact a specialized instance of the broader hyperparameter optimization problem. However, in spite of the close relationship between these two problems, existing comparisons between novel NAS methods and standard hyperparameter optimization methods are inadequate. In particular, to the best of our knowledge, no state-of-the-art hyperparameter optimization methods have been evaluated on standard NAS benchmarks. *Without benchmarking against leading hyperparameter optimization baselines, it difficult to quantify the performance gains provided by specialized NAS methods.*

Complex Methods. We have witnessed a proliferation of novel NAS methods, with research progressing in many different directions. New approaches introduce a significant amount of algorithmic complexity in the search process, including complicated training routines [1; 40; 46; 7], architecture transformations [45; 42; 6; 31; 11], and modeling assumptions [23; 25; 49; 5; 30] (see Figure 1 and Appendix A.1 for more details). While many technically diverse NAS methods demonstrate good empirical performance, they often lack corresponding ablation studies [34; 49; 7], and as a result, *it is unclear what NAS component(s) are necessary to achieve a competitive empirical result*.

Lack of Reproducibility. Experimental reproducibility is of paramount importance in the context of NAS research, given the empirical nature of the field, the complexity of new NAS methods, and the steep computational costs associated with empirical evaluation. In particular, there are (at least) two important notions of reproducibility to consider: (1) "exact" reproducibility i.e., whether it is possible to reproduce explicitly reported experimental results; and (2) "broad" reproducibility, i.e., the degree to which the reported experimental results are themselves robust and generalizable. Broad reproducibility is difficult to measure due to the computational burden of NAS methods and the high variance associated with extremal statistics. However, most of the published results in this field do not even satisfy exact reproducibility. *For example, of the 12 papers published since 2018 at NeurIPS, ICML, and ICLR that introduce novel NAS methods (see Table 4), none are exactly reproducible.* Indeed, each fails to provide all of the necessary components for exact reproducibility: architecture search code, model evaluation code, random seeds used for search and evaluation, and documentation for hyperparameter tuning.

While addressing these challenges will require community-wide efforts, in this work we present results that aim to make some initial progress on each of these issues. In particular, our contributions are as follows:

1. We help ground existing NAS results by providing a new perspective on the gap between traditional hyperparameter optimization and leading NAS methods. Specifically, we evaluate a general hyperparameter optimization method combining random search with early-stopping [29] on two standard NAS benchmarks (CIFAR-10 and PTB). With approximately the same amount of compute as DARTS [33], a state-of-the-art (SOTA) NAS method, this simple method provides a much more competitive baseline for both benchmarks: (1) on PTB, random search with early-stopping reaches test perplexity of 56.4 compared to the published result for ENAS [40], a leading NAS method, of 56.3,[1] and (2) for CIFAR-10, random search with early-stopping achieves a test error of 2.85%, whereas the published result for ENAS is 2.89%. While SOTA NAS methods like DARTS still outperform this baseline, our results demonstrate that the gap is not nearly as large as that suggested by published random search baselines on these tasks [40; 33].

2. We identify a small subset of NAS components that are sufficient for achieving good empirical results. We construct a simple algorithm from the ground up starting from vanilla random search, and demonstrate that properly tuned random search with weight-sharing is competitive with much more complicated methods when using similar computational budgets. In particular, we identify the following meta-hyperparameters that impact the behavior of our algorithm: batch size, number of epochs, network size, and number of evaluated architectures. We evaluate our proposed method using the same search space and evaluation scheme as DARTS [33], a leading NAS method. We explore a few modifications of the meta-hyperparameters to improve search quality and make full use of available GPU memory and computational resources, and observe SOTA performance on the PTB benchmark and comparable performance to DARTS on the CIFAR-10 benchmark. We emphasize that we do not perform additional hyperparameter tuning of the final architectures discovered at the end of the search process.

3. We open-source the code, random seeds, and documentation necessary to reproduce our experiments.[2] Our single machine results shown in Table 5 and Table 1 follow a deterministic experimental setup, given a fixed random seed, and satisfy exact reproducibility. For these experiments on the two standard benchmarks, we study the broad reproducibility of our random search with weight-sharing results by repeating our experiments with a different set of random seeds. We observe non-trivial differences across the two independent runs and identify potential sources for these differences. Our results highlight the need for more careful reporting of experimental results, increased transparency of intermediate results, and more robust statistics to quantify the performance of NAS methods.

1. We could not reproduce this result using the final architecture and code provided by the authors.
2. All material available at `https://github.com/liamcli/randomNAS_release`.

2. Experiments

Due to space constraints, we provide relevant background on NAS, discuss related work, and describe our random search with weight-sharing algorithm in sections A.1, A.2, A.3 of the Appendix, respectively. Here, we focus on the empirical studies that we conducted to establish better baselines for NAS.

In line with prior work [50; 40; 33], we consider the two standard benchmarks for neural architecture search: (1) language modeling on the Penn Treebank (PTB) dataset [36] and (2) image classification on CIFAR-10 [27]. For each of these benchmarks, we consider the same search space and use much of the same experimental setups as DARTS [33], and by association SNAS [46], to facilitate a fair comparison of our results to existing work.

To evaluate the performance of random search with weight-sharing (see Appendix A.3 for a description of the algorithm) on these two benchmarks, we proceed in the same three stages as Liu et al. [33]:

Stage 1: Perform architecture search using a smaller proxy network.

Stage 2: Evaluate the best architecture from the first stage by retraining a larger proxyless network from scratch. This stage is used to select the best architecture from multiple trials.

Stage 3: Perform a full evaluation of the best found architecture from the second stage by either training for more epochs (PTB) or training with more seeds (CIFAR-10).

We start with the same meta-hyperparameter settings used by DARTS to train the shared weights. Then, we incrementally modify the meta-hyperparameters identified in Appendix A.3.1 to improve performance until we either reach state-of-the-art performance (for PTB) or match the performance of DARTS and SNAS (for CIFAR-10).

For our evaluation of random search with early-stopping (i.e., ASHA, see Appendix A.2 for more information on the ASHA algorithm) on these two benchmarks, we apply partial training to the stage (2) evaluation network and then select the best architecture for stage (3) evaluation. For both benchmarks, we run ASHA with a starting resource per architecture of $r = 1$ epoch, a maximum resource of 300 epochs, and a promotion rate of $\eta = 4$, indicating the top $1/4$ of architectures will be promoted in each round and trained for $4\times$ more resource.

Due to space limitations, we present the results for the more commonly studied CIFAR-10 benchmark [46; 49] below and defer the results for the PTB benchmark, which mirror that for CIFAR-10, to Appendix A.5. For the PTB benchmark, our results in Table 5 of the Appendix show ASHA to be a competitive baseline for NAS, matching the published performance of the best architecture found by ENAS, and random search with weight-sharing to reach SOTA for NAS methods.

2.1 CIFAR-10 Benchmark

We first present the final search results for the CIFAR-10 benchmark after stage (3) evaluation, and then dive deeper into these results to explore the impact of meta-hyperparameters on stage (2) intermediate results, and finally evaluate associated reproducibility ramifications. A description of the search space that we used as well as more details on the experimental setup can be found in Appendix A.4.

2.1.1 FINAL SEARCH RESULTS

We use the same stage (3) evaluation scheme as that used by Liu et al. [33] to produce Table 1 of their paper. In particular, we train the proxyless network configured according to the best architectures found by different methods with 10 different seeds and report the average and standard deviation. We discuss these results in the context of the three issues—baselines, complex methods, reproducibility—introduced in Section 1.

First, we evaluate the ASHA baseline using 9 GPU days, which is comparable to the 10 GPU days we allotted to our independent run of DARTS. In contrast to the one random architecture evaluated by Pham et al. [40] and the 24 evaluated by Liu et al. [33] for their random search baselines, ASHA evaluated over

Table 1: **CIFAR-10 Benchmark: Comparison with state-of-the-art NAS methods and manually designed networks.** The results are grouped by those for manually designed networks, published NAS methods, and the methods that we evaluated. Models for all methods are trained with cutout. Test error for our contributions are averaged over 10 random seeds. Table entries denoted by "-" indicate that the field does not apply, while entries denoted by "N/A" indicate unknown entries. The search cost is measured in GPU days. Note that the search cost is hardware dependent and the search cost shown for our results are calculated for Tesla P100 GPUs; all other numbers follow those reported by Liu et al. [33].
* We show results for the variants of these networks with comparable number of parameters. Larger versions of these networks achieve lower errors.
# Reported test error averaged over 5 seeds.
† The stage (1) cost shown is that for 1 trial as opposed to the cost for 4 trials shown for DARTS and Random search WS. It is unclear whether multiple trials followed by stage (2) evaluation are required in order to find a good architecture.
‡ Due to the longer evaluation we employ in stage (2) to account unstable rankings, the cost for stage (2) is 1 GPU day for results reported by Liu et al. [33] and 6 GPU days for our results.

		Test Error		**Params**	**Search Cost**			**Comparable**	**Search**
Architecture	**Source**	Best	Average	(M)	Stage 1	Stage 2	Total	**Search Space?**	**Method**
Shake-Shake#	[9]	N/A	2.56	26.2	-	-	-	-	manual
PyramidNet	[47]	2.31	N/A	26	-	-	-	-	manual
NASNet-A#*	[51]	N/A	2.65	3.3	-	-	2000	N	RL
AmoebaNet-B*	[42]	N/A	2.55 ± 0.05	2.8	-	-	3150	N	evolution
ProxylessNAS†	[7]	2.08	N/A	5.7	4	N/A	N/A	N	gradient-based
GHN#†	[49]	N/A	2.84 ± 0.07	5.7	0.84	N/A	N/A	N	hypernetwork
SNAS†	[46]	N/A	2.85 ± 0.02	2.8	1.5	N/A	N/A	Y	gradient-based
ENAS†	[40]	2.89	N/A	4.6	0.5	N/A	N/A	Y	RL
ENAS	[33]	2.91	N/A	4.2	4	2	6	Y	RL
Random search baseline	[33]	N/A	3.29 ± 0.15	3.2	-	-	4	Y	random
DARTS (first order)	[33]	N/A	3.00 ± 0.14	3.3	1.5	1	2.5	Y	gradient-based
DARTS (second order)	[33]	N/A	2.76 ± 0.09	3.3	4	1	5	Y	gradient-based
DARTS (second order)‡	Ours	2.62	2.78 ± 0.12	3.3	4	6	10	Y	gradient-based
ASHA baseline	Ours	2.85	3.03 ± 0.13	2.2	-	-	9	Y	random
Random search WS‡	Ours	2.71	2.85 ± 0.08	4.3	2.7	6	9.7	Y	random

700 architectures in the allotted computation time. The best architecture found by ASHA achieves an average error of 3.03 ± 0.13, which is significantly better than the random search baseline provided by Liu et al. [33] and comparable to DARTS (first order). Additionally, the best performing seed reached a test error of 2.85, which is lower than the published result for ENAS. Similar to the PTB benchmark, these results suggest that the gap between SOTA NAS methods and standard hyperparameter optimization is much smaller than previously reported [40; 33].

Next, we evaluate random search with weight-sharing with tuned meta-hyperparameters (see Section 2.1.2 for details). This method finds an architecture that achieves an average test error of 2.85 ± 0.08, which is comparable to the reported results for SNAS and DARTS, the top 2 weight-sharing algorithms that use a comparable search space, as well as GHN [49]. Note that while the two manually tuned architectures we show in Table 1 outperform the best architecture discovered by random search with weight-sharing, they have over $7\times$ more parameters. Additionally, the best-performing efficient NAS method, ProxylessNAS, uses a larger proxyless network and a significantly different search space than the one we consider. We hypothesize that using a proxyless network and applying random search with weight-sharing to the same search space as ProxylessNAS would further improve our results; we leave this as a direction for future work.

Finally, we examine the reproducibility of the NAS methods using a comparable search space with available code for both architecture search and evaluation (i.e., DARTS and ENAS; to our knowledge, code

Table 2: **CIFAR-10 Benchmark: Comparison of Stage (2) Intermediate Search Results for Weight-Sharing Methods.** In stage (1), random search is run with different settings to train the shared weights. The shared weights are then used to evaluate the indicated number of randomly sampled architectures. In stage (2), the best of these architectures for each trial is then trained from scratch for 600 epochs. We report the performance of the best architecture after stage (2) for each trial for each search method.
† This run was performed using the DARTS code before we corrected for non-determinism (see Appendix A.4).

	Setting				**Trial**					
Method	Epochs	Gradient Clipping	Initial Channels	# Archs Evaluated	1	2	3	4	Best	Average
Reproduced DARTS†	50	5	16	-	2.92	2.77	3.00	3.05	**2.77**	**2.94**
Random (1)	50	5	16	1000	3.25	4.00	2.98	3.58	2.98	3.45
Random (2)	150	5	16	5000	2.93	3.80	3.19	2.96	2.93	3.22
Random (3)	150	1	16	5000	3.50	3.42	2.97	2.95	2.97	3.21
Random (4)	300	1	16	11000	3.04	2.90	3.14	3.09	2.90	3.04
Random (5) Run 1	150	1	24	5000	2.96	3.33	2.83	3.00	**2.83**	**3.03**
Random (5) Run 2†	150	1	24	5000	2.93	3.01	2.70	2.85	2.70	2.88

is not currently available for SNAS). For DARTS, exact reproducibility was not feasible since the code is non-deterministic and Liu et al. [33] do not provide random seeds for the search process; hence, we focus on broad reproducibility of the results. In our independent run, DARTS reached an average test error of 2.78 ± 0.12 compared to the published result of 2.76 ± 0.09. Notably, we observed that the process of selecting the best architecture in stage (2) is unstable when training stage (2) models for only 100 epochs; see Section 2.1.3 for details. Hence, we use 600 epochs in all of our CIFAR experiments, including our independent DARTS run, which explains the discrepancy in stage (2) costs between original DARTS and our independent run.

For ENAS, the published results do not satisfy exact reproducibility due to the same issues as those for DARTS. We show in Table 1 the broad reproducibility experiment conducted by Liu et al. [33] for ENAS; here, ENAS found an architecture that achieved a comparable test error of 2.91 in $8\times$ the reported stage (1) search cost. We then investigated the reproducibility of random search with weight-sharing. We verified exact reproducibility and then examined broad reproducibility by comparing our results to that from an independent run with a different set of random seeds. In this second experiment, the best architecture achieves 2.86 ± 0.09 average test error across 10 trials, compared to an average test error of 2.85 ± 0.08 in the first experiment. While the final results are quite similar across independent runs for both DARTS and random search with weight-sharing, we investigate various sources of discrepancies in Section 2.1.3.

2.1.2 IMPACT OF META-HYPERPARAMETERS

We next detail the meta-hyperparameter settings that we tried in order to reach competitive performance on the CIFAR-10 benchmark via random search with weight-sharing. Similar to DARTS, in these preliminary experiments we performed 4 separate trials of each version of random search with weight-sharing, where each trial consists of executing stage (1) followed by stage (2). In stage (1), we train the shared weights and use them to evaluate a given number of randomly sampled architectures on the test set. In stage (2), we select the best architecture, according to the shared weights, to train from scratch using the proxyless network for 600 epochs.

We incrementally tune random search with weight-sharing by adjusting the following meta-hyperparameters that impact both the training of shared weights and the evaluation of architectures using these trained weights: number of training epochs, gradient clipping, number of architectures evaluated using shared weights, and network size. The settings we consider for random search along with the performance of the final architectures

across 4 trials after retraining from scratch for each of these settings is shown in Table 2. The best setting for random search was Random (5), which has a larger network size. The best trial for this setting reached a test error of 2.83 when retraining from scratch. In light of these stage (2) results, we focus in stage (3) on the best architecture found by Random (5) Run 1, and achieve an average test error of 2.85 ± 0.08 over 10 random seeds as shown in Table 1.

2.1.3 INVESTIGATING REPRODUCIBILITY

Our results in this section suggest that both DARTS and Random (5) are broadly reproducible on this benchmark, which is surprising given the unstable ranking in architectures observed between 100 and 600 epochs for stage (2) evaluation. To begin, the first row of Table 2 shows our reproduced results for DARTS after training the best architecture for each trial from scratch for 600 epochs. In our reproduced run, DARTS reaches an average test error of 2.94 and a minimum of 2.77 across 4 trials (see Table 2). We do not have a direct comparison to the published result for DARTS since the stage (2) evaluation was performed after training for only 100 epochs.

Next, as shown in the last row of Table 2, we perform an additional run of Random (5) to test the robustness of our result. For this second set of seeds, the minimum and average test error of the best architectures across 4 trials is 2.70 and 2.88 respectively, which differ non-trivially from that of the first run. However, after performing the stage (3) evaluation for the best architecture from the second set of seeds, Random (5) Run 2 reaches an average test error of 2.86 ± 0.09 with a minimum of 2.70 (see Table 3), which closely match the results shown for our first run of 2.71 and 2.85 ± 0.08 respectively.

Delving into the intermediate results, we compare the performance of the best architectures across trials from our independent run of DARTS, Random (5) Run 1, and Random (5) Run 2 after training each from scratch for 100 epochs and 600 epochs (see Table 3). We see that the ranking is unstable between 100 epochs and 600 epochs in two significant cases (i.e., reproduced DARTS and Random (5) Run 2), which motivated our strategy of training the final architectures across trials to 600 epochs in order to select the best architecture for final evaluation across 10 seeds. This suggests we should be cautious when using noisy signals for the performance of different architectures, especially since architecture search is conducted for DARTS and Random (5) for only 50 and 150 epochs respectively.

Table 3: **CIFAR-10 Benchmark: Ranking of Intermediate Test Error.** Architectures are retrained from scratch using the proxyless network and the error on the test set is reported after training for the indicated number of epochs. Rank is calculated per search method across the 4 trials. We also show the average over 10 seeds for the best architecture from the top trial for reference.
‡ These results were run before we fixed the non-determinism in DARTS code (see Appendix A.4).

		Epochs				**Across**	
Search		100		600		**10 Seeds**	
Method	**Trial**	Value	Rank	Value	Rank	Min	Avg
Reproduced	1	7.63	2	2.92	2		
Darts‡	2	7.67	3	2.77	1	2.62	2.78 ± 0.12
	3	8.38	4	3.00	3		
	4	7.51	1	3.05	4		
Random (5)	1	7.48	2	2.96	2		
Run 1	2	8.37	4	3.33	4		
	3	7.46	1	2.83	1	2.71	2.85 ± 0.08
	4	7.58	3	3.00	2		
Random (5)	1	7.37	2	2.93	3		
Run 2‡	2	7.65	3	3.01	4		
	3	8.06	4	2.70	1	2.70	2.86 ± 0.09
	4	7.16	2	2.85	2		

References

[1] Gabriel Bender, Pieter-Jan Kindermans, Barret Zoph, Vijay Vasudevan, and Quoc Le. Understanding and simplifying one-shot architecture search. In *International Conference on Machine Learning*, 2018.

[2] Y. Bengio. Gradient-based optimization of hyperparameters. In *Neural Computation*, 2000.

[3] J. Bergstra et al. Algorithms for hyper-parameter optimization. In *Advances in Neural Information Processing Systems*, 2011.

[4] James Bergstra and Yoshua Bengio. Random search for hyper-parameter optimization. *Journal of Machine Learning Research*, 13(Feb):281–305, 2012.

[5] Andrew Brock, Theo Lim, J.M. Ritchie, and Nick Weston. SMASH: One-shot model architecture search through hypernetworks. In *International Conference on Learning Representations*, 2018.

[6] Han Cai, Jiacheng Yang, Weinan Zhang, Song Han, and Yong Yu. Path-level network transformation for efficient architecture search. In *International Conference on Machine Learning*, 2018.

[7] Han Cai, Ligeng Zhu, and Song Han. ProxylessNAS: Direct neural architecture search on target task and hardware. In *International Conference on Learning Representations*, 2019.

[8] Shengcao Cao, Xiaofang Wang, and Kris M. Kitani. Learnable embedding space for efficient neural architecture compression. In *International Conference on Learning Representations*, 2019.

[9] Terrance Devries and Graham W. Taylor. Improved regularization of convolutional neural networks with cutout. 2017.

[10] T. Domhan, J. T. Springenberg, and F. Hutter. Speeding up automatic hyperparameter optimization of deep neural networks by extrapolation of learning curves. In *International Joint Conferences on Artificial Intelligence*, 2015.

[11] Thomas Elsken, Jan Hendrik Metzen, and Frank Hutter. Multi-objective Architecture Search for CNNs. 2018.

[12] Thomas Elsken, Jan Hendrik Metzen, and Frank Hutter. Neural Architecture Search: A Survey. 2018.

[13] Stefan Falkner, Aaron Klein, and Frank Hutter. Bohb: Robust and efficient hyperparameter optimization at scale. In *International Conference on Machine Learning*, 2018.

[14] Matthias Feurer, Aaron Klein, Katharina Eggensperger, Jost Springenberg, Manuel Blum, and Frank Hutter. Efficient and robust automated machine learning. In *Advances in Neural Information Processing Systems*, 2015.

[15] D. Golovin, B. Sonik, S. Moitra, G. Kochanski, J. Karro, and D.Sculley. Google vizier: A service for black-box optimization. In *SIGKDD Conference on Knowledge Discovery and Data Mining*, 2017.

[16] Google. Google repo for amoebanet. `https://github.com/tensorflow/tpu/tree/master/models/official/amoeba_net`, 2018.

[17] Google. Google automl. `https://cloud.google.com/automl/`, 2018.

[18] Google. Google repo for nasnet. `https://github.com/tensorflow/models/tree/master/research/slim/nets/nasnet`, 2018.

[19] Elad Hoffer, Itay Hubara, and Daniel Soudry. Train longer, generalize better: closing the generalization gap in large batch training of neural networks. In *Advances in Neural Information Processing Systems*, 2017.

[20] F. Hutter, H. Hoos, and K. Leyton-Brown. Sequential model-based optimization for general algorithm configuration. In *Proc. of LION-5*, 2011.

[21] M. Jaderberg, V. Dalibard, S. Osindero, W.M. Czarnecki, J. Donahue, A. Razavi, O. Vinyals, T. Green, I. Dunning, K. Simonyan, et al. Population based training of neural networks. 2017.

[22] K. Jamieson and A. Talwalkar. Non-stochastic best arm identification and hyperparameter optimization. In *International Conference on Artificial Intelligence and Statistics*, 2015.

[23] Haifeng Jin, Qingquan Song, and Xia Hu. Auto-Keras: Efficient Neural Architecture Search with Network Morphism. 2018.

[24] Kirthevasan Kandasamy, Gautam Dasarathy, Junier B Oliva, Jeff Schneider, and Barnabás Póczos. Gaussian process bandit optimisation with multi-fidelity evaluations. In *Advances in Neural Information Processing Systems*, 2016.

[25] Kirthevasan Kandasamy, Willie Neiswanger, Jeff Schneider, Barnabas Poczos, and Eric Xing. Neural Architecture Search with Bayesian Optimization and Optimal Transport. *Advances in Neural Information Processing Systems*, 2018.

[26] A. Klein, S. Falkner, S. Bartels, P. Hennig, and F. Hutter. Fast bayesian optimization of machine learning hyperparameters on large datasets. *International Conference on Artificial Intelligence and Statistics*, 2017.

[27] A. Krizhevsky. Learning multiple layers of features from tiny images. In *Technical report, Department of Computer Science, Univsersity of Toronto*, 2009.

[28] L. Li, K. Jamieson, G. DeSalvo, A. Rostamizadeh, and A. Talwalkar. Hyperband: Bandit-based configuration evaluation for hyperparameter optimization. *International Conference on Learning Representation*, 17, 2017.

[29] Liam Li, Kevin G. Jamieson, Afshin Rostamizadeh, Ekaterina Gonina, Moritz Hardt, Benjamin Recht, and Ameet Talwalkar. Massively parallel hyperparameter tuning. 2019.

[30] Chenxi Liu, Barret Zoph, Maxim Neumann, Jonathon Shlens, Wei Hua, Li-Jia Li, Li Fei-Fei, Alan Yuille, Jonathan Huang, and Kevin Murphy. Progressive Neural Architecture Search. *European Conference on Computer Vision*, 2018.

[31] Hanxiao Liu, Karen Simonyan, Oriol Vinyals, Chrisantha Fernando, and Koray Kavukcuoglu. Hierarchical representations for efficient architecture search. In *International Conference on Learning Representations*, 2018.

[32] Hanxiao Liu, Karen Simonyan, and Yiming Yang. DARTS: differentiable architecture search. *arXiv:1806.09055*, 2018.

[33] Hanxiao Liu, Karen Simonyan, and Yiming Yang. DARTS: Differentiable architecture search. In *International Conference on Learning Representations*, 2019.

[34] Renqian Luo, Fei Tian, Tao Qin, Enhong Chen, and Tie-Yan Liu. Neural Architecture Optimization. *Advances In Neural Information Processing Systems*, 2018.

[35] Dougal Maclaurin, David Duvenaud, and Ryan Adams. Gradient-based hyperparameter optimization through reversible learning. In *International Conference on Machine Learning*, 2015.

[36] M. Marcus, M. Marcinkiewicz, and B. Santorini. Building a large annotated corpus of english: The penn treebank. *Computational Linguistics*, 19(2):313–330, 1993.

[37] S. Merity, N.S. Keskar, and R. Socher. Regularizing and optimizing LSTM language models. In *International Conference on Learning Representations*, 2018.

[38] Renato Negrinho and Geoff Gordon. DeepArchitect: Automatically Designing and Training Deep Architectures. 2017.

[39] Randal S Olson and Jason H Moore. Tpot: A tree-based pipeline optimization tool for automating machine learning. In *Workshop on Automatic Machine Learning*, 2016.

[40] H. Pham, M. Guan, B. Zoph, Q. Le, and J. Dean. Efficient neural architecture search via parameters sharing. In *International Conference on Machine Learning*, 2018.

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

[42] Esteban Real, Alok Aggarwal, Yanping Huang, and Quoc V Le. Regularized Evolution for Image Classifier Architecture Search. 2018.

[43] Jasper Snoek, Hugo Larochelle, and Ryan P Adams. Practical bayesian optimization of machine learning algorithms. In *Advances in Neural Information Processing Systems*, 2012.

[44] K. Swersky, J. Snoek, and R. Adams. Multi-task bayesian optimization. In *Advances in Neural Information Processing Systems*, 2013.

[45] Tao Wei, Changhu Wang, Yong Rui, and Chang Wen Chen. Network morphism. In *International Conference on Machine Learning*, 2016.

[46] Sirui Xie, Hehui Zheng, Chunxiao Liu, and Liang Lin. SNAS: stochastic neural architecture search. In *International Conference on Learning Representations*, 2019.

[47] Yoshihiro Yamada, Masakazu Iwamura, and Koichi Kise. Shakedrop regularization. 2018.

[48] Zhilin Yang, Zihang Dai, Ruslan Salakhutdinov, and William W. Cohen. Breaking the softmax bottleneck: A high-rank RNN language model. In *International Conference on Learning Representations*, 2018.

[49] Chris Zhang, Mengye Ren, and Raquel Urtasun. Graph hypernetworks for neural architecture search. In *International Conference on Learning Representations*, 2019.

[50] Barret Zoph and Quoc V Le. Neural Architecture Search with Reinforcement Learning. *International Conference on Learning Representation*, 2017.

[51] Barret Zoph, Vijay Vasudevan, Jonathon Shlens, and Quoc V. Le. Learning transferable architectures for scalable image recognition. In *Conference on Computer Vision and Pattern Recognition*, 2018.

Appendix A. Appendix

A.1 Background

We first provide an overview of the components of hyperparameter optimization and, by association, NAS. As shown in Figure 1, a general hyperparameter optimization problem has three components, each of which can have NAS-specific approaches. We provide a brief overview of the components below, drawing attention to NAS-specific methods (see the survey by Elsken et al. [12] for a more thorough coverage of NAS).

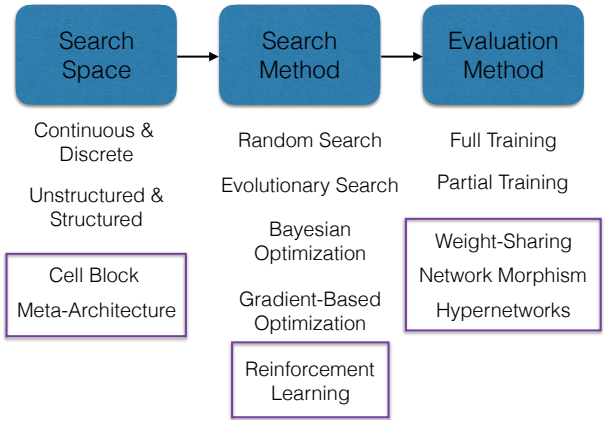


Figure 1: **Components of hyperparameter optimization.** Primarily NAS-specific methods are lined in purple.

Search Space. Hyperparameter optimization involves identifying a good hyperparameter configuration from a set of possible configurations. The search space defines this set of configurations, and can include continuous or discrete hyperparameters in a structured or unstructured fashion [43; 4; 14; 39]. NAS-specific search spaces usually involve discrete hyperparameters with additional structure that can be captured with a directed acyclic graph (DAG) [40; 33]. Additionally, since a search space for designing an entire architecture would have too many nodes and edges, search spaces are usually defined over some smaller building block, i.e., cell blocks, that are repeated in some way via a preset or learned meta-architecture to form a larger architecture [12]. We design our random search NAS algorithm for such a cell block search space, using the same search spaces for the CIFAR-10 and PTB benchmarks as DARTS for our experiments. We provide a concrete example of one such search space in Appendix A.3.

Search Method. Given a search space, there are various search methods to select putative configurations to evaluate. Random search is the most basic approach, yet it is quite effective in practice [4; 28]. Various general and NAS-specific adaptive methods have also been introduced, all of which attempt to bias the search in some way towards configurations that are more likely to perform well. In traditional hyperparameter optimization, the choice of search method can depends on the search space. Bayesian approaches based on Gaussian processes [43; 26; 44; 24] and gradient-based approaches [2; 35] are generally only applicable to continuous search spaces. In contrast, tree-based Bayesian [20; 3], evolutionary strategies [39], and random search are more flexible and can be applied to any search space. NAS-specific search methods can also be categorized into the same broad categories but are tailored for structured NAS search spaces (we provide a more involved discussion in Appendix A.2).

Evaluation Method. For each hyperparameter configuration considered by a search method, we must evaluate its quality. The default approach to perform such an evaluation involves fully training a model with the given hyperparameters, and subsequently measuring its quality, e.g., its predictive accuracy on a validation set. The first generation of NAS methods relied on full training evaluation, and thus required

thousands of GPU days to achieve a desired result [50; 41; 51; 42]. In contrast, partial training methods exploit early-stopping to speed up the evaluation process at the cost of noisy estimates of configuration quality. These methods use Bayesian optimization [26; 24; 13], performance prediction [15; 10], or multi-armed bandits [22; 28; 29] to adaptively allocate resources to different configurations. NAS-specific evaluation methods exploit the structure of neural networks to provide even cheaper, heuristic estimates of quality. Many of these methods center around sharing and reuse: network morphisms build upon previously trained architectures [6; 11; 23]; hypernetworks and performance prediction encode information from previously seen architectures [5; 30; 49]; and weight-sharing methods [40; 33; 1; 46; 7] use a single set of weights for all possible architectures.

A.2 Related Work

We now provide additional context for the three issues we identified with the current state of NAS research in Section 1.

A.2.1 INADEQUATE BASELINES

Existing works in NAS do not provide adequate comparison to random search and other hyperparameter optimization methods. Some works either compare to random search given a budget of just of few evaluations [40; 33] or Bayesian optimization methods without efficient architecture evaluation schemes [23]. While Real et al. [42] and Cai et al. [6] provide a thorough comparison to random search, they use random search with full training even though partial training methods have been shown to be orders-of-magnitude faster than standard random search [28; 29].

Although certain hyperparameter optimization methods [43; 35; 26] require non-trivial modification in order to work with NAS search spaces, others are easily applicable to NAS problems [20; 3; 10; 13; 28; 29]. Of these applicable methods, we choose to use a simple method combining random search with early-stopping called ASHA [29] to provide a competitive baseline for standard hyperparameter optimization. Li et al. [29] showed ASHA to be a state-of-the-art, theoretically principled, bandit-based partial training method that outperforms leading adaptive search strategies for hyperparameter optimization. We compare the empirical performance of ASHA with that of NAS methods in Section 2.

A.2.2 COMPLEX METHODS

Much of the complexity of NAS methods is introduced in the process of adapting search methods for NAS-specific search spaces, which usually involve discrete hyperparameters with a DAG representation where each node represents local computations and edges of the DAG represent the flow of data from one node to another [40; 33]. Evolutionary approaches need to define a set of possible mutations to apply to different architectures [41; 42]; Bayesian optimization approaches [23; 25] rely on specially designed kernels; gradient-based methods transform the discrete architecture search problem into a continuous optimization problem [34; 33; 46; 7]; and Zoph and Le [50], Zoph et al. [51], and Pham et al. [40] use reinforcement learning to train a recurrent neural network controller to generate good architectures. All of these search approaches add a significant amount of complexity with no clear winner, especially since methods some times use different search spaces and evaluation methods. To simplify the search process and help isolate important components of NAS, we use random search to sample architectures from the search space.

Additional complexity is also introduced by NAS-specific evaluation methods—like network morphisms; hypernetworks and performance prediction; and weight-sharing— that exploit the structure of NAS search spaces to speed up the evaluation of the quality of different architectures. Network morphisms require architecture transformations that satisfy certain criteria; hypernetworks and performance prediction methods encode information from previously seen architectures in an auxiliary network; and weight-sharing methods

[40; 33; 1; 46; 7] use a single set of weights for all possible architectures and hence, can require careful training routines.

Despite their complexity, these more efficient NAS evaluation methods are 1–3 orders-of-magnitude cheaper than full training (see Table 1 and Table 5), at the expense of decreased fidelity to the true performance. Of these evaluation methods, network morphism still requires on the order of 100 GPU days [30; 11] and, while hypernetworks and prediction performance based methods can be cheaper, weight-sharing is less complex since it does not require training an auxiliary network. In addition to the computational efficiency of weight-sharing methods [33; 40; 7; 46], which only require computation on the order of fully training a single architecture, this approach has also achieved the best result on the two standard benchmarks [33; 7]. Hence, we use random search with weight-sharing as our starting point for a simple and efficient NAS method.

Our work is inspired by the result of Bender et al. [1], which showed that random search, combined with a well-trained set of shared weights can successfully differentiate good architectures from poor performing ones. However, their work required several modifications to stabilize training (e.g., a tunable path dropout schedule over edges of the search DAG and a specialized ghost batch normalization scheme [19]). Furthermore, they only report experimental results on the CIFAR-10 benchmark, on which they fell slightly short of the results for leading NAS methods. In contrast, our combination of random search with weight-sharing greatly simplifies the training routine and we identify key variables needed to achieve competitive results on both CIFAR-10 and PTB benchmarks.

A.2.3 LACK OF REPRODUCIBILITY

The earliest NAS results lacked exact and broad reproducibility due to the tremendous amount of computation required to achieve the results [50; 51; 42]. Additionally, some of these methods used specialized hardware (i.e., TPUs) that were not easily accessible to researchers at the time [42]. Although the final architectures were eventually provided [18; 16], the code for the search methods used to produce these results has not been released, precluding researchers from reproducing these results even if they had sufficient computational resources.

Table 4: **Reproducibility of NAS Publications.** Summary of the reproducibility status of recent NAS publications appearing in top machine learning conferences. For the hyperparameter tuning column, N/A indicates we are not aware that the authors performed additional hyperparameter optimization.
† Published result is not reproducible for the PTB benchmark when training the reported final architecture with provided code.
* Code to reproduce experiments was requested on OpenReview.

Conference	Publication	Architecture Search Code	Model Evaluation Code	Random Seeds	Hyperparameter Tuning
ICLR 2018	Brock et al. [5]	Yes	Yes	No	N/A
	Liu et al. [31]	No	No		
ICML 2018	Pham et al. [40]†	Yes	Yes	No	Undocumented
	Cai et al. [6]	Yes	Yes	No	N/A
	Bender et al. [1]	No	No		
NIPS 2018	Kandasamy et al. [25]	Yes	Yes	No	N/A
	Luo et al. [34]	Yes	Yes	No	Grid Search
ICLR 2019	Liu et al. [33]	Yes	Yes	No	Undocumented
	Cai et al. [7]	No	Yes	No	N/A
	Zhang et al. [49]*	No	No		
	Xie et al. [46]*	No	No		
	Cao et al. [8]	No	No		

Recently, it has become feasible to evaluate the exact and broad reproducibility of many SOTA methods due to their reduced computational cost. However, while many authors have released code for their work [e.g., 40; 33; 5; 6], others have not made their code publicly available [e.g., 46; 49], including the work most closely related to ours by Bender et al. [1]. We summarize the reproducibility of recent NAS publications at some of the major machine learning conferences in Table 4 according to the availability of the following:

1. **Architecture search code.** The output of this code is the final architecture that should be trained on the evaluation task.

2. **Model evaluation code.** The output of this code is the final performance on the evaluation task.

3. **Hyperparameter tuning documentation.** This includes code used to perform hyperparameter tuning of the final architectures, if any.

4. **Random Seeds.** This includes random seeds used for both the search and post-processing (i.e., retraining of final architecture as well as any additional hyperparameter tuning) phases. Most works provide the final architectures but random seeds are required to verify that the search process actually results in those final architectures and the performance of the final architectures matches the published result. Note the random seeds are only useful if the code for search and post-processing phases are deterministic up to a random seed; this was not the case for the DARTS code used for the CIFAR-10 benchmark.

All 4 criteria are necessary for exact reproducibility. Due to the absence of random seeds for all methods with released code, none of the methods in Table 4 are exactly reproducible from the search phase to the final architecture evaluation phase.

While only criteria 1–3 are necessary to estimate broad reproducibility, there is minimal discussion of the broad reproducibility of existing methods in published work. With the exception of NASBOT [25] and DARTS [33], the methods in Table 4 only report the performance of the best found architecture, presumably resulting from a single run of the search process. Although this is understandable in light of the computational costs for some of these methods [34; 6], the high variance of extremal statistics makes it difficult to isolate the impact of the novel contributions introduced in each work. DARTS is particularly commendable in acknowledging its dependence on random initialization, prompting the use multiple runs to select the best architecture. In our experiments in Section 2, we follow DARTS and report the result of our random weight-sharing method across multiple trials; in fact, we go one step further and evaluate the broad reproducibility of our results with another set of random seeds.

A.3 Random Search with Weight-Sharing

We now introduce our NAS algorithm that combines random search with weight-sharing. Our algorithm is designed for an arbitrary search space with a DAG representation, and in our experiments in Section 2, we use the same search spaces as that considered by DARTS [33] for the standard PTB and CIFAR-10 NAS benchmarks.

For concreteness, consider the search space used by DARTS for designing a recurrent cell for the PTB benchmark: the DAG considered for the recurrent cell has $N = 8$ nodes and the operations considered include tanh, relu, sigmoid, and identity. Figure 2 shows an example of an architecture from this search space. To sample an architecture from this search space, we apply random search in the following manner:

1. For each node in the DAG, determine what decisions must be made. In the case of the PTB search space, we need to choose a node as input and a corresponding operation to apply to generate the output of the node.

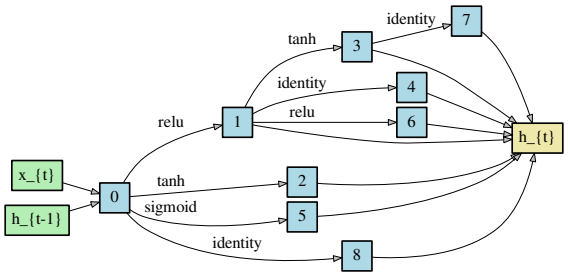


Figure 2: **Recurrent Cell on PTB Benchmark.** The best architecture found by random search with weight-sharing in Section A.5 is depicted. Each numbered square is a node of the DAG and each edge represents the flow of data from one node to another after applying the indicated operation along the edge. Nodes with multiple incoming edges (i.e., node 0 and output node `h_{t}` concatenate the inputs to form the output of the node).

2. For each decision, identify the possible choices for the given node. In the case of the PTB search space, if we number the nodes from 1 to N, node i can take the outputs of nodes 0 to node $i-1$ as input (the initial input to the cell is index 0 and is also a possible input). Additionally, we can choose an operation from $\{$tanh, relu, sigmoid, and identity$\}$ to apply to the output of node i.

3. Finally, moving from node to node, we sample uniformly from the set of possible choices for each decision that needs to be made.

In order to combine random search with weight-sharing, we simply use randomly sampled architectures to train the shared weights. In the case of the PTB benchmark, the same weights are applied to all possible inputs to a node. Shared weights are updated by selecting a single architecture for a given minibatch and updating the shared weights by back-propagating through the network with only the edges and operations as indicated by the architecture activated. Hence, the number of architectures used to update the shared weights is equivalent to the total number of minibatch training iterations.

After training the shared weights for a certain number of epochs, we use these trained shared weights to evaluate the performance of a number of randomly sampled architectures on a separate held out dataset. We select the best performing one as the final architecture, i.e., as the output of our search algorithm.

A.3.1 RELEVANT META-HYPERPARAMETERS

There are a few key meta-hyperparameters that impact the behavior of our search algorithm. We describe each of them below, along with a description of how we expect them to impact the search algorithm, both in terms of search quality and computational costs.

Training epochs. Increasing the number of training epochs while keeping all other parameters the same increases the total number of minibatch updates and hence, the number of architectures used to update the shared weights. Intuitively, training with more architectures should help the shared weights generalize better to what are likely unseen architectures in the evaluation step. Unsurprisingly, more epochs increase the computational time required for architecture search.

Batch size. Decreasing the batch size while keeping all other parameters the same also increases the number of minibatch updates but at the cost of noisier gradient update. Hence, we expect reducing the batch size to have a similar effect as increasing the number of training epochs but may necessitate adjusting other meta-hyperparameters to account for the noisier gradient update. Intuitively, more minibatch updates increase the computational time required for architecture search.

Network size. Increasing the search network size increases the dimension of the shared weights. Intuitively, this should boost performance since a larger search network can store more information about different architectures. Unsurprisingly, larger networks require more GPU memory.

Number of evaluated architectures. Increasing the number of architectures that we evaluate using the shared weights allows for more exploration in the architecture search space. Intuitively, this should help assuming that there is a high correlation between the performance of an architecture evaluated using shared weights and the ground truth performance of that architecture when trained from scratch [1]. Unsurprisingly, evaluating more architectures increases the computational time required for architecture search.

Other learning meta-hyperparameters will likely need to be adjusted accordingly for different settings of the key relevant meta-hyperparameters listed above. In our experiments in Section 2, we tune *gradient clipping* as a fifth meta-hyperparameter, though there are other possible meta-hyperparameters that may benefit from additional tuning (e.g., learning rate, momentum).

In Section 2, following these intuitions, we incrementally explore the design space of our search method in order to improve search quality and make full use of the available GPU memory and computational resources.

A.4 CIFAR-10 Benchmark

In this section, we provide additional detail for the experiments in Section 2.1.

Following DARTS, the DAG considered for the convolutional cell has $N = 4$ search nodes and the operations considered include 3×3 and 5×5 separable convolutions, 3×3 and 5×5 dilated separable convolutions, 3×3 max pooling, and 3×3 average pooling, and zero [33]. To sample an architecture from this search space, we have to choose, for each node, 2 input nodes from previous nodes and associated operations to perform on each input (there are two initial inputs to the cell that are also possible choices); we sample in this fashion twice, once for the normal convolution cell and one for the reduction cell (e.g., see Figure 3).

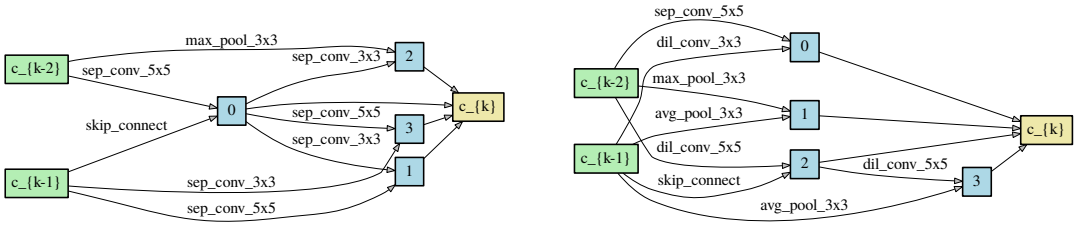
Due to higher memory requirements for weight-sharing, Liu et al. [33] uses a smaller proxy network with 8 stacked cells and 16 initial channels to perform the convolutional cell search, followed by a larger proxyless network with 20 stacked cells and 36 initial channels to perform the evaluation. We use the same proxy and proxyless networks for the relevant stages of the experiment. We discuss how our setup differs from that of DARTS below.

Architecture Operations. In stage (1), DARTS trained the shared weights network with the zero operation included in the list of considered operations but removed the zero operation when selecting the final architecture to evaluate in stages (2) and (3). For our random search with weight-sharing, we decided to *include* the zero operation for both search and evaluation. We hypothesize that our results may improve if we impose a higher complexity on the final architectures by excluding the zero operation.

Stage 1 Procedure. For random search with weight-sharing, after the shared weights are fully trained, we evaluate randomly sampled architectures using the shared weights and select the best one for stage (2) evaluation. Due to the higher cost of evaluating on the full validation set, we evaluate each architecture using 10 minibatches instead. We split the total number of architectures to be evaluated into sets of 1000. For each 1000, we select the best 10 according the cheap evaluation on part of the validation set and evaluate on the full validation set. Then we select the top architecture across all sets of 1000 for stage (2) evaluation.

Reproducibility. The code released by Liu et al. [33] did not produce deterministic results for the CNN benchmark due to non-determinism in CuDNN and in data loading. We removed the non-deterministic behavior in CuDNN by setting

```
cudnn.benchmark = False
cudnn.deterministic = True
cudnn.enabled=True
```



(a) Normal Cell (b) Reduction Cell

Figure 3: **Convolutional Cells on CIFAR-10 Benchmark:** Best architecture found by random search with weight-sharing.

Note that this only disables the non-deterministic functions in CuDNN and does not adversely affect training time as much as turning off CuDNN completely. We fix additional non-determinism from data loading by setting the seed for the `random` package in addition to `numpy.random` and `pytorch` seed and turning off multiple threads for data loading.

We ran ASHA and one set of trials for random search with weight-sharing using the non-deterministic code before fixing the seeding to get deterministic results. Hence, the result for ASHA does not satisfy exact reproduciblity due to non-deterministic training and asynchronous updates. Due to the demanding computational cost of these experiments, we use the non-deterministic runs of random with weight-sharing as the second set of trials for Random (5) in Table 2, all other settings for random search with weight-sharing are deterministic.

The best architecture found by Random (5) Run 1 for the normal and reduction cells is shown in Figure 3.

A.5 PTB Benchmark

We now present results for the PTB benchmark. We use the DARTS search space for the recurrent cell, which is described in Appendix A.3. For this benchmark, due to higher memory requirements for their mixture operation, DARTS used a small recurrent network with embedding and hidden dimension of 300 to perform the architecture search followed by a larger network with embedding and hidden dimension of 850 to perform the evaluation. For the PTB benchmark, we refer to the network used in the first stage as the *proxy* network and the network in the later stages as the *proxyless* network.

Our setup matches that of DARTS with the following exceptions:

Architecture Operations. In stage (1), DARTS trained the shared weights network with the zero operation included in the list of considered operations but removed the zero operation when selecting the final architecture to evaluate in stages (2) and (3). For our random search with weight-sharing, we decided to exclude the zero operation for both search and evaluation.

Stage 3 Procedure. For stage (3) evaluation, we follow the ArXiv version of DARTS [32], which reported two sets of results, one after training for 1600 epochs and another fine tuned result after training for an additional 1000 epochs. In the ICLR version, Liu et al. [33] simply say they trained the final network to convergence. We trained for another 1000 epochs for a total of 3600 epochs to approximate training to convergence.

We next present the final search results. We subsequently explore the impact of various meta-hyperparameters on random search with weight-sharing, and finally evaluate the reproducibility of various methods on this benchmark.

A.5.1 FINAL SEARCH RESULTS

We now present our final evaluation results in Table 5. Specifically, we report the output of stage (3), in which we train the proxyless network configured according to the best architectures found by different methods for 3600 epochs. We discuss various aspects of these results in the context of the three issues—baselines, complex methods, reproducibility—introduced in Section 1.

Table 5: **PTB Benchmark: Comparison with state-of-the-art NAS methods and manually designed networks.** Lower test perplexity is better on this benchmark. The results are grouped by those for manually designed networks, published NAS methods, and the methods that we evaluated. Table entries denoted by "-" indicate that the field does not apply, while entries denoted by "N/A" indicate unknown entries. The search cost, unless otherwise noted, is measured in GPU days. Note that the search cost is hardware dependent and the search cost shown for our results are calculated for Tesla P100 GPUs; all other numbers are those reported by Liu et al. [33].
[#] Search cost is in CPU-days.
[*] We could not reproduce this result using the code released by the authors at `https://github.com/melodyguan/enas`.
[†] The stage (1) cost shown is that for 1 trial as opposed to the cost for 4 trials shown for DARTS and Random search WS. It is unclear whether ENAS requires multiple trials followed by stage (2) evaluation in order to find a good architecture.

		Test Perplexity		**Params**	**Search Cost**			**Comparable**	**Search**
Architecture	**Source**	Valid	Test	(M)	Stage 1	Stage 2	Total	**Search Space?**	**Method**
LSTM + DropConnect	[37]	60.0	57.3	24	-	-	-	-	manual
ASHA + LSTM + DropConnect	[29]	58.1	56.3	24	-	-	13	N	HP-tuned
LSTM + MoS	[48]	56.5	54.4	22	-	-	-	-	manual
NAS[#]	[50]	N/A	64.0	25	-	-	1e4	N	RL
ENAS[*†]	[40]	N/A	56.3	24	0.5	N/A	N/A	Y	RL
ENAS[†]	[33]	60.8	58.6	24	0.5	N/A	N/A	Y	random
Random search baseline	[33]	61.8	59.4	23	-	-	2	Y	random
DARTS (first order)	[33]	60.2	57.6	23	0.5	1	1.5	Y	gradient-based
DARTS (second order)	[33]	58.1	55.7	23	1	1	2	Y	gradient-based
DARTS (second order)	Ours	58.2	55.9	23	1	1	2	Y	gradient-based
ASHA baseline	Ours	58.6	56.4	23	-	-	2	Y	random
Random search WS	Ours	57.8	55.5	23	0.25	1	1.25	Y	random

First, we evaluate the ASHA baseline using 2 GPU days, which is equivalent to the total cost of DARTS (second order). In contrast to the one random architecture evaluated by Pham et al. [40] and the 8 evaluated by Liu et al. [33] for their random search baselines, ASHA evaluated over 300 architectures with the allotted computation time. The best architecture found by ASHA achieves a test perplexity of 56.4, which is comparable to the published result for ENAS and significantly better than the random search baseline provided by Liu et al. [33], DARTS (first order), and the reproduced result for ENAS [33]. Our result demonstrates that the gap between SOTA NAS methods and standard hyperparameter optimization approaches on the PTB benchmark is significantly smaller than that suggested by the existing comparisons to random search [40; 33].

Next, we evaluate random search with weight-sharing with tuned meta-hyperparameters (see Appendix A.5.2 for details). With slightly lower search cost than DARTS, this method finds an architecture that reaches test perplexity 55.5, achieving SOTA perplexity compared to previous NAS approaches. We note that manually designed architectures are competitive with RNN cells designed by NAS methods on this benchmark. In fact, the work by Yang et al. [48] using LSTM with mixture of experts in the softmax layer (MoS) outperforms automatically designed cells. Our architecture would likely also improve significantly with MoS, but we train without MoS to provide a fair comparison to ENAS and DARTS.

Finally, we examine the reproducibility of the NAS methods with available code for both architecture search and evaluation. For DARTS, exact reproducibility was not feasible since Liu et al. [33] do not provide random seeds for the search process; however, we were able to reproduce the performance of their reported best architecture. We also evaluated the broad reproducibility of DARTS through an independent run, which reached a test perplexity of 55.9, compared to the published value of 55.7. For ENAS, end-to-end exact reproducibility was infeasible due to non-deterministic code and missing random seeds for both the search and evaluation steps. Additionally, when we tried to reproduce their result using the provided final architecture, we could not match the reported test perplexity of 56.3 in our rerun. Consequently, in Table 5 we show the test perplexity for the final architecture found by ENAS trained using the DARTS code base, which Liu et al. [33] observed to give a better test perplexity than using the architecture evaluation code provided by ENAS. We next considered the reproducibility of random search with weight-sharing. We verified the exact reproducibility of our reported results, and then investigated their broad reproducibility by running another experiment with different random seeds. In this second experiment, we observed a final text perplexity of 56.5, compared with a final test perplexity of 55.5 in the first experiment. Our detailed investigation in Appendix A.5.3 shows that the discrepancies across both DARTS and random search with weight-sharing are unsurprising in light of the differing convergence rates among architectures on this benchmark.

A.5.2 IMPACT OF META-HYPERPARAMETERS

We now detail the meta-hyperparameter settings that we tried for random search with weight-sharing in order to achieve SOTA performance on the PTB benchmark. Similar to DARTS, in these preliminary experiments we performed 4 separate trials of each version of random search with weight-sharing, where each trial consists of executing stage (1) followed by stage (2). In stage (1), we train the shared weights and then use them to evaluate 2000 randomly sampled architectures. In stage (2), we select the best architecture out of 2000, according to the shared weights, to train from scratch using the proxyless network for 300 epochs.

We incrementally tune random search with weight-sharing by adjusting the following meta-hyperparameters associated with training the shared weights in stage (1): (1) gradient clipping, (2) batch size, and (3) network size. The settings we consider proceed as follows:

Random (1): We train the shared weights of the proxy network using the same setup as DARTS with the same values for number of epochs, batch size, and gradient clipping; all other meta-hyperparameters are the same.

Random (2): We decrease the maximum gradient norm to account for discrete architectures, as opposed to the weighted combination used by DARTS, so that gradient updates are not as large in each direction.

Random (3): We decrease batch size from 256 to 64 in order to increase the number of architectures used to train the shared weights.

Random (4): We train the larger proxyless network architecture with shared weights instead of the proxy network, thereby significantly increasing the number of parameters in the model.

The stage (2) performance of the final architecture after retraining from scratch for each of these settings is shown in Table 6. With the extra capacity in the larger network used in Random (4), random search with weight-sharing achieves average validation perplexity of 64.7 across 4 trials, with the best architecture (shown in Figure 2 in Appendix A.3) reaching 63.8. In light of these stage (2) results, we focused in stage (3) on the best architecture found by Random (4) Run 1, and achieved test perplexity of 55.5 after training for 3600 epochs as reported in Table 5.

A.5.3 INVESTIGATING REPRODUCIBILITY

We next examine the stage (2) intermediate results in Table 6 in the context of reproducibility. The first two rows of Table 6 show a comparison of the published stage (2) results for DARTS and our independent runs of DARTS. Both the best and average across 4 trials are worse in our reproduction of their results. Additionally,

Table 6: **PTB Benchmark: Comparison of Stage (2) Intermediate Search Results for Weight-Sharing Methods.** In stage (1), random search is run with different settings to train the shared weights. The resulting networks are used to evaluate 2000 randomly sampled architectures. In stage (2), the best of these architectures for each trial is then trained from scratch for 300 epochs. We report the performance of the best architecture after stage (2) across 4 trials for each search method.

	Setting				**Trial**					
Method	Network Config	Epochs	Batch Size	Gradient Clipping	1	2	3	4	Best	Average
DARTS [33]	proxy	50	256	0.25	67.3	66.3	63.4	63.4	**63.4**	**65.1**
Reproduced DARTS	proxy	50	256	0.25	64.5	67.7	64.0	67.7	64.0	66.0
Random (1)	proxy	50	256	0.25	65.6	66.3	66.0	65.6	65.6	65.9
Random (2)	proxy	50	256	0.1	65.8	67.7	65.3	64.9	64.9	65.9
Random (3)	proxy	50	64	0.1	66.1	65.0	64.9	64.5	64.5	65.1
Random (4) Run 1	proxyless	50	64	0.1	66.3	64.6	64.1	63.8	**63.8**	**64.7**
Random (4) Run 2	proxyless	50	64	0.1	63.9	64.8	66.3	66.7	63.9	65.4

Table 7: **PTB Benchmark: Ranking of Intermediate Validation Perplexity.** Architectures are retrained from scratch using the proxyless network and the validation perplexity is reported after training for the indicated number of epochs. The final test perplexity after training for 3600 epochs is also shown for reference.

	Validation Perplexity by Epoch										**Test**	
	300		500		1600		2600		3600		**Perplexity**	
Search Method	Value	Rank	Value	Rank	Value	Rank	Value	Rank	Value	Rank	Value	Rank
DARTS	64.0	4	61.9	2	59.5	2	58.5	2	58.2	2	55.9	2
ASHA	63.9	2	62.0	3	59.8	4	59.0	3	58.6	3	56.4	3
Random (4) Run 1	63.8	1	61.7	1	59.3	1	58.4	1	57.8	1	55.5	1
Random (4) Run 2	63.9	2	62.1	4	59.6	3	59.0	3	58.8	4	56.5	4

as previously mentioned, we perform an additional run of Random (4) with 4 different random seeds to test the broad reproducibility our result. The minimum stage (2) validation perplexity over these 4 trials is 63.9, compared to a minimum validation perplexity of 63.8 for the first set of seeds.

Next, in Table 7 we compare the validation perplexities of the best architectures from ASHA, Random (4) Run 1, Random (4) Run 2, and our independent run of DARTS after training each from scratch for up to 3600 epochs. The swap in relative ranking across epochs demonstrates the risk of using noisy signals for the reward. In this case, we see that even partial training for 300 epochs does not recover the correct ranking; training using shared weights further obscures the signal. The differing convergence rates explain the difference in final test perplexity of the best architecture from Random (4) Run 2 and those from DARTS and Random (4) Run 1, despite Random (4) Run 2 reaching a comparable perplexity after 300 epochs.

Overall, the results of Tables 6 and 7 demonstrate a high variance in the stage (2) intermediate results across trials, along with issues related to differing convergence rates for different architectures. These two issues help explain the differences between the independent runs of DARTS and random search with weight-sharing. A third potential source of variation, which could in particular adversely impact our random search with weight-sharing results, stems from the fact that we did not perform any additional hyperparameter tuning in stage (3); instead we used the same training hyperparameters that were tuned by Liu et al. [33] for the final architecture found by DARTS.

A.6 Available Code

Unless otherwise noted, our results are exactly reproducible from architecture search to final evaluation using the code available at `https://github.com/liamcli/randomNAS_release`. The code we use for random search with weight-sharing on both benchmarks is deterministic conditioned on a fixed random seed. We provide the final architectures used for each of the trials shown in the tables above, as well as the random seeds used to find those architectures. In addition, we perform no additional hyperparameter tuning for final architectures and only tune the meta-hyperparameters according to the discussion in the text itself. We also provide code, final architectures, and random seeds used for our experiments using ASHA. However, we note that there is one uncontrolled source of randomness in our ASHA experiments—in the distributed setting, the asynchronous nature of the algorithm means that the results depend on the order in which different architectures finish (partially) training. Lastly, our experiments were conducted using Tesla P100 and V100 GPUs on Google Cloud. We convert GPU time on V100 to equivalent time on P100 by applying a multiple of 1.5.