NOSBench-101: Towards Reproducible Neural Optimizer Search

Goktug Karakasli[1,2] **Steven Adriaensen**[2] **Frank Hutter**[2,3]

[1]IDEALworks GmbH
[2]University of Freiburg
[3]ELLIS Institute Tübingen

Abstract Recent advances in neural network architecture and hardware have revolutionized deep learning and made it a pervasive technology. Nonetheless, it is crucial to acknowledge that the achievement of training neural networks with millions and billions of parameters would not have been feasible without the advancement of effective optimization techniques. This has motivated the search for new efficient optimization algorithms that can improve the performance of deep learning networks even more. Despite the considerable manual (re)search effort, few of these methods have found their way into deep learning practice. Recently, various researchers have explored different search methods to learn/discover novel optimizers in an automated way, but the associated computational costs and lack of a standardized evaluation protocol have hindered progress in this field. Motivated by the success of Neural Architecture Search (NAS), which benefits from established and compute-efficient benchmarks like NASBench, we introduce a benchmark called NOSBench that can be used to test different Neural Optimizer Search (NOS) methods on the same tasks. We compare different NOS methods on a Prior-Data Fitted Networks (PFNs) meta-training task and show that the optimizer found transfer to other PFN training tasks (e.g., TabPFN, LC-PFN, PFNs4BO). Our experiments show that the NOSBench provides a useful way to compare and contrast different approaches in this field efficiently by caching and identifying identical optimizers, which we believe can help researchers identify promising search strategies as they search for new optimizers automatically, thereby bringing NOS into the mainstream.

1 Introduction

Deep Learning (1) has numerous applications such as speech recognition (2), natural language processing (3), computer vision (4), game playing (5), and protein structure prediction (6). Before being deployed to do these tasks, neural networks must be trained, often involves optimizing millions of parameters, for example, GPT-3 (3) has 175 billion parameters. Training methods vary, but backpropagation (7) and gradient descent are predominant. A key challenge with first-order optimization is setting the step size, which can lead to poor accuracy due to the non-convex and ill-conditioned nature of the optimization problem. To address this, numerous optimization techniques have been developed, including Stochastic Gradient Descent (SGD) (8), Momentum, RMSprop (9), Adagrad (10), Adam (11), and Hypergradient (12). Adagrad and Adam are particularly popular. However, many other optimization techniques proposed in the literature have not been used much (13). The best optimizer depends on the task, requiring parameter tuning for optimal performance. Designing task-specific optimizers is difficult and often necessitates deep knowledge of the task, which is hard to obtain. This challenge has led to the exploration of automating optimizer search, termed Neural Optimizer Search (NOS) (14). NOS aims to automatically discover and evaluate novel optimizers by exploring a search space to find those that perform well on various tasks with different objectives and training regimes. However, no standard search space exists, and recent methods lack open-source code, making performance comparisons difficult. Additionally, the high computational cost of evaluating optimizers hinders widespread adoption of NOS.

Summary of Contributions:. We introduce NOSBench, the first benchmark for evaluating neural optimizer search methods. We use it to compare different NOS approaches to discover a novel optimizer for training Prior-data Fitted Networks (PFNs)(15), evaluating their effectiveness in discovering task-specific optimizers. Additionally, we conducted generalization experiments on various PFN tasks and introduced an optimizer capable of efficiently training PFNs. We open-sourced NOSBench[1] to facilitate further research in this area.

2 Related Work

Traditional optimization algorithms, such as SGD and Adam, have been manually crafted by researchers and widely used in deep learning applications due to their effectiveness and simplicity. Previous benchmark studies (13; 16) provided insights into the relative performance of these methods under different conditions, but these compared hand-crafted optimizers, not NOS methods.

Learning to Optimize (L2O) (17) and Learning to Learn (L2L) (18) techniques aim to automatically derive optimization algorithms from data. While promising, the optimizers obtained often struggle to generalize beyond the training data (19). Also, L2L/L2O methods approach this learning problem sub-symbolically, i.e., the learned optimizers are typically represented as a deep neural network, making them difficult to interpret and trust.

We adopt the term Neural Optimizer Search (NOS) to refer to symbolic approaches that represent optimizers as computer programs, enabling the systematic exploration and discovery of novel optimization algorithms. The concept was first explored by Bengio et al. (20), who utilized genetic programming to discover new update rules for neural networks. Other notable works in this domain include "Neural Optimizer Search with Reinforcement Learning" (21) and "Symbolic Discovery of Optimization Algorithms" (22). The latter builds upon the foundations laid by AutoML-Zero (23) which evolves machine learning algorithms from scratch.

The term NASBench refers to a benchmark suite designed for evaluating Neural Architecture Search (NAS) methods. The first NASBench was NAS-Bench-101 (24). Notable variants include NAS-Bench-201 (25), NAS-Bench-Suite (26) and JAHS-Bench-201 (27), which provide standardized tasks and evaluation metrics for comparing different NAS methods.

3 NOSBench

3.1 Optimizer Search Space

While designing the search space, we followed a similar approach as in AutoMLZero (23) and LION (22), and formulated the discovery of optimizers as program search and programs are represented symbolically. The difference between our search space and (23; 22) is that hyperparameters are not directly part of our search space; rather, they can be jointly optimized using predefined constants. More details on the optimizer search space can be found in Appendix A

3.2 Caching Mechanism

While the NOSBench search space is vast, it also contains an abundance of equivalent and suboptimal optimizers. Recognizing equivalence or identifying invalid / diverging optimizers and efficiently caching these outcomes can significantly accelerate the search process. Within NOSBench, this acceleration is achieved by conducting pre-evaluation on a cheap proxy task, and using a hash of the optimization trajectory as a key in a persistent cache. More details on the caching mechanism can be found in Appendix B. In our experiments in Appendix D, we observed high hit rates, indicating the efficacy of our caching approach.

[1] `https://github.com/automl/NOSBench`

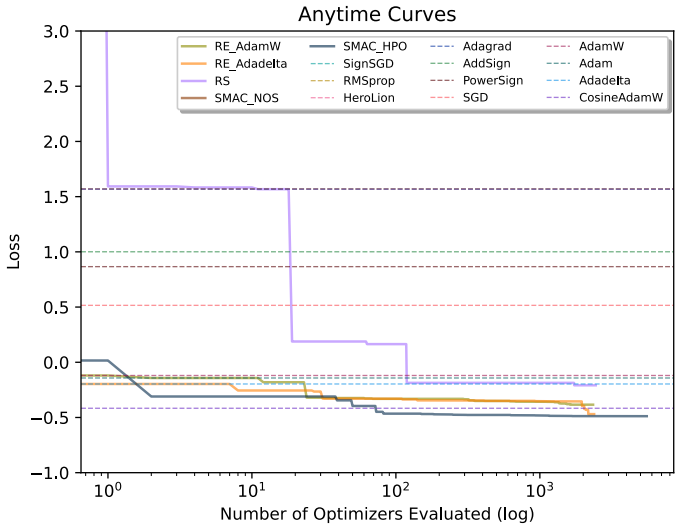

Figure 1: Optimizer baselines and anytime curves of search strategies

3.3 Meta-Training Task

In designing a benchmark for evaluating neural optimizer search methods, the selection of a suitable learning task is crucial. It must be hard enough to meaningfully distinguish the effectiveness of different optimizers. However, too hard tasks would impose significant resource demands in our experiments and to the end users of the benchmark. We opted for the task of training a small Prior-Data Fitted Network (PFN) for Bayesian linear regression. While the PFN model has over two million parameters, using a state-of-the-art optimizer and a modern GPU, training converges in just about a minute. Furthermore, PFNs utilize a decoder-only transformer architecture similar to the one used in many foundation models (e.g., LLMs) that currently represent the leading trend in modern deep learning. More details on our meta-training task can be found in Appendix C

4 Experiments

In this section, we present the experiments conducted to evaluate the proposed Neural Optimizer Search Benchmark (NOSBench, Section 3). Experimental details are given in Appendix F.

4.1 Benchmarking Optimizer Search

In this section, we compare different search algorithms within the context of NOSBench. Specifically, we examine the performance of Random Search (RS), Regularized Evolution (28), and SMAC (29) (Bayesian Optimization, SMAC_NOS), and compare the performance of optimizers discovered by these algorithms to that of popular optimizers such as Adam (11), AdamW (30), Adagrad (10), SGD (8), Adadelta (31), SignSGD (32), HeroLion (22), RMSprop (9), PowerSign and AddSign (21). For the optimizer baselines, we utilized the default parameters provided by PyTorch (33). Additionally, we conducted hyperparameter search (HPO) across all these optimizers using SMAC to identify one optimizer as a baseline (SMAC_HPO). The configuration space for this HPO run included categorical variables representing the choice of optimizer and conditionals for optimizer-specific hyperparameters. Each candidate optimizer is given 20 epochs to train the meta-training task.

Anytime curves of Random Search (RS), Regularized Evolution (RE) variants[2] starting from AdamW and Adadelta, and SMAC (Sequential Model-based Algorithm Configuration) on NOSBench

[2]Regularized Evolution variants refers to the Regularized Evolution algorithm, initialized with different populations.

are given in Figure 1. Even though RS outperformed some of the default optimizers, it struggled to identify a notable optimizer and failed to surpass the default PFN pipeline optimizer AdamW with Cosine Annealing. SMAC_NOS outperformed RS, but only marginally. The similarly of its trajectory to RS, suggests SMAC mainly explores the vast and sparse search space, and would require even longer runs to exploit its surrogate model. Both versions of RE performed comparably well, surpassing all default optimizers. Particularly, the version of RE starting from AdamW found an incumbent superior to the default PFN pipeline optimizer, which was chosen as the incumbent. More details on our incumbent can be found in Appendix E. SMAC_HPO was able to find an optimizer configuration performing even better than our incumbent in the PFN training task.

4.2 Evaluation of Incumbent

To evaluate the effectiveness of our incumbent optimizer, we conducted experiments across various PFN training tasks. These tasks included LC-PFN (34), PFNs4BO (35), and TabPFN (36). Our findings demonstrate that the incumbent optimizer not only performed well within the single meta-training task but also generalized effectively to different PFN tasks.

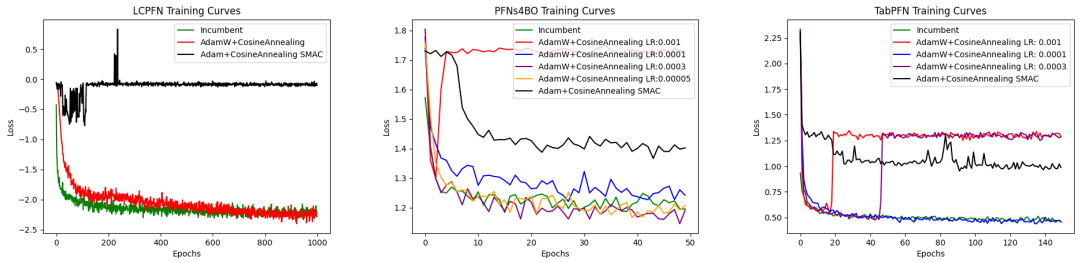

Figure 2: Learning Curves for LC-PFN (34) (**Left**), PFNs4BO (35) (**Middle**), TabPFN (36) (**Right**)

4.2.1 LC-PFN. In the LC-PFN (34) task, our incumbent optimizer showed a more aggressive training approach, resulting in a faster decrease in loss compared to the default training pipeline of LC-PFN (see Figure 2). However, both the incumbent and default pipeline converged to the same point at the end of the training, suggesting that while the incumbent optimizer may achieve quicker initial progress, it ultimately reaches a similar performance level as the default pipeline. The optimizer found by the SMAC_HPO baseline failed to generalize and performed poorly.

4.2.2 PFNs4BO. Similarly, in the PFNs4BO (35) task our incumbent performed well. We trained PFNs4BO with four different learning rates as provided in the original paper (35). Our incumbent performed better than learning rates of 0.001 and 0.0001, but learning rates of 0.0003 and 0.00005 outperformed our incumbent. Notably, the learning rate of 0.001 diverged, while learning rates of 0.0001, 0.0003, and 0.00005, along with our incumbent, resulted in similar performances.

4.2.3 TabPFN. In the TabPFN (36) task, initially, all models trained with three different learning rates—0.001, 0.0001, and 0.0003, as provided in the original paper—performed similarly, including our incumbent optimizer, as shown in Figure 2. By the end of the 150 epochs, our incumbent optimizer and the model with a 0.0001 learning rate performed similarly, with the latter achieving a marginally better loss during training.

5 Conclusion

Summary and envisioned impact. In this paper, we introduced NOSBench-101, the first benchmark for Neural Optimizer Search (NOS). Recent advances in deep learning have led to the development of neural networks with billions of parameters, making optimization techniques crucial for training

these models efficiently. This highlights the need for developing new optimizers. Developing new optimizers is hard, yet crucial for pushing the boundaries of deep learning performance. However, the process can be hard and time-consuming. NOSBench aims to address this challenge by providing a systematic and efficient approach to compare and contrast various NOS methods. By providing a standardized benchmark and tasks for NOS methods to automatically search, NOSBench facilitates the development and evaluation of optimization algorithms, ultimately advancing the state-of-the-art in neural network optimization. Our experiments demonstrate the effectiveness of NOSBench in evaluating different NOS methods on a single PFNs task and transfer of the discovered optimizers to other PFN tasks. By providing a unified framework for testing, NOSBench enables researchers to analyze the performance of optimization algorithms. This systematic evaluation facilitates the identification of promising search strategies for discovering new optimizers automatically.

Future work and limitations. For future work, expanding the benchmark to include more tasks would provide a more comprehensive evaluation of NOS methods across a broader range of applications. Furthermore, jointly searching across tasks to find an optimizer that can work effectively across diverse tasks would be particularly valuable, as it could lead to the development of more versatile optimization algorithms capable of handling a wide range of scenarios. Additionally, exploring joint optimization objectives, such as performance and memory consumption, could further improve our benchmark, enabling a more comprehensive evaluation of optimization algorithms across multiple criteria.

Our choice of the specific caching mechanism over alternatives such as a tabular or surrogate approach comes from several reasons. A tabular approach would be impractical due to the large and potentially continuous nature of the search space. On the other hand, a surrogate approach poses challenges in generalizing or interpolating between optimizers or programs, which remains an unclear task and could be a subject for future work. In contrast, our empirical approach, though susceptible to false positives, offers practical advantages such as lower overhead in calculating the key and a higher hit rate with a low false positive rate. Furthermore, more analytical approaches may exclude false positives and evaluating and implementing different hashing mechanisms could enhance the benchmark's accuracy.

Through a caching mechanism, NOSBench lowers the compute used in NOS, therefore decreasing the carbon emission and contributing to Green AutoML (37). As future work, hosting the benchmark on a server would make it accessible to a wider community of researchers and allow further carbon emission reduction due to the global persistent cache.

In conclusion, NOSBench offers a valuable tool for researchers to systematically evaluate and compare different NOS methods, ultimately advancing the state-of-the-art in neural network optimization. Looking at the broader, possibly negative impact of our work, automating the process of designing optimizers requires significant computational resources, negatively impacting the environment. To mitigate this impact we implemented a caching mechanism in our benchmark. Also note that we hope this line of work will produce more efficient optimizers, generally reducing the computational cost of deep learning, and allowing these design costs to be amortized over time.

Acknowledgements. Frank Hutter is a Hector Endowed Fellow at the ELLIS Institute Tübingen. All authors acknowledge funding by the German Research Foundation (DFG) through grant number 417962828, and the European Union (via ERC Consolidator Grant Deep Learning 2.0, grant no. 101045765), TAILOR, a project funded by EU Horizon 2020 research and innovation programme under GA No 952215. Views and opinions expressed are however those of the author(s) only and do not necessarily reflect those of the European Union or the European Research Council. Neither the European Union nor the granting authority can be held responsible for them.

References

[1] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016. `http://www.deeplearningbook.org`.

[2] A. Radford, J. W. Kim, T. Xu, G. Brockman, C. McLeavey, and I. Sutskever, "Robust speech recognition via large-scale weak supervision," 2022.

[3] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. M. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, and D. Amodei, "Language models are few-shot learners," 2020.

[4] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in Neural Information Processing Systems* (F. Pereira, C. Burges, L. Bottou, and K. Weinberger, eds.), vol. 25, Curran Associates, Inc., 2012.

[5] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. van den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, S. Dieleman, D. Grewe, J. Nham, N. Kalchbrenner, I. Sutskever, T. Lillicrap, M. Leach, K. Kavukcuoglu, T. Graepel, and D. Hassabis, "Mastering the game of Go with deep neural networks and tree search," *Nature*, vol. 529, pp. 484–489, Jan. 2016.

[6] J. Jumper, R. Evans, A. Pritzel, T. Green, M. Figurnov, O. Ronneberger, K. Tunyasuvunakool, R. Bates, A. Žídek, A. Potapenko, A. Bridgland, C. Meyer, S. A. A. Kohl, A. J. Ballard, A. Cowie, B. Romera-Paredes, S. Nikolov, R. Jain, J. Adler, T. Back, S. Petersen, D. Reiman, E. Clancy, M. Zielinski, M. Steinegger, M. Pacholska, T. Berghammer, S. Bodenstein, D. Silver, O. Vinyals, A. W. Senior, K. Kavukcuoglu, P. Kohli, and D. Hassabis, "Highly accurate protein structure prediction with alphafold," *Nature*, vol. 596, pp. 583–589, Aug 2021.

[7] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning representations by back-propagating errors," *Nature*, vol. 323, pp. 533–536, Oct 1986.

[8] H. Robbins and S. Monro, "A Stochastic Approximation Method," *The Annals of Mathematical Statistics*, vol. 22, no. 3, pp. 400 – 407, 1951.

[9] T. Tieleman, G. Hinton, *et al.*, "Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude," *COURSERA: Neural networks for machine learning*, vol. 4, no. 2, pp. 26–31, 2012.

[10] J. Duchi, E. Hazan, and Y. Singer, "Adaptive subgradient methods for online learning and stochastic optimization," *Journal of Machine Learning Research*, vol. 12, no. 61, pp. 2121–2159, 2011.

[11] D. Kingma and J. Ba, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014.

[12] A. G. Baydin, R. Cornish, D. M. Rubio, M. Schmidt, and F. Wood, "Online learning rate adaptation with hypergradient descent," 2018.

[13] R. M. Schmidt, F. Schneider, and P. Hennig, "Descending through a crowded valley - benchmarking deep learning optimizers," 2021.

[14] B. Zoph and Q. V. Le, "Neural architecture search with reinforcement learning," 2017.

[15] S. Müller, N. Hollmann, S. P. Arango, J. Grabocka, and F. Hutter, "Transformers can do bayesian inference," 2023.

[16] G. E. Dahl, F. Schneider, Z. Nado, N. Agarwal, C. S. Sastry, P. Hennig, S. Medapati, R. Eschenhagen, P. Kasimbeg, D. Suo, J. Bae, J. Gilmer, A. L. Peirson, B. Khan, R. Anil, M. Rabbat, S. Krishnan, D. Snider, E. Amid, K. Chen, C. J. Maddison, R. Vasudev, M. Badura, A. Garg, and P. Mattson, "Benchmarking neural network training algorithms," 2023.

[17] K. Li and J. Malik, "Learning to optimize," 2016.

[18] M. Andrychowicz, M. Denil, S. Gomez, M. W. Hoffman, D. Pfau, T. Schaul, B. Shillingford, and N. de Freitas, "Learning to learn by gradient descent by gradient descent," 2016.

[19] L. Metz, N. Maheswaranathan, J. Nixon, D. Freeman, and J. Sohl-Dickstein, "Understanding and correcting pathologies in the training of learned optimizers," in *International Conference on Machine Learning*, pp. 4556–4565, PMLR, 2019.

[20] S. Bengio, Y. Bengio, and J. Cloutier, "Use of genetic programming for the search of a new learning rule for neural networks," in *Proceedings of the First IEEE Conference on Evolutionary Computation. IEEE World Congress on Computational Intelligence*, pp. 324–327 vol.1, 1994.

[21] I. Bello, B. Zoph, V. Vasudevan, and Q. V. Le, "Neural optimizer search with reinforcement learning," 2017.

[22] X. Chen, C. Liang, D. Huang, E. Real, K. Wang, Y. Liu, H. Pham, X. Dong, T. Luong, C.-J. Hsieh, Y. Lu, and Q. V. Le, "Symbolic discovery of optimization algorithms," 2023.

[23] E. Real, C. Liang, D. R. So, and Q. V. Le, "Automl-zero: Evolving machine learning algorithms from scratch," 2020.

[24] C. Ying, A. Klein, E. Real, E. Christiansen, K. Murphy, and F. Hutter, "Nas-bench-101: Towards reproducible neural architecture search," 2019.

[25] X. Dong and Y. Yang, "Nas-bench-201: Extending the scope of reproducible neural architecture search," 2020.

[26] Y. Mehta, C. White, A. Zela, A. Krishnakumar, G. Zabergja, S. Moradian, M. Safari, K. Yu, and F. Hutter, "Nas-bench-suite: Nas evaluation is (now) surprisingly easy," 2022.

[27] A. Bansal, D. Stoll, M. Janowski, A. Zela, and F. Hutter, "Jahs-bench-201: A foundation for research on joint architecture and hyperparameter search," in *Advances in Neural Information Processing Systems* (S. Koyejo, S. Mohamed, A. Agarwal, D. Belgrave, K. Cho, and A. Oh, eds.), vol. 35, pp. 38788–38802, Curran Associates, Inc., 2022.

[28] E. Real, A. Aggarwal, Y. Huang, and Q. V. Le, "Regularized evolution for image classifier architecture search," 2019.

[29] F. Hutter, H. H. Hoos, and K. Leyton-Brown, "Sequential model-based optimization for general algorithm configuration," in *Learning and Intelligent Optimization*, 2011.

[30] I. Loshchilov and F. Hutter, "Decoupled weight decay regularization," in *International Conference on Learning Representations*, 2019.

[31] M. D. Zeiler, "Adadelta: An adaptive learning rate method," 2012.

[32] D. Wang, Y. Liu, W. Tang, F. Shang, H. Liu, Q. Sun, and L. Jiao, "signadam++: Learning confidences for deep neural networks," in *2019 International Conference on Data Mining Workshops (ICDMW)*, pp. 186–195, 2019.

[33] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Köpf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, "Pytorch: An imperative style, high-performance deep learning library," 2019.

[34] S. Adriaensen, H. Rakotoarison, S. Müller, and F. Hutter, "Efficient bayesian learning curve extrapolation using prior-data fitted networks," in *Thirty-seventh Conference on Neural Information Processing Systems*, 2023.

[35] S. Müller, M. Feurer, N. Hollmann, and F. Hutter, "PFNs4BO: In-context learning for Bayesian optimization," in *Proceedings of the 40th International Conference on Machine Learning* (A. Krause, E. Brunskill, K. Cho, B. Engelhardt, S. Sabato, and J. Scarlett, eds.), vol. 202 of *Proceedings of Machine Learning Research*, pp. 25444–25470, PMLR, 23–29 Jul 2023.

[36] N. Hollmann, S. Müller, K. Eggensperger, and F. Hutter, "TabPFN: A transformer that solves small tabular classification problems in a second," in *The Eleventh International Conference on Learning Representations*, 2023.

[37] T. Tornede, A. Tornede, J. Hanselle, F. Mohr, M. Wever, and E. Hüllermeier, "Towards green automated machine learning: Status quo and future directions," *Journal of Artificial Intelligence Research*, vol. 77, pp. 427–457, 2023.

[38] T. K. Ho, "Random decision forests," in *Proceedings of 3rd international conference on document analysis and recognition*, vol. 1, pp. 278–282, IEEE, 1995.

[39] M. Lindauer, K. Eggensperger, M. Feurer, A. Biedenkapp, J. Marben, P. Müller, and F. Hutter, "Boah: A tool suite for multi-fidelity bayesian optimization analysis of hyperparameters," *arXiv:1908.06756 [cs.LG]*, 2019.

Submission Checklist

1. For all authors...

   (a) Do the main claims made in the abstract and introduction accurately reflect the paper's contributions and scope? [Yes]

   (b) Did you describe the limitations of your work? [Yes] See Section 5

   (c) Did you discuss any potential negative societal impacts of your work? [Yes] See Section 5

   (d) Did you read the ethics review guidelines and ensure that your paper conforms to them? `https://2022.automl.cc/ethics-accessibility/` [Yes]

2. If you ran experiments...

   (a) Did you use the same evaluation protocol for all methods being compared (e.g., same benchmarks, data (sub)sets, available resources)? [Yes]

   (b) Did you specify all the necessary details of your evaluation (e.g., data splits, pre-processing, search spaces, hyperparameter tuning)? [Yes] See Appendix F

   (c) Did you repeat your experiments (e.g., across multiple random seeds or splits) to account for the impact of randomness in your methods or data? [Yes]

   (d) Did you report the uncertainty of your results (e.g., the variance across random seeds or splits)? [No]

   (e) Did you report the statistical significance of your results? [Yes] We conducted experiments on caching efficiency. See Appendix D

   (f) Did you use tabular or surrogate benchmarks for in-depth evaluations? [N/A]

   (g) Did you compare performance over time and describe how you selected the maximum duration? [Yes] We compared search methods over time. See Section 4. We did not describe how we selected the maximum duration.

   (h) Did you include the total amount of compute and the type of resources used (e.g., type of GPUs, internal cluster, or cloud provider)? [No]

   (i) Did you run ablation studies to assess the impact of different components of your approach? [Yes] We ran experiments on caching mechanism. See Appendix D

3. With respect to the code used to obtain your results...

   (a) Did you include the code, data, and instructions needed to reproduce the main experimental results, including all requirements (e.g., `requirements.txt` with explicit versions), random seeds, an instructive `README` with installation, and execution commands (either in the supplemental material or as a URL)? [Yes] We provided a link to the code in the introduction.

   (b) Did you include a minimal example to replicate results on a small subset of the experiments or on toy data? [Yes]

   (c) Did you ensure sufficient code quality and documentation so that someone else can execute and understand your code? [Yes]

   (d) Did you include the raw results of running your experiments with the given code, data, and instructions? [No]

   (e) Did you include the code, additional data, and instructions needed to generate the figures and tables in your paper based on the raw results? [No]

4. If you used existing assets (e.g., code, data, models)…

   (a) Did you cite the creators of used assets? [Yes] We used `https://github.com/automl/PFNs` in our benchmark and cited the main paper.

   (b) Did you discuss whether and how consent was obtained from people whose data you're using/curating if the license requires it? [N/A] Not applicable

   (c) Did you discuss whether the data you are using/curating contains personally identifiable information or offensive content? [N/A] Not applicable

5. If you created/released new assets (e.g., code, data, models)…

   (a) Did you mention the license of the new assets (e.g., as part of your code submission)? [Yes] Our code is licensed under the MIT License.

   (b) Did you include the new assets either in the supplemental material or as a URL (to, e.g., GitHub or Hugging Face)? [Yes] It is linked in the introduction.

6. If you used crowdsourcing or conducted research with human subjects…

   (a) Did you include the full text of instructions given to participants and screenshots, if applicable? [N/A] Not applicable

   (b) Did you describe any potential participant risks, with links to Institutional Review Board (IRB) approvals, if applicable? [N/A] Not applicable

   (c) Did you include the estimated hourly wage paid to participants and the total amount spent on participant compensation? [N/A] Not applicable

7. If you included theoretical results…

   (a) Did you state the full set of assumptions of all theoretical results? [N/A] Not applicable

   (b) Did you include complete proofs of all theoretical results? [N/A] Not applicable

A Optimizer Search Space Details

The search space in NOSBench includes a variety of unary, binary, and ternary functions. These functions are shown in Table 1. In designing the search space, we avoided including an extensive array of hand-designed functions. This decision was made to mitigate bias towards specific optimization approaches. Instead, we opted for a selection of essential unary and binary functions from PyTorch (33) library, to enable the discovery of novel optimization strategies. Moreover, to aid the search process and to navigate the optimization landscape more efficiently, we introduced two key ternary functions: `interpolate` and `bias_correct`.

Table 1: Functions in the NOSBench Search Space

Function Type	Functions
Unary	square, exp, log, sign, sqrt, abs, norm, clip, sin, cos, tan, arcsin, arccos, arctan, mean, std, size
Binary	div, mul, add, sub, minimum, maximum, heaviside
Ternary	interpolate, bias_correct

The memory in NOSBench is divided into two regions, with the first nine locations being designated as readonly, containing essential variables and constants for optimizer calculations. Some primitive constant values are provided for joint hyperparameter optimization, enabling search algorithms to calculate different values for various hyperparameters. The subsequent memory locations are dynamically allocated and can grow indefinitely, with a cap set at 20 in our experiments. Each memory location can store either a scalar or an array. Since all of the operations in the search space either produce a scalar or an array of the same size, memory can be homogeneous without any issues.

Table 2: Memory Allocation in NOSBench

Memory Location	Description
`0`	Parameters of the network
`1`	Calculated gradients
`2`	Current optimization step
`3-8`	Primitive constant values for joint hyperparameter optimization (`1.0`, `0.5`, `1e-01`, `1e-02`, `1e-03`, `1e-06`)

Below are programs within the NOSBench search space to illustrate the implementation of common optimization algorithms.

`SGD` (8) can be implemented as follows:

```
NamedProgram(
    "SGD",
    [
        Instruction(Function(mul), [Pointer(1), Pointer(6)], Pointer(9)),
    ],
)
```

In this example, we define a program named `SGD` within the NOSBench search space. The program consists of a single instruction, which is a multiplication operation (`mul`) applied to two operands. The first operand is the gradient vector stored in memory location `memory[1]`, accessed using `Pointer(1)`. The second operand is a predefined constant value representing the learning rate (`1e-02`), accessed via `Pointer(6)`. The result of this multiplication operation is stored in

memory location `memory[9]`, indicated by `Pointer(9)`. Importantly, it is worth noting that the last instruction in a program is used to update the parameters of the neural network.

This instruction represents the update step calculation in the SGD optimization algorithm. The gradient vector, representing the direction of steepest descent, is scaled by the learning rate to determine the magnitude of the update applied to the model parameters.

`AdamW` (30), a more complex optimizer, can be implemented as follows:

```
NamedProgram(
    "AdamW",
    [
        Instruction(square, [1], 11), # Calculate squared gradients
        Instruction(sub, [3, 5], 9), # Calculate Beta1
        Instruction(sub, [3, 7], 10), # Calculate Betat2
        Instruction(interpolate, [12, 1, 9], 12), # First Moment (m)
        Instruction(interpolate, [13, 11, 10], 13), # Second Moment (v)
        Instruction(bias_correct, [12, 9, 2], 14), # Bias correct m (m_hat)
        Instruction(bias_correct, [13, 10, 2], 15), # Bias correct v (v_hat)
        Instruction(mul, [6, 8], 16), # Calculate eps
        Instruction(sqrt, [15], 17), # Calculate square root of v_hat
        Instruction(add, [17, 16], 17), # Add eps to v_hat
        Instruction(div, [14, 17], 19), # Calculate the update
        Instruction(mul, [0, 16], 18), # Calculate the weight decay
        Instruction(add, [19, 18], 19), # Apply weight decay to update
        Instruction(mul, [19, 7], 19), # Apply learning rate to update
    ],
)
```

In this example, we define a program named `AdamW` within the NOSBench search space. The program consists of several instructions representing the operations of the AdamW optimizer. For brevity, we have omitted the explicit use of `Function` and `Pointer` for function and memory pointer definitions, respectively.

B Caching Mechanism Details

The proxy task involves optimizing a simple function with known properties. Let $\mathbf{x}$ denote learnable parameters. The proxy task can be formulated as follows:

$$\frac{\left\|(\mathbf{x} \cdot \mathbf{A} + \mathbf{B})^2\right\|_2}{\left\|(\mathbf{x} \cdot \mathbf{A} + \mathbf{B})^2\right\|_2 + 1} \tag{1}$$

Here, $\cdot$ denotes the dot product operation. An affine transform $(\mathbf{x} \cdot \mathbf{A} + \mathbf{B})$ is applied to $\mathbf{x}$ to make the function asymmetric. The proxy task is to minimize this function (Equation 1).

Hashing mechanism involves computing the weighted average of the output of the function (Equation 1). We used weighted average rather than plain average to also encode the positional information.

The hashing mechanism operates as follows:

1. Initialize the parameters $\mathbf{A}$, $\mathbf{B}$, and $\mathbf{x}$ by randomly sampling from a normal distribution with specified mean and standard deviation. It's important to note that while these parameters are randomly sampled, they remain deterministic across function calls.
2. Instantiate the optimizer that is being hashed.
3. Perform optimization iterations on the proxy task, updating the $\mathbf{x}$ to minimize the loss.
4. Compute a running average of the loss over optimization iterations to encode not only the value but also the position of the loss.

5. Finally, compute the hash value based on the running average of the loss, using the default Python hashing function.

C Meta-training Task

Our meta-training task revolves around Prior-Data Fitted Networks (PFNs) (15), which are designed to fit priors. PFNs are neural networks, typically decoder-only style transformers, that are pre-trained to perform Bayesian prediction for censored samples for a given prior; and at test time perform Bayesian prediction through in-context learning, on real data, in a single forward pass (without fine-tuning, retraining). As such, a prior distribution is essential for this task.

For our prior, we begin by describing the prior of ridge regression, which serves as the foundation for our PFN task. The prior comprises two integral components:

1. The mapping from x to y, denoted as $f = x^T w$, follows a normal distribution:

$$y \sim \mathcal{N}(f, a^2 I)$$

   Here, f represents the function value without noise, and a signifies the standard deviation of our outputs, predetermined to a fixed value, such as 0.1. The symbol w plays a pivotal role in shaping our function, influencing its behavior concerning the input x. It determines essential aspects of the function, such as whether it trends upwards or downwards.

2. We establish a prior distribution over w to further refine our model. This distribution is characterized by:

$$w \sim \mathcal{N}(0, b^2 I)$$

   Here, w represents the latent variable controlling the distribution's characteristics, such as its shape and behavior. This distribution is crucial in shaping the overall structure of our prior.

In Bayesian terms, w is considered a latent variable since it governs the distribution's form without being directly observed.

D Caching Experiments

The caching mechanism is evaluated to assess its effectiveness in improving training speed and reducing computational overhead. We perform experiments focusing on different aspects of the caching mechanism, including cache size and its impact on speedups across multiple runs and methods. In all caching experiments, we used Regularized Evolution (28) and Random Search to assess the effect of caching with different methods. This allows us to evaluate the robustness and generalizability of the caching mechanism across diverse optimization approaches. Additionally, we evaluate the false positive rate of the caching mechanism, comparing it to exact equivalence by comparing proxy equivalent optimizers on meta-traning task.

D.1 Single Run Caching and Cumulative Speed up

To understand the influence of cache size on training speed within a run, we conduct experiments within a single run, starting with an empty cache.

We utilized two optimization methods, Random Search and Regularized Evolution, with 18 seeds and 8000 evaluations/generations each. It's worth noting that for caching experiments, the Regularized Evolution population was initialized randomly, differing from optimizer search experiments where the population was initialized with AdamW (30) and Adadelta (31).

Figure 3 illustrates the results obtained from these experiments. The left plot shows the evolution of cache size within a single run. Each thin line represents an individual run, while the thicker line represents the mean across all runs. Additionally, the diagonal dashed line (x=y) indicates how the

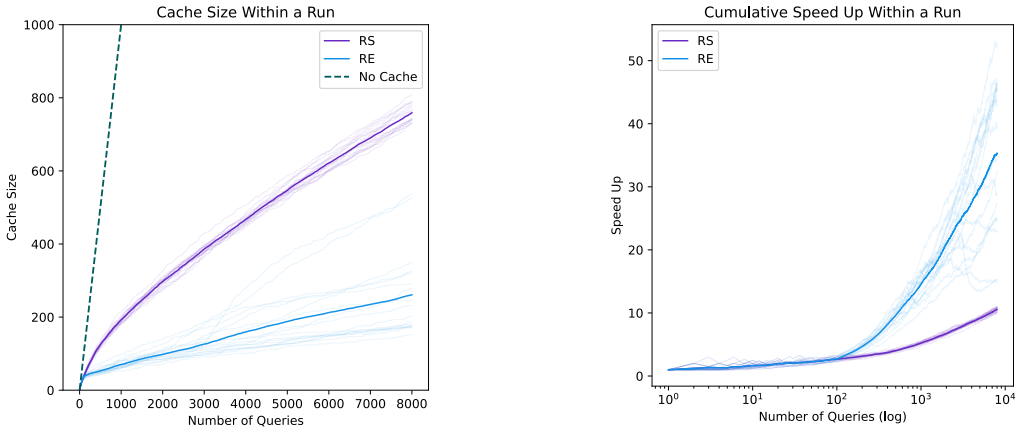

Figure 3: Cache size evolution within a single run **(Left)** and Cumulative speedup within a single run **(Right)**

cache size would increase if there were no caching. As expected, Random Search exhibits a larger cache size, given that the samples are independent and identically distributed (iid). In contrast, Regularized Evolution primarily performs local search, resulting in a more modest increase in cache size.

Additionally, we analyzed the cumulative speedup within a single run, assuming a fixed cost for every cache miss. The right plot in Figure 3 shows the cumulative speedup on a logarithmic scale. If cache_size(x) represents the cache size after x evaluations, the cumulative speedup is calculated as $x/\text{cache\_size}(x)$. Furthermore, it can be observed that Regularized Evolution achieves a higher speedup compared to Random Search, indicating its effectiveness in leveraging caching mechanisms for improved training efficiency.

D.2 Across Runs Caching and Cumulative Speed up

To assess the impact of caching across multiple runs, we conducted a comprehensive experiment using both Random Search and Regularized Evolution. The experiment was structured as follows:

1. **Initial Runs**: We initiated the experiment with 6 different seeds for both Random Search and Regularized Evolution. Each seed was subjected to 8000 evaluations.

2. **Second Runs**: For each seed used in the initial runs, we further expanded the experiment by introducing an additional set of 6 seeds. These additional seeds evaluated for another 8000 iterations.

3. **Final Runs**: Finally, we concluded the experiment with the final set of runs, once again utilizing 6 seeds for both optimization methods.

This experiment design resulted in a total permutation of runs, with 6 runs conducted at each stage. Such a structured approach allowed us to comprehensively evaluate the efficacy of caching mechanisms across multiple iterations and seed variations.

Figure 4 illustrates the results obtained from these experiments. The left plot shows the evolution of cache size across runs. Thin lines represent individual runs, while the thicker line represents the mean across all runs. Vertical lines at the 8000th and 16000th evaluation marks indicate the start of new runs.

Additionally, we analyzed the cumulative speedup across runs, as shown in Figure 4. The right plot displays the cumulative speedup on a logarithmic scale.

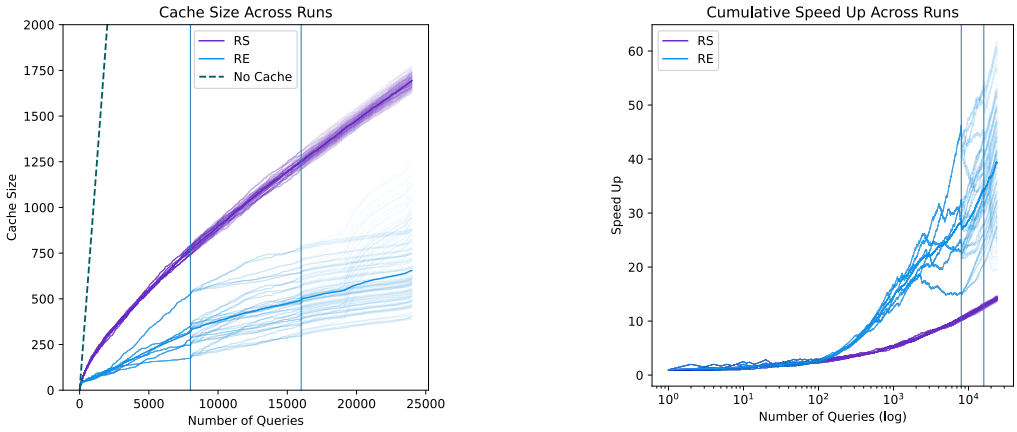

Figure 4: Cache size evolution across runs (**Left**) and Cumulative speedup across runs (**Right**)

In RS, the cache size increases more linearly across runs, as expected. This behavior aligns with the nature of Random Search, where samples are independently and identically distributed (iid). Consequently, each run contributes relatively equally to the cache size growth, resulting in a steady increase over successive runs.In contrast, RE exhibits a more logarithmic increase in cache size across runs. While there are small jumps between each run, indicating the introduction of new samples into the cache, the overall growth is more gradual. This can be attributed to RE's focus on local search, where successive runs tend to explore similar regions of the optimization landscape, leading to a more moderate increase in cache size over time.

D.3 Across Methods Caching

In this experiment, we assessed the impact of initializing the cache using different methods. The table below presents the mean cache hits and standard deviation errors for each combination of methods .

Table 3: Mean cache hits for different cache initialization methods

Cache Initialization	Mean Cache Hits	Mean Cache Hits (excl. NaNs)
RS-RS	0.9382	0.9273
RE-RE	0.9778	0.9740
RS-RE	0.9823	0.9791
RE-RS	0.9179	0.9034

The table summarizes the results obtained from running Random Search (RS) and Regularized Evolution (RE) after initializing the cache with either RS or RE. Note that the values represent the proportion of cache hits to the total number of evaluations. Additionally, the initial runs were solely used to initialize the cache and did not contribute to cache hits. We ran 9 seeds to initialize the cache and different 9 seeds for cache hits.

Regularized Evolution (RE) exhibited higher cache utilization compared to Random Search (RS), consistent with its nature as a more localized search algorithm.

- **RS-RS**: When both cache initialization and cache hit methods were RS, a mean cache hit rate of 0.9382 was observed. This finding aligns with the expected behavior of RS, which tends to explore a broader search space, resulting in a moderate level of cache hits.
- **RS-RE**: Surprisingly, initializing the cache with RS and conducting cache hits with RE yielded the highest mean cache hit rate of 0.9823. This unexpected result may be attributed to the random

mutations in RE, which might produce optimizers similar to those encountered by most RS methods. However, RS's global exploration may introduce different optimizers not seen by all RS methods, leading to diverse cache hits.

- **RE-RS**: In contrast, when RE was used for cache initialization followed by RS for cache hits, a lower mean cache hit rate of 0.9179 was observed. This outcome is consistent with the localized nature of RE, which might have constrained the cache initialization to a narrower search space, resulting in fewer cache hits during subsequent RS iterations.
- **RE-RE**: When both cache initialization and cache hit methods were RE, a mean cache hit rate of 0.9778 was achieved. This result shows RE utilizes the cache efficiently, even when employed iteratively.

D.4 False Positive Rate

In this section, we address the possibility of false positives in our benchmark due to the use of a proxy task for evaluating optimizer equality. False positives occur when two optimizers yield the same hash result in the proxy task but perform differently in the actual task, indicating that they are actually different optimizers. This trade-off is made to prioritize evaluation speed.

To quantify the false positive rate of our benchmark, we conducted experiments comparing randomly selected pairs of optimizers that have the same hash in the proxy task. The experimental setup mirrors that of our caching experiments: we ran 18 seeds for both Regularized Evolution (RE) and Random Search (RS), each for 8000 generations. Subsequently, we sampled 1000 pairs of optimizers from each algorithm and evaluated the false positive rate for each algorithm separately.

Table 4: False positive rates where a pair of optimizers have the same hash but performs different in meta-training task.

Method	False Positive Rate
RE	33/1000
RS	10/1000

For RE, out of 1000 randomly sampled pairs of optimizers with identical hashes in the proxy task, 33 pairs (3.3%) resulted in different performances in the meta-training task. Similarly, for RS, 10 out of 1000 pairs (1.0%) resulted in different performances. These results suggest that while the proxy task efficiently evaluates optimizer equality, there remains a non-negligible chance of false positives. These results emphasize the necessity of refining evaluation methodologies to mitigate the occurrence of false positives.

E Incumbent

Our simplified incumbent is shown below as NOSBench program and pseudocode. We conduct various evaluations to assess its performance and characteristics compared to baseline optimizers such as AdamW (30) and Adadelta (31).

```
Program(
    [
        Instruction(Function(sub), inputs=[13, 1], output=17),
        Instruction(Function(square), inputs=[17], output=11),
        Instruction(Function(sub), inputs=[11, 8], output=13),
        Instruction(Function(add), inputs=[1, 13], output=13),
        Instruction(Function(sub), inputs=[3, 5], output=10),
        Instruction(Function(interpolate), inputs=[13, 11, 10], output=13),
        Instruction(Function(add), inputs=[13, 8], output=15),
```

```
        Instruction(Function(clip), inputs=[6, 15], output=9),
        Instruction(Function(maximum), inputs=[15, 8], output=15),
        Instruction(Function(sqrt), inputs=[15], output=17),
        Instruction(Function(tan), inputs=[7], output=14),
        Instruction(Function(div), inputs=[14, 17], output=19),
        Instruction(Function(maximum), inputs=[9, 17], output=13),
        Instruction(Function(mul), inputs=[19, 1], output=18),
    ]
)
```

Algorithm 1 Incumbent

Require: $f(\theta, D)$: Objective function with parameters θ and dataset D
Require: θ_0: Initial parameter vector

$\beta \leftarrow 0.9$
$m \leftarrow 0, v \leftarrow 0$
while θ not converged **do**
  $g \leftarrow \nabla_\theta f(\theta, D)$ (Compute gradient of objective function)
  $m \leftarrow (v - g)^2$
  $\hat{m} \leftarrow (g + m - 1\text{e}{-6}) * \beta + m * (1.0 - \beta)$
  $\hat{m}_{sqrt} \leftarrow \sqrt{max(\hat{m}, 1\text{e}{-6})}$
  $v \leftarrow max(clip(0.01, \hat{m}), \hat{m}_{sqrt})$
  $\theta \leftarrow \theta - g * \frac{0.001}{\hat{m}_{sqrt}}$ (Update parameters)
end while
return θ (Resulting parameters)

Firstly, we examine the learning curves of the incumbent optimizer in the proxy task and compare them to those of AdamW and Adadelta. Additionally, we analyze the learning curves of the incumbent optimizer in the meta-learning task.

Furthermore, we investigate the norm of the optimizer step of the incumbent optimizer. This provides insights into the behavior of the optimizer during training.

As shown in Figure 5, our incumbent follows a more conservative update approach compared to AdamW and Adadelta, resulting in better performance in both proxy task and meta-training task. These findings also translate to other PFNs (15) tasks, LC-PFN (34), PFNs4BO (35), TabPFN (36).

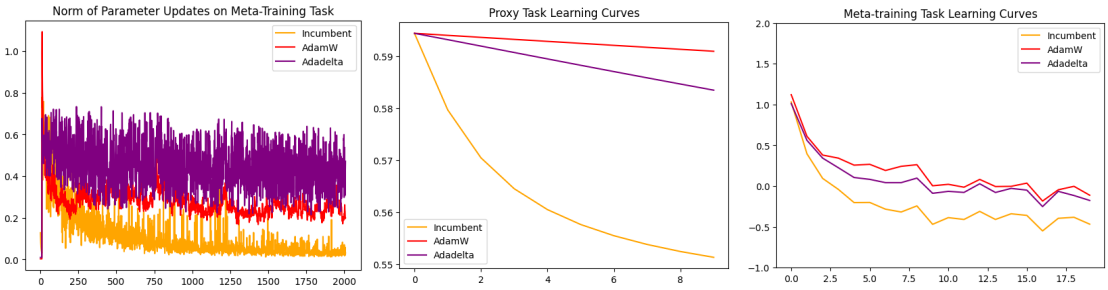

Figure 5: Norm of parameter updates on meta-training task (**Left**), proxy task learning curves (**Middle**), meta-training task learning curves (**Right**)

F Experimental Details

In all our experiments, we set the maximum memory size to 20 and the meta training task is evaluated for 20 epochs, and the final training loss is used as our fitness score.

For Regularized Evolution (28), in all our experiments, we used a population size of 100 and a tournament size of 25. Furthermore, we adopted the same mutation strategies as those outlined in (23; 22), given the similarity of the search space. These strategies include:

- **Insertion of Random Instruction**: A random instruction is inserted into a randomly selected location within the program.
- **Deletion of Instruction**: An instruction is randomly removed from the program.
- **Modification of Instruction**: A random instruction undergoes modification by altering one of its inputs or output to a random memory location.

Algorithm 2 shows Regularized Evolution algorithm in the context of NOSBench.

Algorithm 2 Regularized Evolution for NOSBench

population ← empty queue ▷ The population.
history ← ∅ ▷ Will contain all optimizers.
benchmark ← initialize benchmark ▷ Initialize the benchmark.
while |*population*| < *P* **do** ▷ Initialize population.
    optimizer.program ← SAMPLE_OPTIMIZER_PROGRAM()
    optimizer.performance ← *benchmark*.QUERY(*optimizer.program*) ▷ Query the benchmark.
    add *optimizer* to right of *population*
    add *optimizer* to *history*
end while
while |*history*| < *C* **do** ▷ Evolve for *C* cycles.
    sample ← ∅ ▷ Parent candidates.
    while |*sample*| < *S* **do**
        candidate ← random element from *population* ▷ The element stays in the population.
        add *candidate* to *sample*
    end while
    parent ← highest-performing optimizer in *sample*
    child.program ← MUTATE(*parent.program*) ▷ Mutate the parent optimizer program.
    child.performance ← *benchmark*.QUERY(*child.program*) ▷ Query the benchmark with the mutated optimizer.
    add *child* to right of *population*
    add *child* to *history*
    remove *oldest* from left of *population* ▷ Remove the oldest optimizer.
    discard *oldest*
end while
return highest-performing optimizer in *history*

In the case of SMAC, we employed a Random Forest (38) surrogate model and used log expected improvement as the acquisition function. Both SMAC and Random Search use the same configuration space:

1. **Construction of Search Space**: We defined a search space using ConfigSpace (39), a Python library for defining configuration spaces. Within this space, the number of instructions is initially sampled, capped at 20.
2. **Sampling Instructions**: Following the determination of the number of instructions, we proceed to sample the specified number of instructions. This sampling process involves:

(a) **Function Sampling**: Initially, a function is sampled randomly from the available set of functions.

(b) **Parameter Sampling**: Subsequently, parameters for the selected function are sampled based on whether it is a unary, binary, or ternary function.