PASHA: Efficient HPO with Progressive Resource Allocation

Ondrej Bohdal[1] **Lukas Balles**[2] **Beyza Ermis**[2] **Cedric Archambeau**[2] **Giovanni Zappella**[2]

[1]The University of Edinburgh. Work done during an internship at AWS, Berlin.
[2]Amazon Web Services (AWS).

Abstract Hyperparameter optimization (HPO) and neural architecture search (NAS) are methods of choice to obtain the best-in-class machine learning models, but in practice they can be costly to run. When models are trained on large datasets, tuning them with HPO or NAS rapidly becomes prohibitively expensive for practitioners, even when efficient multi-fidelity methods are employed. We propose an approach to tackle the challenge of tuning machine learning models trained on large datasets with limited computational resources. Our approach, named PASHA, is able to dynamically allocate maximum resources for the tuning procedure depending on the need. The experimental comparison shows that PASHA identifies well-performing hyperparameter configurations and architectures while consuming significantly fewer computational resources than solutions like ASHA.

1 Introduction

Hyperparameter optimization (HPO) and neural architecture search (NAS) yield state-of-the-art models, but are often a very costly endeavor, especially when working with large datasets and models. For example, using the results of (Sharir et al., 2020) we can estimate that evaluating 10 configurations for a 340-million-parameter BERT model (Devlin et al., 2019) on the 15GB Wikipedia and Book corpora would cost around $100,000. To make HPO and NAS more efficient, researchers explored how we can learn from cheaper evaluations (e.g. on a subset of the data) to later allocate more resources only to promising configurations. This created a family of methods often described as multi-fidelity methods. Two well-known algorithms in this family are Successive Halving (SH) (Jamieson and Talwalkar, 2016; Karnin et al., 2013) and Hyperband (HB) (Li et al., 2018).

Multi-fidelity methods significantly lower the cost of the tuning. Li et al. (2018) reported speedups up to 30x compared to standard Bayesian Optimization (BO) and up to 70x compared to random search. Unfortunately, the cost of current multi-fidelity methods is still too high for many practitioners, also because of the large datasets used for training the model. As a workaround, they need to design heuristics which can select a set of hyperparameters or an architecture with a cost comparable to training a single configuration. For example, by training the model with multiple configurations for a single epoch and then selecting the best-performing candidate.

Such heuristics lack robustness and need to be adapted to the specific use-cases in order to provide good results. At the same time, they build on an extensive amount of practical experience suggesting that multi-fidelity methods are often not sufficiently aggressive in leveraging early performance measurements and that identifying the best performing set of hyperparameters (or the best architecture) does not require training a model until convergence. For example, Bornschein et al. (2020) show that it is possible to find the best hyperparameter – number of channels in ResNet-101 architecture (He et al., 2015) for ImageNet (Deng et al., 2009) – using only 10% of the data. We provide a broader overview of related work in Appendix A.

The aim of our work is to design a method that consumes fewer resources than standard multi-fidelity algorithms such as Hyperband (Li et al., 2018) or ASHA (Li et al., 2020) and nonetheless is able to identify configurations that produce models with a similar predictive performance after being fully re-trained from scratch. In order to do this, we propose a variant of ASHA, called

 © 2022 the authors, released under CC BY 4.0

PASHA (Progressive ASHA), that starts with a small amount of initial maximum resources and gradually increases them. ASHA in contrast has a fixed amount of maximum resources. Our empirical evaluation shows that PASHA can save a significant amount of resources while finding similarly well-performing configurations as conventional ASHA.

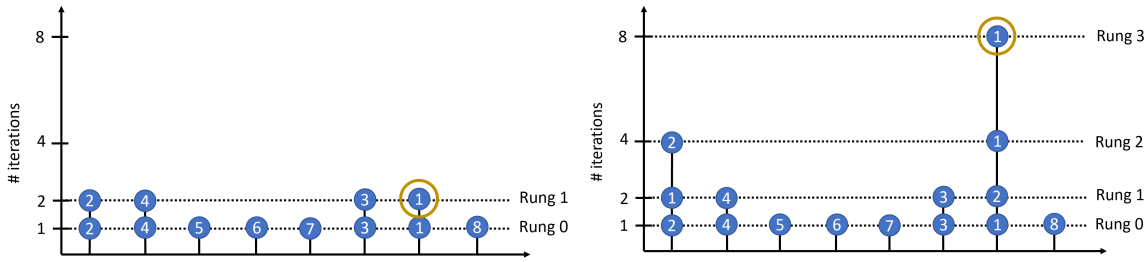
To summarize, our contributions are as follows: 1) We introduce a new approach called PASHA that dynamically selects the amount of maximum resources to allocate for HPO or NAS (up to a certain budget), 2) Our empirical evaluation shows the approach significantly speeds up HPO and NAS without sacrificing the performance, and 3) We show the approach can be successfully combined with sample-efficient strategies based on Bayesian Optimization, highlighting the generality of our approach. Our implementation is based on the Syne Tune library (Salinas et al., 2022) and the code is available at `https://github.com/ondrejbohdal/pasha`.

2 Method

The problem of selecting the best configuration of a machine learning algorithm to be trained is formalized in Jamieson and Talwalkar (2016) as a non-stochastic bandit problem. In this setting the learner (the hyperparameter optimizer) receives N hyperparameter configurations and it has to identify the best performing one with the constraint of not spending more than a fixed amount of resources R (e.g., total number of training epochs) on a specific configuration. Setting R correctly is easy in an academic benchmark where the maximum amount of resources for a configuration is defined by the benchmark creator, but it becomes significantly harder in practical scenarios because the optimal value is problem-dependent and varies significantly. If the value is too small, the model performance will be sub-optimal, while if the budget is too large, the user will incur a significant cost without any practical return. Our algorithm removes the necessity of setting this hyperparameter, avoiding users overestimating it, and providing a significant speedup.

Our approach, named PASHA, is an extension of ASHA (Li et al., 2020) inspired by the "doubling trick" (Auer et al., 1995). PASHA targets improvements for hyperparameter tuning on large datasets by leveraging empirical observations that "crossing points" between learning curves are rare (excluding noise) and almost always happen in the initial part of the training procedure. We discuss more details about our problem setup in Appendix B.

PASHA starts by allowing a small initial amount of maximum resources and progressively increases them if the ranking of the configurations in the top two *rungs* (rounds of promotion) has not stabilized. The ability of our approach to stop early automatically is the key benefit. We illustrate the approach in Figure 1, showing how we stop evaluating configurations for additional rungs if their ranking is stable.



(a) The ranking of the configurations (displayed inside the circles) has stabilized, so we can select the best configuration and stop the search.

(b) The ranking has not stabilized, so we continue.

Figure 1: Illustration of how PASHA stops early if the ranking of configurations has stabilized.

We give a formal description of the PASHA algorithm in Algorithm 1. Given η, a hyperparameter used both in ASHA and PASHA to control how many configurations to prune, PASHA sets the maximum resources to be used for evaluating a configuration using the reduction factor η and the minimum amount of resources r to be used. In particular, PASHA starts with $R_0 = \eta^2 r$ maximum resources allocated to promising configurations and increases them by a factor of η each time the ranking of configurations in the top two rungs becomes inconsistent, in principle "unlocking" a new rung. For example, if we can currently train configurations up to rung 2 and the ranking of configurations in rung 1 and rung 2 is not consistent, then we allow training part of the configurations up to rung 3, i.e. one additional rung. Algorithm 1 uses get_job function that returns the next promotable configuration and is defined in Algorithm 2 in Appendix C.

The minimum amount of resources r is a hyperparameter to be set by the user. It is significantly easier to set compared to R as r is the minimum amount of resources required to see a meaningful difference in the performance of the models, and it can be easily estimated empirically by running a few small-scale experiments.

Algorithm 1 PASHA()

input minimum resource r, reduction factor η
$t = 0, R_0 = \eta^2 r, K_0 = \lfloor \log_\eta(R_0/r) \rfloor$
while desired **do**
    for for each free worker **do**
        (θ, k) = `get_job()`
        `run_then_return_val_loss`$(\theta, r\eta^k)$
    end for
    for completed job (θ, k) with loss l **do**
        Update configuration θ in rung k with loss l.
        if $k \geq K_t - 2$ **then**
            Σ_k = `configuration_ranking`(k)
        end if
        if $k = K_t - 1$ and $\Sigma_k \not\equiv \Sigma_{k-1}$ **then**
            $t = t + 1$
            $R_t = \eta^t R_0$
            $K_t = \lfloor \log_\eta(R_t/r) \rfloor$
        end if
    end for
end while

Given that deep learning algorithms typically rely on stochastic gradient descent, ranking inconsistencies can occur between similarly performing configurations. Hence, we need some benevolence in estimating the ranking. We propose to use a soft ranking approach (detailed in Appendix D) where we group configurations based on their validation performance metric (e.g., accuracy). Configurations are considered equivalent if the performance difference is smaller than a fixed value ϵ (or equal to it). For practical purposes we also set a boundary for maximum amount of resources R so that PASHA can default to ASHA if needed and avoid increasing the resources indefinitely. While it is not generally reached, it provides a safety net in case some components are misconfigured or the configurations do not reflect the expectations of the user.

3 Experiments

In this section we quantify the advantage provided by PASHA. Its goal is not to provide a model with a higher accuracy, but to identify the best configuration in a shorter amount of time such that we can then re-train the model from scratch. Overall, we target a significantly faster tuning time

and on-par predictive performance when comparing with the models identified by state-of-the-art optimizers like ASHA. Re-training after HPO or NAS is important because HPO and NAS in general require to reserve a significant part of the data (often around 20 or 30%) to be used as a validation set. Training with fewer data is not desirable because in practice it is observed that training a model on the union of training and validation sets provides better results.

We tested our method on two different sets of experiments. The first set evaluates the algorithm on NAS problems and employs NASBench201 (Dong and Yang, 2020), while the second set focuses on hyperparameter optimization and is ran over two custom search spaces using public datasets. Our HPO experiments are in Appendix E, and they show the speedup can be large, for example taking only one fifth of the time compared to ASHA for large datasets with millions of examples.

3.1 Setup

Our experimental setup consists of two phases: 1) run the hyperparameter optimizer until $N = 256$ candidate configurations are evaluated; and 2) use the best configuration identified in the first phase to re-train the model from scratch. For the purpose of these experiments we re-train all the models using only the training set. This avoids introducing an arbitrary choice on the validation set size and allows us to leverage standard benchmarks such as NASBench201. In real-world applications the model can be trained on both training and validation sets. All our results report only the time invested in identifying the best configuration since the re-training time is comparable for all optimizers. All results are averaged over multiple repetitions, with the details specified for each set of experiments separately.

We use 4 workers to perform evaluations in parallel and asynchronously. The choice of R is sensitive for ASHA since it can make the optimizer consume too many resources and penalize the performance of the algorithm. To have a fair comparison, we make R dataset-dependent adopting either the maximum amount of resources for tabulated benchmarks (i.e., for NASBench201) or by performing some preliminary experiments and setting it to the value for which the loss curve flattens. r is also dataset dependent and η, the halving factor, is set to 3 unless otherwise specified. The same values are used for both ASHA and PASHA. For the soft ranking in PASHA we selected $\epsilon = 2.5\%$, which consistently provides strong results across various scenarios.

We compare PASHA with ASHA (Li et al., 2020), a recent state-of-the-art approach for hyperparameter optimization already described earlier, and selected baselines described separately.

3.2 NAS experiments

For our NAS experiments we leverage the well-known NASBench201 (Dong and Yang, 2020) benchmark. The task is to identify the network structure providing the best accuracy on three different datasets (CIFAR-10, CIFAR-100 and ImageNet16-120) independently. We use $r = 1$ epoch and $R = 200$ epochs. We repeat the experiments using 5 random seeds for the scheduler and 3 random seeds for NASBench201 (all that are available), resulting in 15 repetitions. Some configurations in NASBench201 do not have all seeds available, so we impute them by averaging over the available seeds. To measure the predictive performance we report the best accuracy on the combined validation and test set provided by the creators of the benchmark.

We also report performance for two simple baselines: the "one-epoch baseline" that trains all configurations for one epoch (the minimum available resources) and then selects the most promising configuration, and "random baseline" that randomly selects the configuration. For our one-epoch baseline we sample $N = 256$ configurations, using the same scheduler and NASBench201 seeds as for PASHA and ASHA. For our random baseline we sample $N = 2560$ configurations to obtain a better performance estimate of a model with random configuration.

The results in Table 1 suggest that PASHA consistently leads to strong improvements in runtime, while achieving similar accuracy values as the baseline algorithm ASHA. The one-epoch baseline has noticeably worse accuracies than ASHA or PASHA, suggesting that PASHA does a good job of

deciding when to continue increasing the resources – it does not stop too early. Random baseline is a lot worse than the one-epoch baseline, so there is value in performing NAS. We also report the maximum resources used to find how early the ranking becomes stable in PASHA.

It is important to observe that the time required to train a model is about 1.3h for CIFAR-10 and CIFAR-100, and about 4.1h for ImageNet16-120, making the total tuning time required by PASHA comparable with the one required for training.

We also ran additional experiments testing PASHA with alternative ranking functions (Appendix F), a reduction factor of $\eta = 2$ and $\eta = 4$ instead of $\eta = 3$ (Appendix G), and the usage of PASHA as a scheduler in MOBSTER (Klein et al., 2020) that uses Bayesian Optimization (Appendix H). These experiments provided similar findings as the above and are described in the appendix. More fundamentally, our MOBSTER experiments have shown we can successfully combine PASHA with more advanced search strategies based on Bayesian Optimization to obtain improvements in accuracy at a fraction of the time.

Table 1: NASBench201 results. PASHA leads to large improvements in runtime, while achieving similar accuracy as ASHA. PASHA uses soft ranking and $\epsilon = 0.025$ (2.5%).

Dataset	Approach	Accuracy (%)	Runtime	Speedup factor	Max resources
CIFAR-10	ASHA	93.85 ± 0.25	3.0h ± 0.6h	1.0x	200.0 ± 0.0
	PASHA	93.78 ± 0.31	2.3h ± 0.5h	1.3x	144.5 ± 59.4
	One epoch baseline	93.30 ± 0.61	0.3h ± 0.0h	8.5x	1.0 ± 0.0
	Random baseline	72.93 ± 19.55	0.0h ± 0.0h	–	0.0 ± 0.0
CIFAR-100	ASHA	71.69 ± 1.05	3.2h ± 0.9h	1.0x	200.0 ± 0.0
	PASHA	71.41 ± 1.15	1.5h ± 0.7h	2.1x	88.3 ± 74.4
	One epoch baseline	65.57 ± 5.53	0.3h ± 0.0h	9.2x	1.0 ± 0.0
	Random baseline	42.98 ± 18.34	0.0h ± 0.0h	–	0.0 ± 0.0
ImageNet16-120	ASHA	45.63 ± 0.81	8.8h ± 2.2h	1.0x	200.0 ± 0.0
	PASHA	46.01 ± 1.00	3.2h ± 1.0h	2.8x	28.6 ± 27.7
	One epoch baseline	41.42 ± 4.98	1.0h ± 0.0h	8.8x	1.0 ± 0.0
	Random baseline	20.97 ± 10.01	0.0h ± 0.0h	–	0.0 ± 0.0

4 Limitations and broader impact

PASHA is an algorithm that is designed to make HPO and NAS more accessible to machine learning practitioners as it significantly lowers the time and cost required to find well-performing configurations. Its limitation is similar to that of ASHA and other multi-fidelity methods: it may early prune configurations that would perform well if they were trained for longer before making a stopping decision. Another limitation is that setting the ϵ parameter for soft ranking requires some amount of domain expertise. We ran all our experiments using the same value ($\epsilon = 0.025$) and found that it provides a good rule-of-thumb in practice.

5 Conclusions and future work

In this work we introduced a new variant of Successive Halving, called PASHA, that progressively increases the resources allocated to promising configurations. PASHA considers their rankings at successive rung levels and if the rankings have stabilized, stops the HPO procedure, returning the best configuration found. Despite its simplicity, PASHA has led to strong improvements in the tuning runtime. For example, it halves the time to find the best configuration compared to ASHA without a noticeable impact on the quality of the found configuration. In the case of large datasets with millions of examples, the speedup has been larger, for example taking only one fifth of the time compared to ASHA.

References

Auer, P., Cesa-Bianchi, N., Freund, Y., and Schapire, R. (1995). Gambling in a rigged casino: The adversarial multi-armed bandit problem. In *36th Annual Symposium on Foundations of Computer Science*, pages 322–331.

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

Bergstra, J. and Bengio, Y. (2012). Random search for hyper-parameter optimization. *JMLR*, 13:281–305.

Blackard, J. A., Dean, D. J., and Anderson, C. W. (1998). Covertype data set. In *UCI Machine Learning Repository*.

Bornschein, J., Visin, F. V., and Osindero, S. (2020). Small data, big decisions: Model selection in the small-data regime. In *ICML*.

Deng, J., Dong, W., Socher, R., Li, L.-j., Li, K., and Fei-fei, L. (2009). Imagenet: A large-scale hierarchical image database.

Devlin, J., Chang, M.-W., Lee, K., and Toutanova, K. (2019). BERT: Pre-training of deep bidirectional transformers for language understanding. In *ACL*.

Domhan, T., Springenberg, J. T., and Hutter, F. (2015). Speeding up automatic hyperparameter optimization of deep neural networks by extrapolation of learning curves. In *IJCAI*.

Dong, X. and Yang, Y. (2020). Nas-bench-201: Extending the scope of reproducible neural architecture search. In *ICLR*.

Falkner, S., Klein, A., and Hutter, F. (2018). BOHB: Robust and efficient hyperparameter optimization at scale. In *ICML*.

George, N. (2019). All lending club loan data. In *Kaggle*.

He, K., Zhang, X., Ren, S., and Sun, J. (2015). Deep residual learning for image recognition. In *CVPR*.

Ivkin, N., Karnin, Z., Perrone, V., and Zappella, G. (2021). Cost-aware adversarial best arm identification. In *ICLR NAS workshop*.

Jamieson, K. and Talwalkar, A. (2016). Non-stochastic best arm identification and hyperparameter optimization. In *AISTATS*.

Karnin, Z., Koren, T., and Somekh, O. (2013). Almost optimal exploration in multi-armed bandits. In *International Conference on Machine Learning*, pages 1238–1246.

Kingma, D. P. and Ba, J. (2015). Adam: a method for stochastic optimization. In *ICLR*.

Klein, A., Tiao, L. C., Lienart, T., Archambeau, C., and Seeger, M. (2020). Model-based asynchronous hyperparameter and neural architecture search. In *arXiv*.

Li, L., Jamieson, K., Rostamizadeh, A., Gonina, E., Hardt, M. H., Recht, B., and Talwalkar, A. (2020). A system for massively parallel hyperparameter tuning. In *MLSys*.

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

Mohr, F. and van Rijn, J. N. (2022). Learning curves for decision making in supervised machine learning - a survey. In *arXiv*.

Salinas, D., Seeger, M., Klein, A., Perrone, V., Wistuba, M., and Archambeau, C. (2022). Syne tune: A library for large scale hyperparameter tuning and reproducible research. In *First Conference on Automated Machine Learning (Main Track)*.

Sharir, O., Peleg, B., and Shoham, Y. (2020). The cost of training nlp models: A concise overview. In *arXiv*.

Shim, J.-h., Kong, K., and Kang, S.-J. (2021). Core-set sampling for efficient neural architecture search. In *ICML Workshops*.

Viering, T. and Loog, M. (2021). The shape of learning curves: a review. In *arXiv*.

Visalpara, S., Killamsetty, K., and Iyer, R. (2021). A data subset selection framework for efficient hyper-parameter tuning and automatic machine learning. In *ICML Workshops*.

Webber, W., Moffat, A., and Zobel, J. (2010). A similarity measure for indefinite rankings. In *ACM Transactions on Information Systems*.

Xu, Y., Xie, L., Zhang, X., Chen, X., Qi, G.-J., Tian, Q., and Xiong, H. (2020). PC-DARTS: Partial channel connections for memory-efficient architecture search. In *ICLR*.

Zavadskyy, V. (2017). Lots of code. In *Kaggle*.

Zhou, D., Zhou, X., Zhang, W., Loy, C. C., Yi, S., Zhang, X., and Ouyang, W. (2020). EcoNAS: Finding proxies for economical neural architecture search. In *CVPR*.

6 Reproducibility 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] We wrote an accurate abstract and introduction.

   (b) Did you describe the limitations of your work? [Yes] We have a dedicated section that discusses the limitations (and broader impact).

   (c) Did you discuss any potential negative societal impacts of your work? [Yes] We have a section that discusses this topic.

   (d) Have you read the ethics review guidelines and ensured that your paper conforms to them? [Yes] We have read them and ensured our paper conforms to them.

2. If you are including theoretical results…

   (a) Did you state the full set of assumptions of all theoretical results? [N/A] Our work does not include theoretical results.

   (b) Did you include complete proofs of all theoretical results? [N/A] Our work does not include theoretical results.

3. If you ran experiments…

(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 version), an instructive `README` with installation, and execution commands (either in the supplemental material or as a URL)? [Yes] We provide the code as an extension of the SyneTune library, making it simple to use our proposed algorithm. To make the experiments easily reproducible, we provide a Jupyter notebook that reproduces the main results (NASBench201). The code is available at `https://anonymous.4open.science/r/pasha`.

(b) Did you include the raw results of running the given instructions on the given code and data? [Yes] Our Jupyter notebook shows the raw results.

(c) Did you include scripts and commands that can be used to generate the figures and tables in your paper based on the raw results of the code, data, and instructions given? [Yes] Our Jupyter notebook outputs LaTeX code used to generate the content of the tables in our paper.

(d) Did you ensure sufficient code quality such that your code can be safely executed and the code is properly documented? [Yes] We have detailed explanations in the Jupyter notebook, the code is well-commented and docstrings for functions are provided. Unit tests are part of the public repository. We also give detailed comments for the parts of the SyneTune library extended in our work.

(e) Did you specify all the training details (e.g., data splits, pre-processing, search spaces, fixed hyperparameter settings, and how they were chosen)? [Yes] We specify all of the details needed to reproduce the experiments. The main ones are reported in the experiments section, the remaining ones are in the appendix (see details in the text).

(f) Did you ensure that you compared different methods (including your own) exactly on the same benchmarks, including the same datasets, search space, code for training and hyperparameters for that code? [Yes] All methods are evaluated in exactly the same way.

(g) Did you run ablation studies to assess the impact of different components of your approach? [Yes] We have ablation studies that investigate a wide variety of ranking functions (and hyperparameters for them), various reduction factor values and also the combination of PASHA and ASHA with Bayesian Optimization.

(h) Did you use the same evaluation protocol for the methods being compared? [Yes] We have tried to ensure the protocol is the same for all methods.

(i) Did you compare performance over time? [N/A] Performance over time is not an appropriate way to compare PASHA with ASHA and other methods because the key idea behind PASHA is to train all configurations only partially (e.g. for a few epochs) and identify the most promising ones early using significantly fewer resources. PASHA prioritizes evaluating more configurations rather than training them for longer, so the intermediate performance of configurations (without retraining) will be worse in PASHA. It is only after retraining that the performance should be comparable.

(j) Did you perform multiple runs of your experiments and report random seeds? [Yes] We have repeated our NASBench201 experiments 15 times, using combinations of 5 scheduler seeds and 3 NASBench201 seeds. Our HPO experiments are repeated 10 times. The random seeds are reported in the code for NASBench201.

(k) Did you report error bars (e.g., with respect to the random seed after running experiments multiple times)? [Yes] We report the standard deviations for all our experiments.

(l) Did you use tabular or surrogate benchmarks for in-depth evaluations? [Yes] We used NASBench201 for our main experiments.

(m) Did you include the total amount of compute and the type of resources used (e.g., type of GPUs, internal cluster, or cloud provider)? [Yes] We have a section in the appendix that specifies the details about the compute we have used.

(n) Did you report how you tuned hyperparameters, and what time and resources this required (if they were not automatically tuned by your AutoML method, e.g. in a NAS approach; and also hyperparameters of your own method)? [Yes] We have a section in the appendix that explains these details.

4. If you are using existing assets (e.g., code, data, models) or curating/releasing new assets…

(a) If your work uses existing assets, did you cite the creators? [Yes] We cite the creators of the existing assets.

(b) Did you mention the license of the assets? [Yes] We have a dedicated section in the appendix to discuss the license of the assets.

(c) Did you include any new assets either in the supplemental material or as a URL? [Yes] We provide the source code for PASHA in the supplemental material and PASHA has also been open-sourced within SyneTune.

(d) Did you discuss whether and how consent was obtained from people whose data you're using/curating? [N/A] The data that we use have already been open-sourced and so additional consent is not required.

(e) Did you discuss whether the data you are using/curating contains personally identifiable information or offensive content? [N/A] The data that we use do not include personally identifiable information or offensive content.

5. 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] We did not use crowdsourcing and did not conduct research with human subjects.

(b) Did you describe any potential participant risks, with links to Institutional Review Board (IRB) approvals, if applicable? [N/A] We did not use crowdsourcing and did not conduct research with human subjects.

(c) Did you include the estimated hourly wage paid to participants and the total amount spent on participant compensation? [N/A] We did not use crowdsourcing and did not conduct research with human subjects.

Acknowledgements. We would like to thank the Syne Tune developers for providing us with a library to easily extend and use in our experiments. We would like to thank Aaron Klein, Matthias Seeger and David Salinas for their support on questions regarding Syne Tune and hyperparameter optimization more broadly. We would also like to thank Valerio Perrone, Sanyam Kapoor and Aditya Rawal for insightful discussions when working on the project.

A Related work

Machine learning systems used in real-world applications often rely on a large number of hyperparameters, and require testing many combinations of them in order to identify the optimal solution. This makes data-inefficient techniques such as Grid Search or Random Search (Bergstra and Bengio, 2012) very expensive in most practical scenarios. Various approaches have been proposed to find good parameters more quickly, and they can be classified into two main families: 1) Bayesian Optimization: evaluate the most promising configurations by modelling their performance. The methods are sample-efficient but they are often designed for environments with limited amount of parallelism; 2) Multi-fidelity: sequentially allocate more resources to configurations with better performance and allow high level of parallelism during the tuning. From these two families, multi-fidelity methods have typically been faster when run at scale and are the focus of this work. It is also possible to combine ideas from the two families together, for example as done in BOHB by Falkner et al. (2018), and we test a similar method in our experiments.

Successive halving (Karnin et al., 2013; Jamieson and Talwalkar, 2016) is conceptually the simplest multi-fidelity method. Its key idea is to run all configurations using a small amount of resources, which depend on the minimum resources and the minimum early stopping rate, and then successively promote only a fraction of the most promising configurations to be trained using more resources. Another popular multi-fidelity method, called Hyperband (Li et al., 2018), performs successive halving with different early stopping rates. ASHA (Li et al., 2020) extends the simple and very efficient idea of successive halving by introducing asynchronous evaluation of different configurations, which leads to further practical speedups thanks to better resource allocation.

Related to the problem of efficiency in HPO, cost-aware HPO explicitly accounts for the cost of the evaluations of different configurations. Previous work on cost-aware HPO for multi-fidelity algorithms such as CAHB (Ivkin et al., 2021) keeps a tight control on the budget spent during the HPO process. This is different from our work, as we reduce the budget spent by terminating the HPO procedure early instead of allocating the compute budget in its entirety. Moreover, PASHA could be combined with CAHB to leverage the cost-based resources allocation.

Recently, work considered dataset subsampling to speedup the search of the best hyperparameters or architectures. Shim et al. (2021) have combined coresets with PC-DARTS (Xu et al., 2020) and showed that they can find well-performing architectures using only 10% of the data and 8.8x less search time. Similarly, Visalpara et al. (2021) have combined subset selection methods with the Tree-structured Parzen Estimator (TPE) for hyperparameter optimization (Bergstra et al., 2011). With a 5% subset they obtained between an 8x to 10x speedup compared to standard TPE. However, in both cases it is difficult to say in advance what subsampling ratio to use. For example, the 10% ratio in (Shim et al., 2021) incurs no decrease in accuracy, while reducing further to 2% leads to a substantial (2.6%) drop in accuracy. In practice, it is difficult to find a trade-off between the time required for tuning (proportional to the subset size) and the loss of performance for the final model because these change, sometimes wildly, between datasets. We approach this issue in a principled way using our PASHA algorithm.

Further, Zhou et al. (2020) have observed that for a fixed number of iterations, rank consistency is better if we use more training samples and fewer epochs rather than fewer training samples and more epochs. This observation gives further motivation for using the whole dataset for HPO/NAS and using approaches like PASHA to save computational resources.

B Problem setup – additional discussion

The considered setting introduced in (Jamieson and Talwalkar, 2016) can be extended with additional assumptions based on empirical observation, removing some extreme cases and leading to a more practical setup. In particular, when working with large datasets we observe that the curve of the loss for configurations (called arms in the bandit literature) continuously decreases (in expectation).

Moreover, "crossing points" between the curves are rare (excluding noise), and they are almost always in the initial part of the training procedure. For example, analysis from Domhan et al. (2015) suggests that crossings between curves at later stages of training are relatively rare. Further, Viering and Loog (2021); Mohr and van Rijn (2022) provide an analysis of learning curves and note that in practice most learning curves are well-behaved, with Bornschein et al. (2020) reporting similar findings.

More formally, let us define R as the total number of resources needed to train an ML algorithm to convergence, given $\Sigma_m(i)$ the ranking of configuration i after using m resources for training, $\exists R^* \ll R : \forall i \in [n], \forall r > R^*, \Sigma_{R^*}(i) = \Sigma_r(i)$. The existence of such a quantity, limited to the best performing configuration, is also assumed by Jamieson and Talwalkar (2016), and it is leveraged to quantify the budget required to identify the best performing configuration. If we knew R^*, it would be sufficient to run all configurations with exactly that amount of resources to identify the best one and then just train the model from scratch with all the data using that configuration. Unfortunately that quantity is unknown and can only be estimated during the optimization procedure.

C Algorithm – additional details

We describe get_job function that returns the next promotable configuration in Algorithm 2.

Algorithm 2 `get_job()`

```
// Check if there is a promotable config.
for k = K_t − 1, ..., 1, 0 do
    candidates = top_k(rung k, |rung k|/η)
    promotable = {c ∈ candidates : c not promoted}
    if |promotable| > 0 then
        return promotable[0], k + 1
    end if
    // If not, grow bottom rung.
    Draw random configuration θ.
    return θ, 0
end for
```

D Soft ranking

In soft ranking, configurations are still sorted by predictive performance but they are considered equivalent if the performance difference is smaller than a fixed value ϵ (or equal to it). Instead of producing a sorted list of configuration, this provides a list of lists where for every position of the ranking there is a list of equivalent configurations. The concept is explained graphically in Figure 2, and we also provide a formal definition. For a set of n configurations $c_1, c_2, \cdots, c_i, \cdots, c_n$ and performance metric f (e.g. accuracy) with $f(c_1) \leq f(c_2) \leq \cdots \leq f(c_i) \leq \cdots \leq f(c_n)$, soft rank at position i is defined as

$$\text{soft rank}_i = \{c \in \text{configurations} : |f(c_i) - f(c)| \leq \epsilon\}.$$

We tested different ranking functions and reported the results in the appendix, but in most cases the simple soft-ranking solution with ϵ set to 0.025 provides solid results.

E HPO experiments

E.1 Overview

To validate the effectiveness of PASHA in different scenarios, we designed a set of experiments where the search spaces are defined over the hyperparameters used to train a Multi-Layer Perceptron

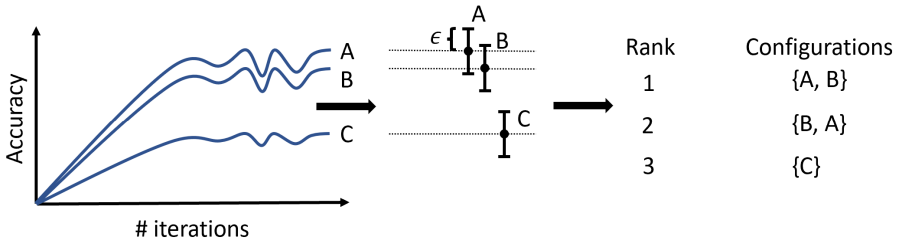


Figure 2: Illustration of soft ranking. There are three lists with the first two containing two items because the scores of the two configurations are closer to each other than ϵ.

(MLP). Two different search spaces were defined, and we trained the MLP on three different public datasets: Codes (Zavadskyy, 2017), LendingC (George, 2019) and CoverType (Blackard et al., 1998) – all of which are tabular datasets with a classification task. The hyperparameters considered include for example the number of layers, their size, minibatch size, learning rate, type of the optimizer and the use of a residual connection. The goal is to demonstrate that the speedup improvements brought by PASHA are not tied to a specific benchmark but generalize to different HPO tasks.

E.2 Search spaces

We have considered two search spaces, defined in Table 2 and 3. In search space #1, the neural network is a small fully-connected multi-layer perceptron, where the precise architecture is considered a hyperparameter. We consider seven different types of architecture, ranging from a single layer neural network up to a neural network with three layers. The number of units in each layer is variable, but it is relatively small in the range 50 to 300 units per layer. In all cases we use Adam (Kingma and Ba, 2015) optimizer. This search space can be considered as easier compared to the second search space, which we define in Table 3. We always use a two-layer fully-connect neural network in search space #2. In the second search space the range of batch size values is from 256 to 1024 examples for Codes and LendingC datasets, while it is from 8 to 128 for the significantly smaller CoverType dataset.

Table 2: Search space #1 for our HPO experiments – can be considered as simpler.

Hyperparameter	Options
Learning rate	log-uniform$(10^{-6}, 10^{-2})$
Batch size	rand-int$(128, 512)$
Dropout	uniform$(0.25, 0.5)$
Architecture	7 variations
Weight decay	log-uniform$(10^{-12}, 10^{-2})$
Residual connection	Yes or no

E.3 Datasets

We have used three publicly available tabular datasets, Codes (Zavadskyy, 2017), LendingC (George, 2019) and CoverType (Blackard et al., 1998). Both Codes and CoverType are multi-way classification problems, while LendingC is a binary classification problem. We show the total number of examples and dataset size after feature-processing of each dataset in Table 4. We can see these are larger-scale datasets with many datapoints. We randomly split the dataset into training, validation and test sets using 80/10/10 split. Feature-processing is generally kept to minimum and includes only the

Table 3: Search space #2 for our HPO experiments – can be considered as more challenging.

Hyperparameter	Range
Number of units in layer 1	rand-int$(4, 1024)$
Number of units in layer 2	rand-int$(4, 1024)$
Batch size	rand-int$(256, 1024)$ or rand-int$(8, 128)$
Dropout in layer 1	uniform$(0, 0.99)$
Dropout in layer 2	uniform$(0, 0.99)$
Learning rate	log-uniform$(10^{-6}, 1.0)$
Weight decay	log-uniform$(10^{-8}, 1.0)$
Momentum	uniform$(0, 0.99)$
Optimizer	SGD or Adam
Activation	ReLU, Leaky ReLU, Sigmoid, Tanh or ELU

standard steps such as converting categorical variables into one-hot representation[1]. The datasets are not balanced, so we report F1 macro score in our experiments.

Table 4: Datasets that we used for HPO.

Dataset	N	GB
Codes (Zavadskyy, 2017)	22,889,691	15.9
LendingC (George, 2019)	1,760,668	29.3
CoverType (Blackard et al., 1998)	581,012	2.5

E.4 Experiments

Since in this case we are running training jobs for every experiment instead of levering a tabulated benchmark, we have the flexibility to define rung levels in terms of the number of datapoints processed rather than the number of epochs. Minimum resources are defined as 1% of the data for Codes and LendingC, while it is 2% for CoverType. Maximum resources are defined as the equivalent of 10 epochs for Codes, 1 epoch for LendingC and 10 epochs for CoverType. These values were selected to reflect the respective sizes of the datasets.

We also define a simple baseline, *PASHA – no increase*, that corresponds to training each configuration with the minimum resources, doing one round of promotion and then selecting the best configuration (PASHA with no resource increases). This baseline was introduced to mimic the behaviour of the "one-epoch baseline" in a benchmark where resources are not measured in epochs.

We give the results of our experiments in Table 5 and 6, across 10 repetitions. We see that in most cases PASHA is able to obtain similar F1 macro score as ASHA, with a significant improvement in the runtime. The speed improvement is most notable in the case of the Codes dataset that includes almost 23 million examples – more than 5x speedup, highlighting that PASHA can make the biggest impact for large-scale datasets where we can find a strong configuration even before seeing all datapoints. *PASHA – no increase* is usually not sufficient to obtain similarly good accuracy as ASHA.

F NAS experiments with alternative ranking functions

We have considered a variety of alternative ranking functions, among them simple ranking (equivalent to soft ranking with $\epsilon = 0.0$), Rank Biased Overlap (RBO) (Webber et al., 2010), our own reciprocal rank regret metric (RRR) that considers the objective values of configurations, and also

[1]The implementation of the feature processors is available at `https://github.com/aws/sagemaker-scikit-learn-extension`

Table 5: Results of the comparison of various tabular datasets on search space #1.

Dataset	Approach	F1 macro (%)	Runtime	Speedup factor
Codes	ASHA	76.56 ± 0.49	15.1h ± 6.0h	1.0x
	PASHA	76.64 ± 0.63	2.9h ± 0.5h	5.2x
	PASHA – soft ranking 2σ	74.98 ± 1.92	2.0h ± 0.8h	7.6x
	PASHA – no increase	70.22 ± 5.60	1.5h ± 0.0h	9.9x
LendingC	ASHA	97.70 ± 0.95	1.3h ± 0.1h	1.0x
	PASHA	86.22 ± 9.85	1.0h ± 0.5h	1.3x
	PASHA – soft ranking 2σ	98.28 ± 0.40	1.1h ± 0.2h	1.2x
	PASHA – no increase	97.00 ± 1.73	0.4h ± 0.0h	2.9x
CoverType	ASHA	79.04 ± 1.16	1.0h ± 0.3h	1.0x
	PASHA	77.19 ± 12.09	0.5h ± 0.1h	2.0x
	PASHA – soft ranking 2σ	78.52 ± 1.20	0.5h ± 0.1h	1.9x
	PASHA – no increase	59.72 ± 14.31	0.2h ± 0.0h	5.7x

Table 6: Results of the comparison of various tabular datasets on search space #2.

Dataset	Approach	F1 macro (%)	Runtime	Speedup factor
Codes	ASHA	76.59 ± 1.11	18.3h ± 8.1h	1.0x
	PASHA	76.31 ± 1.78	5.4h ± 1.4h	3.4x
	PASHA – soft ranking 2σ	74.19 ± 3.69	3.3h ± 0.8h	5.6x
	PASHA – no increase	68.22 ± 5.84	2.0h ± 0.1h	9.1x
LendingC	ASHA	98.19 ± 0.15	1.5h ± 0.9h	1.0x
	PASHA	87.66 ± 21.09	1.1h ± 0.7h	1.3x
	PASHA – soft ranking 2σ	93.07 ± 15.87	1.2h ± 0.4h	1.3x
	PASHA – no increase	90.05 ± 20.68	0.4h ± 0.0h	3.5x
CoverType	ASHA	77.70 ± 3.20	1.5h ± 1.0h	1.0x
	PASHA	78.18 ± 3.93	0.9h ± 0.3h	1.7x
	PASHA – soft ranking 2σ	75.14 ± 15.46	0.4h ± 0.1h	4.1x
	PASHA – no increase	53.52 ± 14.12	0.2h ± 0.0h	6.6x

further variations of soft ranking, for example with automatic selection of ϵ based on the standard deviation of objective values in the previous rung.

F.1 Description

PASHA employs a ranking function whose choice is completely arbitrary. For the purpose of our experiments, we just considered the ranking of the candidates up to some precision in the evaluation. In this set of experiments we would like to evaluate different criteria to define the ranking of the candidates. We describe the functions considered next.

F.1.1 Direct ranking. As a baseline, we study if we can use the simple ranking of configurations by predictive performance (e.g., sorting from the ones with the highest accuracy to the ones with the lowest). If any of the configurations change their order, we consider the ranking unstable and increase the resources.

F.1.2 Rank biased overlap (RBO) (Webber et al., 2010). A score that can be broadly interpreted as a weighted correlation between rankings. We can specify how much we want to prioritize the top

of the ranking using parameter p that is between 0.0 and 1.0, with a smaller value giving more priority to the top of the ranking. The best value is 1.0, while it gives value of 0.0 for rankings that are completely the opposite. We compute the RBO value and then compare it to the selected threshold t, increasing the resources if the value is less than the threshold.

F.1.3 Reciprocal Rank Regret (RRR). A key insight is that configurations can be very similar to each other and differences in their rankings will not affect the quality of the found solution significantly. To account for this we look at the objective values of the configurations (e.g. accuracy) and compute the relative regret that we would pay at the current rung if we would have assumed the ranking at the previous rung correct.

We define reciprocal rank regret (RRR) as:

$$\text{RRR} = \sum_{i=0}^{n-1} \frac{(f_i - f_i')}{f_i} w^i,$$

where f represents the ordered scores in the top rung, f' represents the reordered scores from the top rung according to the second rung, n is the number of configurations in the top rung and p is the parameter that says how much attention to give to the top of the ranking. The weights w_i sum to 1 and can be selected in different ways to e.g. give more priority to the top of the ranking. For example, we could use the following weights:

$$w_i = \frac{p^i}{\sum_{i=0}^{n-1} p^i}$$

The metric has an intuitive interpretation: it is the average relative regret with priority on top of the ranking. The best value of RRR is 0.0, while the worst possible value is 1.0.

We also consider a version of RRR which considers the absolute values of the differences in the objectives - Absolute RRR (ARRR).

F.1.4 Soft ranking variations. We consider several variations of soft ranking. The first variation is to modify the value of the ϵ parameter. We have considered values 0.01, 0.02, 0.025, 0.03, 0.05. The second set of variations aim to estimate the value of ϵ automatically, using various heuristics. The heuristics we have evaluated include:

- Standard deviation: calculate the standard deviation of the considered performance measure (e.g. accuracy) of the configurations in the previous rung and set a multiple of it as the value of ϵ – we tried multiples of 1, 2 and 3.
- Mean distance: value of ϵ is set as the mean distance between the score of the configurations in the previous rung.
- Median distance: similar to the mean distance, but using the median distance.

We have evaluated these additional ranking functions using NASBench201 benchmark.

F.2 Results

We report the results in Table 7, 8 and 9. We see there are also several other variations that achieve strong results across a variety of datasets within NASBench201, most notably soft ranking 2σ and variations based on RRR. In these cases we obtain similar performance as ASHA, but at a significantly shorter time. We also see that simple ranking is not sufficiently robust – some benevolence is needed. Due to its simplicity and strong performance, we have selected soft ranking with fixed value of ϵ as our main ranking function.

Table 7: NASBench201 – CIFAR-10 results for a variety of ranking functions.

Approach	Accuracy (%)	Runtime	Speed comparison	Max resources
ASHA	93.85 ± 0.25	3.0h ± 0.6h	1.0x	200.0 ± 0.0
PASHA direct ranking	93.79 ± 0.26	2.7h ± 0.6h	1.1x	198.4 ± 6.0
PASHA soft ranking $\epsilon = 0.01$	93.79 ± 0.26	2.6h ± 0.5h	1.1x	194.3 ± 21.2
PASHA soft ranking $\epsilon = 0.02$	93.78 ± 0.31	2.4h ± 0.5h	1.2x	152.4 ± 58.3
PASHA soft ranking $\epsilon = 0.025$	93.78 ± 0.31	2.3h ± 0.5h	1.3x	144.5 ± 59.4
PASHA soft ranking $\epsilon = 0.03$	93.78 ± 0.32	2.2h ± 0.6h	1.3x	128.6 ± 58.3
PASHA soft ranking $\epsilon = 0.05$	93.79 ± 0.49	1.8h ± 0.7h	1.6x	76.0 ± 66.0
PASHA soft ranking 1σ	93.75 ± 0.32	2.4h ± 0.5h	1.2x	186.4 ± 35.2
PASHA soft ranking 2σ	93.88 ± 0.28	1.9h ± 0.5h	1.5x	132.7 ± 68.7
PASHA soft ranking 3σ	93.56 ± 0.69	0.9h ± 0.3h	3.1x	16.2 ± 19.9
PASHA soft ranking mean distance	93.73 ± 0.52	2.3h ± 0.4h	1.3x	184.1 ± 40.5
PASHA soft ranking median distance	93.82 ± 0.26	2.3h ± 0.5h	1.3x	169.2 ± 51.2
PASHA RBO p=1.0, t=0.5	93.49 ± 0.78	0.7h ± 0.1h	4.2x	4.6 ± 6.0
PASHA RBO p=0.5, t=0.5	93.77 ± 0.35	2.2h ± 0.6h	1.3x	144.0 ± 71.2
PASHA RRR p=1.0, t=0.05	93.49 ± 0.78	0.7h ± 0.0h	4.4x	3.0 ± 0.0
PASHA RRR p=0.5, t=0.05	93.76 ± 0.31	2.1h ± 0.6h	1.4x	140.9 ± 69.7
PASHA ARRR p=1.0, t=0.05	93.71 ± 0.35	2.4h ± 0.4h	1.2x	179.0 ± 42.9
PASHA ARRR p=0.5, t=0.05	93.81 ± 0.30	2.5h ± 0.4h	1.2x	181.0 ± 40.9
One epoch baseline	93.30 ± 0.61	0.3h ± 0.0h	8.5x	1.0 ± 0.0
Random baseline	72.93 ± 19.55	0.0h ± 0.0h	–	0.0 ± 0.0

Table 8: NASBench201 – CIFAR-100 results for a variety of ranking functions.

Approach	Accuracy (%)	Runtime (s)	Speed comparison	Max resources
ASHA	71.69 ± 1.05	3.2h ± 0.9h	1.0x	200.0 ± 0.0
PASHA direct ranking	71.69 ± 1.05	2.8h ± 0.7h	1.1x	200.0 ± 0.0
PASHA soft ranking $\epsilon = 0.01$	71.55 ± 1.04	2.5h ± 0.7h	1.3x	198.3 ± 6.5
PASHA soft ranking $\epsilon = 0.02$	70.94 ± 0.85	2.0h ± 0.5h	1.6x	160.5 ± 62.9
PASHA soft ranking $\epsilon = 0.025$	71.41 ± 1.15	1.5h ± 0.7h	2.1x	88.3 ± 74.4
PASHA soft ranking $\epsilon = 0.03$	71.00 ± 1.38	1.0h ± 0.5h	3.2x	39.4 ± 63.4
PASHA soft ranking $\epsilon = 0.05$	70.71 ± 1.66	0.7h ± 0.0h	4.9x	3.0 ± 0.0
PASHA soft ranking 1σ	71.56 ± 1.03	2.5h ± 0.6h	1.3x	184.1 ± 40.5
PASHA soft ranking 2σ	71.14 ± 0.97	1.9h ± 0.7h	1.7x	136.4 ± 75.8
PASHA soft ranking 3σ	71.63 ± 1.60	1.0h ± 0.3h	3.3x	20.2 ± 25.3
PASHA soft ranking mean distance	71.51 ± 0.99	2.4h ± 0.5h	1.4x	189.8 ± 30.3
PASHA soft ranking median distance	71.52 ± 0.98	2.4h ± 0.6h	1.3x	189.5 ± 30.6
PASHA RBO p=1.0, t=0.5	70.69 ± 1.67	0.7h ± 0.1h	4.6x	3.8 ± 2.0
PASHA RBO p=0.5, t=0.5	71.51 ± 0.93	2.4h ± 0.7h	1.3x	180.5 ± 50.6
PASHA RRR p=1.0, t=0.05	70.71 ± 1.66	0.7h ± 0.0h	4.9x	3.0 ± 0.0
PASHA RRR p=0.5, t=0.05	71.42 ± 1.51	1.2h ± 0.5h	2.6x	39.3 ± 51.4
PASHA ARRR p=1.0, t=0.05	70.80 ± 1.70	0.8h ± 0.4h	3.8x	22.9 ± 51.3
PASHA ARRR p=0.5, t=0.05	71.41 ± 1.05	1.8h ± 0.6h	1.7x	110.0 ± 68.7
One epoch baseline	65.57 ± 5.53	0.3h ± 0.0h	9.2x	1.0 ± 0.0
Random baseline	42.98 ± 18.34	0.0h ± 0.0h	–	0.0 ± 0.0

G NAS experiments with various reduction factors η

An important parameter for the performance of multi-fidelity algorithms like ASHA is the reduction factor used in the algorithm. This hyperparameter controls the fraction of pruned candidates at

Table 9: NASBench201 – ImageNet16-120 results for a variety of ranking functions.

Approach	Accuracy (%)	Runtime (s)	Speedup factor	Max resources
ASHA	45.63 ± 0.81	8.8h ± 2.2h	1.0x	200.0 ± 0.0
PASHA direct ranking	45.63 ± 0.81	8.3h ± 2.5h	1.1x	200.0 ± 0.0
PASHA soft ranking $\epsilon = 0.01$	45.52 ± 0.89	7.0h ± 1.5h	1.3x	185.7 ± 36.1
PASHA soft ranking $\epsilon = 0.02$	45.79 ± 1.16	4.4h ± 1.4h	2.0x	71.4 ± 50.8
PASHA soft ranking $\epsilon = 0.025$	46.01 ± 1.00	3.2h ± 1.0h	2.8x	28.6 ± 27.7
PASHA soft ranking $\epsilon = 0.03$	45.62 ± 1.48	2.4h ± 0.7h	3.6x	11.0 ± 10.0
PASHA soft ranking $\epsilon = 0.05$	44.90 ± 1.42	1.8h ± 0.0h	5.0x	3.0 ± 0.0
PASHA soft ranking 1σ	45.63 ± 0.89	6.5h ± 1.3h	1.4x	177.1 ± 44.2
PASHA soft ranking 2σ	45.39 ± 1.22	4.5h ± 1.4h	1.9x	91.2 ± 58.0
PASHA soft ranking 3σ	44.90 ± 1.42	1.8h ± 0.0h	5.0x	3.0 ± 0.0
PASHA soft ranking mean distance	45.50 ± 1.12	6.2h ± 1.5h	1.4x	157.7 ± 54.7
PASHA soft ranking median distance	45.67 ± 0.95	6.3h ± 1.6h	1.4x	156.3 ± 52.2
PASHA RBO p=1.0, t=0.5	44.90 ± 1.42	1.8h ± 0.0h	5.0x	3.0 ± 0.0
PASHA RBO p=0.5, t=0.5	45.24 ± 1.13	6.4h ± 1.3h	1.4x	148.3 ± 56.9
PASHA RRR p=1.0, t=0.05	44.90 ± 1.42	1.8h ± 0.0h	5.0x	3.0 ± 0.0
PASHA RRR p=0.5, t=0.05	44.90 ± 1.42	1.8h ± 0.0h	5.0x	3.0 ± 0.0
PASHA ARRR p=1.0, t=0.05	44.90 ± 1.42	1.8h ± 0.0h	5.0x	3.0 ± 0.0
PASHA ARRR p=0.5, t=0.05	44.90 ± 1.42	1.8h ± 0.0h	5.0x	3.0 ± 0.0
One epoch baseline	41.42 ± 4.98	1.0h ± 0.0h	8.8x	1.0 ± 0.0
Random baseline	20.97 ± 10.01	0.0h ± 0.0h	–	0.0 ± 0.0

every rung. The optimal theoretical value is e and the most common values used in practice are probably 2 and 3. The results in Table 10 show that the gains provided by PASHA are consistent also for $\eta = 2$ and $\eta = 4$.

Table 10: NASBench201 results with various reduction factors η.

Dataset	Reduction factor	Approach	Accuracy (%)	Runtime	Speedup factor	Max resources
CIFAR-10	$\eta = 2$	ASHA	93.88 ± 0.27	3.6h ± 1.1h	1.0x	200.0 ± 0.0
		PASHA	93.77 ± 0.56	2.6h ± 0.6h	1.4x	134.9 ± 52.7
	$\eta = 3$	ASHA	93.85 ± 0.25	3.0h ± 0.6h	1.0x	200.0 ± 0.0
		PASHA	93.78 ± 0.31	2.3h ± 0.5h	1.3x	144.5 ± 59.4
	$\eta = 4$	ASHA	93.75 ± 0.28	2.4h ± 0.6h	1.0x	200.0 ± 0.0
		PASHA	93.72 ± 0.30	2.1h ± 0.5h	1.2x	154.7 ± 64.1
CIFAR-100	$\eta = 2$	ASHA	71.67 ± 0.84	3.8h ± 1.0h	1.0x	200.0 ± 0.0
		PASHA	71.79 ± 1.38	2.1h ± 0.7h	1.8x	101.5 ± 55.0
	$\eta = 3$	ASHA	71.69 ± 1.05	3.2h ± 0.9h	1.0x	200.0 ± 0.0
		PASHA	71.41 ± 1.15	1.5h ± 0.7h	2.1x	88.3 ± 74.4
	$\eta = 4$	ASHA	71.43 ± 1.13	2.7h ± 0.9h	1.0x	200.0 ± 0.0
		PASHA	71.76 ± 1.20	1.1h ± 0.5h	2.4x	59.2 ± 74.8
ImageNet16-120	$\eta = 2$	ASHA	46.09 ± 0.68	11.9h ± 4.0h	1.0x	200.0 ± 0.0
		PASHA	45.72 ± 1.36	4.2h ± 1.7h	2.8x	49.5 ± 44.2
	$\eta = 3$	ASHA	45.63 ± 0.81	8.8h ± 2.2h	1.0x	200.0 ± 0.0
		PASHA	46.01 ± 1.00	3.2h ± 1.0h	2.8x	28.6 ± 27.7
	$\eta = 4$	ASHA	45.43 ± 0.98	7.9h ± 3.0h	1.0x	200.0 ± 0.0
		PASHA	45.48 ± 1.36	3.2h ± 1.7h	2.4x	40.3 ± 49.8

H NAS experiments with Bayesian Optimization

Bayesian Optimization combined with multi-fidelity methods such as Successive Halving can improve the predictive performance of the final model (Klein et al., 2020). In this set of experiments, we verify PASHA can speedup also these kinds of methods. Our results are reported in Table 11, where we can clearly see PASHA obtains a similar accuracy result as ASHA with significant speedup.

Table 11: NASBench201 results for ASHA with Bayesian Optimization searcher – MOBSTER (Klein et al., 2020) and similarly extended version of PASHA. The results show PASHA can be successfully combined with a smarter configuration selection strategy.

Dataset	Approach	Accuracy (%)	Runtime	Speedup factor	Max resources
CIFAR-10	ASHA BO	94.10 ± 0.22	5.0h ± 1.3h	1.0x	200.0 ± 0.0
	PASHA BO	94.17 ± 0.17	4.4h ± 1.7h	1.1x	156.7 ± 62.4
CIFAR-100	ASHA BO	72.76 ± 0.64	5.6h ± 2.0h	1.0x	200.0 ± 0.0
	PASHA BO	72.07 ± 1.80	3.8h ± 1.7h	1.5x	157.9 ± 72.7
ImageNet16-120	ASHA BO	45.79 ± 1.18	13.8h ± 5.0h	1.0x	200.0 ± 0.0
	PASHA BO	45.02 ± 1.15	6.2h ± 5.8h	2.2x	50.0 ± 75.4

I Additional information

I.1 Compute

NASBench201 experiments included in the Jupyter notebook can run on a laptop in about 1.5 hour – excluding the Bayesian Optimization experiments. NASBench201 experiments that use BO benefit from using a cluster with multiple CPUs – we have used our own internal cluster with CPUs. We have allocated 4 CPUs to each experiment. BO experiments take at least a few minutes to run and parallelization across many nodes can significantly speedup the time it takes to complete the multiple runs. For the HPO experiments we have utilized Amazon SageMaker (AWS cloud provider), selecting instances with multiple CPUs (no GPUs were used).

I.2 Hyperparameter tuning

Most of the hyperparameters were selected by following the choices of hyperparameters in the literature and also by using the default hyperparameters within SyneTune library. For our HPO experiments we observed e.g. how many epochs are needed to obtain convergence and selected the value accordingly. For our PASHA algorithm the main hyperparameter is the choice of the ranking function and its details, and we have evaluated various choices as part of the ablation study (trying out a few reasonable hyperparameters for these ranking functions). These experiments required only a small amount of compute as we used a NASBench201 simulator backend available in SyneTune.

For our larger-scale HPO experiments we did not perform extensive hyperparameter tuning and selected options which worked well on NASBench201. When deciding on how to specify the minimum resources, we used our judgement to decide what could be reasonable values.

I.3 Licensing

Information about licensing:

- SyneTune `https://github.com/awslabs/syne-tune`: Apache-2.0 license (our code is released under the same license)
- NASBench201 (Dong and Yang, 2020): MIT license

- Codes dataset (Zavadskyy, 2017): CC BY-SA 4.0 license
- LendingC dataset (George, 2019): CC0: Public Domain
- CoverType dataset (Blackard et al., 1998): unspecified, but the dataset has been donated for public use