"Mini-Benchmarking" Approach to Optimize Evolutionary Methods of Neural Architecture Search

Kamil Khamitov(✉) and Nina Popova

Lomonosov Moscow State University, Moscow, Russia
popova@cs.msu.ru

Abstract. Due to the rapid development of Artificial Neural Networks (ANN) models, the number of hyperparameters constantly grows. With such a number of parameters, it's necessary to use automatic tools for building or adapting new models for new problems. It leads to the expansion of Neural Architecture Search (NAS) methods usage, which performs hyperparameters optimisation in a vast space of model hyperparameters, so-called hyperparameters tuning. Since modern NAS techniques are widely used to optimise models in different areas or combine many models from previous experiences, it requires a lot of computational power to perform specific hyperparameters optimisation routines. Despite the highly parallel nature of many NAS methods, they still need a lot of computational time to converge and reuse information from the generations of previously synthesised models. Therefore it creates demands for parallel implementations to be available in different cluster configurations and utilise as many nodes as possible with high scalability. However, simple approaches when the NAS solving is performed without considering results from the previous launches lead to inefficient cluster utilisation. In this article, we introduce a new approach of optimisation NAS processes, limiting the search space to reduce the number of search parameters and dimensions of the search space, using information from the previous NAS launches that allow decreasing demands of computational power and improve cluster utilisation as well.

Keywords: Hyperparameters tuning · NNI · HPC · Neural Architecture Search

1 Introduction

Recent studies demonstrate a significant increase of ANN's hyperparameters of models currently used in production. The number of BERT's [4] parameters exceeds 110M. It means that even if we use computing nodes with modern Tesla GPU's (V100/A100), it is still not enough for this amount of parameters neither inference nor training. Nowadays, many researchers utilize already pre-trained ANN models as a starting point, since it allows to decrease time to

© Springer Nature Switzerland AG 2021
V. Voevodin and S. Sobolev (Eds.): RuSCDays 2021, CCIS 1510, pp. 352–364, 2021.
https://doi.org/10.1007/978-3-030-92864-3_27

production and many analytical experiments. But in the work [7] it was demonstrated that new fully-synthesized architecture may beat existing well-known analytically-created models for certain areas of tasks. In [3] authors mentioned other approaches which allow them to increase speedup convergence of such methods, but with the certain amounts of assumptions, that limits parallel efficiency. It leads us to the question about Neural Architecture Search and, in general, hyperparameters tuning. Such a task can be formulated performing optimization in vast search space of myriads of parameters, but since ANNs is not a simple graph with certain nodes, such specific of HPO optimization (that resulting graph of optimization should be a valid ANN) should be taken into consideration. It means that we require a particular set of large-scale optimization problem with certain constraints: the NAS (Neural Architecture search) problem and refining particular network topology problem. This search methodology and connecting nodes in graphs oblige us to use algorithms with prebuilt constraints to limit search space [7]. But such generic approaches can utilize a lot of computational power, although can demonstrate remarkable results [13]. In production, the range of optimizing models (during the iterative process of refining) usually are not so broad, so it leads us to the question: How we can use the information of the previously optimized generations to improve Neural Architecture Search convergence and computation power utilization?

2 Neural Architecture Search Problems

In this article we mainly focus on two formulated neural architecture search problems:

- refining existing topology by applying to new particular task,
- synthesizing new topology from scratch (Neural Architecture search).

Because of the increased resource utilization of modern DNNs, topology adaptation of neural networks has become a significant problem. In this case, the process looks like the best model development for the particular task that can be used to tune different sets of hyperparameters and then build a "distilled" model that fits the resource limitations on the particular device. In this article, we want to cover both steps of a process. The implementation of hyperparameters tuning requires a lot of computational resources, and it's significant to provide a possibility to perform such tuning on HPC clusters.

Adaptation Problem. The adaptation problem is considered as refining the existing model $min_\theta L(\theta_n), \theta_n = X(\theta_{n-1}, ...\theta_{n-k})$, where L – loss function, θ – hyperparameters set, θ_0 – initial hyperparameters (initial model) that are used as a core of method, X – the iterative process of new model building, based on previous hyperparameters.

Neural Architecture Search Definition. In the neural architecture search problem we don't have the initial value of hyperparameters. It limits the capabilities of methods that rely on the quality of the initial approximation. The formal process can be described as follows: $L(\theta_n), \theta_n = X(\theta_{n-1}, ...\theta_{n-k}),\ \theta_0 = \mathbf{0}$.

Distributed Hyperparameters Optimization. Since both problems that we regard in this article are large-scale hyperparameters optimization problems that typically utilize a lot of computational resources. The usage of distributed technologies and HPC is essential to carry on such type of problems in a meaningful time. Moreover, since CoDeepNEAT utilizes evolution-based techniques, it has a natural fit for parallel computations. With other methods, even which doesn't imply parallel implementation, benefits of distributed optimization can be obtained by running different tuners at the same time. The system we've chosen is NNI (Neural Network Intelligence) [6], because it provides a tunable interface that allows easily integrate bindings to different HPC schedulers in rather broad cluster configurations. Some notes about SLURM integration to NNI were presented in [13].

3 Techniques for Tuning Hyperparameters

CoDeepNEAT. The CoDeepNEAT (CoEvolution DeepNEAT) [7] is a rather popular NAS method based on utilizing ideas of evolution methods to the NAS problems. In general, CoDeepNEAT employs the concept of cooperative evolution of modules and the blueprints and use the idea of representing and encoding a group of layers as a subject for evolution, proposed in the DeepNEAT [7]. In general, CoDeepNEAT operates two populations: the modules population that encodes the set of layers or parts of the network (like LSTM cells, GRU cells, etc.), and the population of blueprints, which describes how modules of each should be connected to form a fully-functional network. Of course, blueprints don't encode the particular blocks, but the type-identification (evolution level) of the block. So, the blueprint is a network graph where the vertices correspond to modules and edges to connections between them. But due to the mutations modules on each side of the edge may have different dimensions, CoDeepNEAT handles it in a semi-automatic manner and allows the user to either use general upsampling/downsampling strategies or provides custom once. During the fitness step of the evolution, the ANN built from modules according to the particular blueprint. Then it performs training on the provided dataset. Still, with a small number of epochs, the fitness function's value is distributed in the population of modules and templates as an average among all networks containing this module or constructed using this blueprint. Since evolution-based techniques are respectively widely used for general optimization problems and have possibilities of parallel implementations, the performance analysis of CoDeepNEAT implementation and its comparison to other NAS methods of different types were made in [12]. It proves a high degree of parallelism and beneficial usage of NVLink technologies when the large NAS-tuning tasks like data-labelling with LSTM, building RNN blocks from scratch and impact of different types of interconnecting GPU-GPU utilization multi-GPU clusters.

The CoDeepNEAT evolution scheme is demonstrated in Fig. 1. CoDeepNEAT proves its convergence rate and applicability compared to its predecessors since this method has more possibilities for mutations. The most negligible mutations

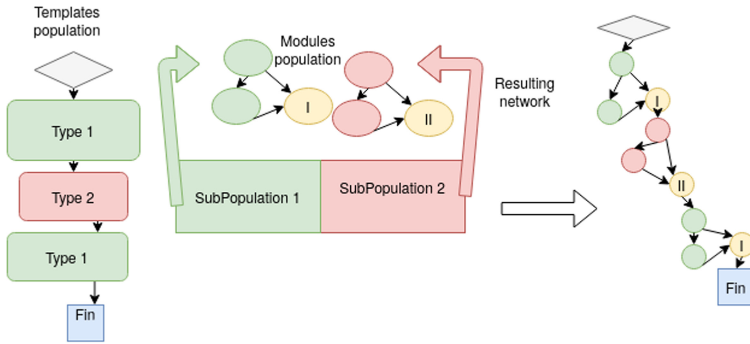

Fig. 1. Evolution scheme of CoDeepNEAT [7]

among the population of templates or modules lead to a significant change in final networks.

4 NAS Search Space Optimization

Rather different approaches can be used for optimizing the efficiency of evolutionary algorithms, like using different types of mutations per each set, using egalitism policies (transferring best items to the next generations). In generic NAS trainers like CoDeepNEAT, one of the main points of the optimization is the search space and its limitations (e.g. it's not the best variant to use many GRU parts before the feature extraction CNN parts even in reinforcement learning tasks). So one of the root causes is to use the information of the previous tasks launches. Such optimizations part considered as the crucial part of the system described in [13]. Since we've observed the difference in performance between launches in adaptation mode and NAS mode, we've performed tests in this article separately; however, we can't reuse the observations that correspond to the different problem kind (Adaptation or FullNAS).

4.1 "Mini-Benchmarking" Approach

The key idea of the "mini-benchmarking" approach is to create some divided clusters of networks where some pre-optimised (either with MorphNET or with CDN-tuned) ANN topologies, where the block constraints are obtained as the blocks used in the l recent generations. The k different classes corresponding to the other problem types are tuned with CDN/MoprhNET, and topologies are stored in the off-line section. In the initial trial, where the model is optimised, the corresponding class is picked with the Random Forest method. Classification is performed by analysing the input model as a graph of blocks and particular features. As the base of such benchmarks set, we decided to use as base two most popular and up-to-date benchmarks in NAS: NAS-HPO-BENCH [15] and NAS-BENCH-201 [14] and add its semi-class tasks from [13] like HRM prediction and TinyImageNET.

How to Apply Information in Fine-Tuning Adaptation. The key idea of our method is to use a similar encoding as in DeepNEAT [7]. We encode the input model as the graph with nodes corresponds to a block of layers with some predefined max folding size. Then we use such a set of input layers and use the similarity approach from CoDeepNEAT to compare blocks and the order of such folded layers to the original encoding and provide such certain block information to the later phases.

After determining the number of the closest classes, we can obtain the initial distributions of both blueprints and modules. With such encoding, it's possible to use CoDeepNEAT decomposing procedure to set up both initial populations and perform a co-evolution step from the closest first approximation. The limitations of the possible modules can be defined as follows:

– if we peek class t we peek all l initial distributions as the first steps with the initial model.
– All footprints from the last t have been picked from.
– If the $l * generation_size > current_trialgenerationsize$, the first selection is performed in the bipartite tournament manner.

If the number of classes is much larger than the t, the possible search space can be decreased on module steps up to $1 - \frac{Numclasses}{t}$.

How to Apply Information in NAS. In the NAS tasks, where the user doesn't provide the particular input model, the user can peek at the mix of the already presented classes to obtain a similar speedup as in fine-tuning adaptation.

Building the "Benchmarking" Databases. A lot of tools that provide different levels of NAS benchmarking approaches have been established recently. Of course, different tools and benchmarks have different purposes. In this paper, we are taking into consideration only a few of them that have the following properties:

– Has general applicability and has capability of testing modern NAS methods with modern ANN-based datasets.
– Has ability to reflects certain difference of hyperparameters tuning specific of methods.
– Has wide acceptance in NAS community and have ability of automatic integration.

To fit these criteria, we've picked two major benchmarking datasets: NAS-HPO-BENCH [15] and NAS-BENCH-201 [14]. In NAS-HPO-BENCH authors proposed an approach for building a small benchmarking suite that can be performed in a relatively small amount of time. It consists of 4 suites (Protein, Naval, Slice and Parkinson) and about 150 tasks correspondingly. However, this suite only covers a limited amount of ANNs' areas of applicability and mainly focuses on the precision of hyperparameters optimization with limited discrete areas of certain hyperparameters. The authors used the fANOVA method [16] to analyze the importance of specific hyperparameters across the whole models in different parts of the benchmarks suite. After all, the two problems of

this benchmark are narrow to the particular range of the tasks (medical image recognition). Another approach is described in [14]. The authors provided a quite different approach for building a dataset. They took only three novel tasks from the image-recognition area: CIFAR-10, CIFAR-100, ImageNet-16-120. After all considerations, we have decided to use both approaches. We have built our benchmarking suite on top of the NAS-HPO-BENCH [15] with adding more info from NAS-BENCH-201 [14] and custom tasks for RNNs, including LSTM/GRUs and complex tasks as HRM data prediction.

Adding New Classes to the Benchmark DB. With our approach adding new benchmarks requires running the new set of experiments and obtaining the slice of the last l generations of certain tasks. Then new reclassification to get the new classification to be performed. It means that the peeking classifier method should be scalable for reclassification. However, since the number l and the number of classes is usually small, like in NAS-HPO-BENCH (144 classes), it shouldn't be considered as a serious problem.

5 Automatic System Architecture

The brief architecture of the NAS system that supports execution of tuning problems using HPC cluster was described in [13]. The following changes were applied to support this new approach in our system of hyperparameters tuning:

1. The new entity that is used as storage for modules and corresponding classes in each part of the "benchmarking suite".
2. Classifier that obtains certain set of available modules for the trials.

So now, the NAS or adaptation can be run in two different modes: utilizing the previous information from storage or producing new benchmarking data for it. Before NAS, we run our classifier and then pick the limited set of the available modules. So the classifier runs after the tuners and cluster configuration choose but before the actual optimization problem starts. Post-processing is unaffected for the tuners selection. Also, we supported additional workflow for integrating such data for model-dependent post-tuners. Such integration was described in [13].

6 Experiments

6.1 Experimental Setup and Configuration

Clusters Configuration. The following clusters and their configurations were used for the experimental study of the proposed approach (Fig. 2).

- 4 × nodes of Polus cluster with 2 × IBM POWER8 processors with up to 160 threads, 256 GB RAM and 2 × Nvidia Tesla P100 on each node [9].
- 33 nodes of Lomonosov-2 cluster with Intel Xeon E5-2697 v3 2.60 GHz with up to 28 threads, 64 GB RAM and Tesla K40s on each node [10].

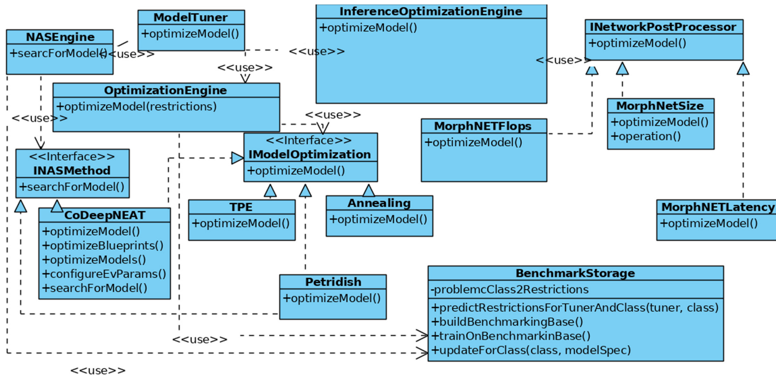

Fig. 2. Modules UML class diagram, that briefly describe system's architecture with Tuners [6]

Setting Up the Custom "Benchmarking DB" and It's Configuration. We've decided to build two sets of benchmarks to analyze scalability and further performance improvements of suggested Mini-benchmarking for limiting search space of HPO tuners. The first one is the modified versions of NAS-BENCH-HPO and NAS-BENCH-201. In both cases, we've added three different problems to cover RNNs, and semi CNN-RNN approaches with canonical solutions as well. As new problems, we've added:

– HRM data prediction on UBFC-RPPG [13]
– Tiny ImageNET
– minified CoLa dataset (sampled 10%).

Two modified datasets were divided into the two groups of "benchmarks":

– Small dataset, where the all resulted networks are CNN without recurrent layers (Fashion-MNIST, Tiny ImageNET, and CIFAR-100, minified CoLa).
– Large dataset where hybrid structures of ANN provide some benefits to convergence like: UBFC-RPPG, Protein, ImageNET-16-120, Parkinson.

Baseline Results. As baseline results we run only NAS problem on both large and small groups. Accuracy on each is presented in Table 1. Speedup of NAS task can be observed in Fig. 3. Of course, in some datasets (like CoLa and Protein, ImageNet-16-120), the provided accuracy is far from perfect but comparable to some simple implementations which can be analytically obtained like simple approaches from [15]. For some others like UBFC-RPPG and MNIST-like task, the provided accuracy is comparable to the well-known implementations [13].

The Parallel efficiency highly depends on the number of modules available for NAS and tuning correspondingly, such effect was also shown in [13], and it means the less number of different types of modules we use in problem is the more benefits from the parallel implementation we can obtain. It means we should either minimize extra-types of nodes for each problem with better

Table 1. CoDeepNEAT accuracy on different parts of the composed benchmarking data

Protein	UBFC-RPPG	Fashion-MNIST	Tiny ImageNET	CIFAR-100	Imagenet-16-120	CoLa
0.643	0.702	0.83	0.672	0.738	0.651	0.682

classification or decrease the difference between modules increasing the folding factor for analysis.

Convergence and Speedup in Small and Large Groups of "Benchmarks". For evaluation, we tried to analyze the difference between the convergence and speedup of using the "benchmarking" approach in the different groups and among the whole set of datasets to analyze which parts are more crucial or essential dependencies or blocks variations among each type of the problems.

Folding Factor Analysis. To analyze benefits, we vary folding factor between 1 and 10, and performed tests on data in a small group. The impact of such variation can be observed in Fig. 4. It's shown that folding factor 3 is optimal in many cases for the "small part" of our "benchmarking" data; increasing the folding factor to more than 7 leads to loss of accuracy in many parts of benchmarks like UBFC-RPPG and Protein. The importance of the folding factor is related to the particular classifier and how particularly thus encoding represents similarities for each node. For CoDeepNEAT, it means we can utilize the same encoding that we use in modules population [7] and such layout can be reformulated with the same folding factor for a group of layers or even connected noes of the modules population or not.

Convergence Analysis. We analysed average accuracy and loss per generation in NAS and HPO-tuning tasks. Graphs can be found in Fig. 5. In each of the experiments, we also close two or more class tasks to measure how accurate the generalised method can found networks from the unknown tasks. Accuracy on this validation dataset is presented in Fig. 6, except for Tiny ImageNET/ImageNET-16-120 because of similarities in group encodings. So as we can see, in general, CoDeepNEAT, even in such unknown tasks, can rely on the similarities that he can observe in data. So if the tasks are pretty unknown (like UBFC-RPPG or Protein), it can recreate carefully only partial networks with worse accuracy. The average population accuracy among first 50 epochs is presented in Fig. 7.

Parallel Efficiency. Parallel experiments have been performed on 32 nodes of the Lomonosov cluster for the Large dataset and on 4 nodes of Polus in Small datasets. Graphs for parallel efficiency (the ratio between observed speedup and the ideal one) after experimenting in both parts are presented in Fig. 8. Increasing of parallel efficiency during the NAS in both configurations demonstrates the how beneficial of decreasing search-space dimensions for large-scalable tasks can be done with such approach and it means that it's possible to reduce the required computational power for NAS and cluster utilization if you already have set of tasks that you've optimized before.

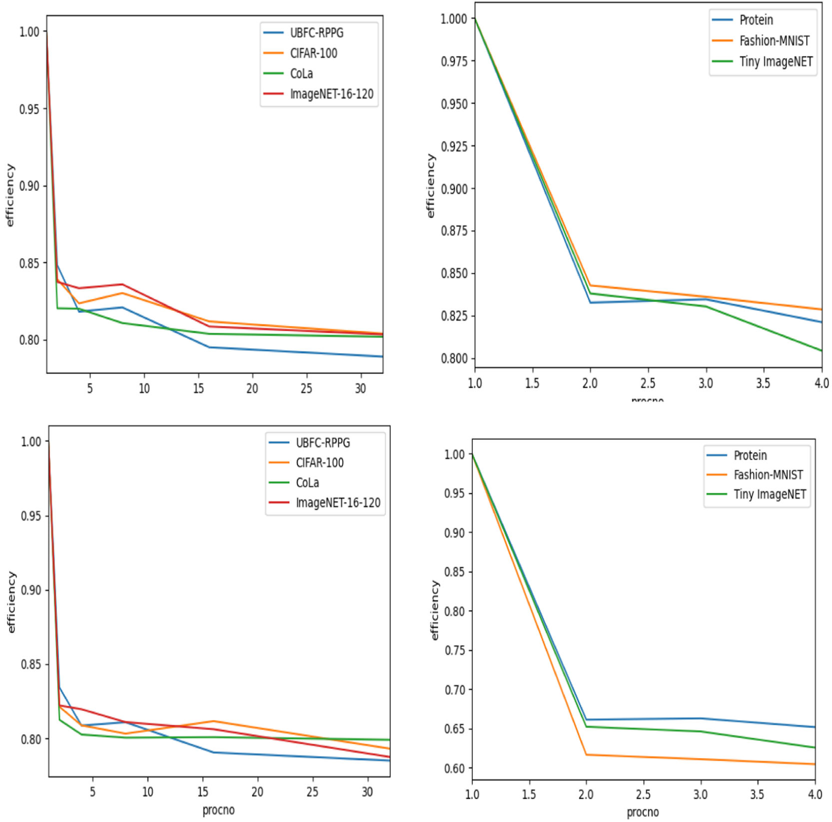

Fig. 3. Parallel efficiency of CoDeepNEAT on "large" and "small" parts of "benchmarking" data

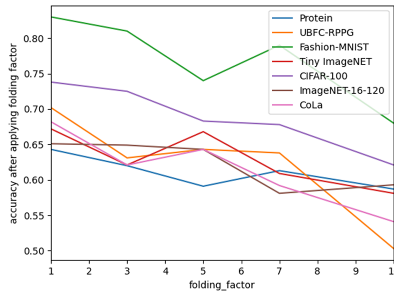

Fig. 4. Impact of folding factor to the accuracy of resulting networks and speedup in Adaptation and NAS

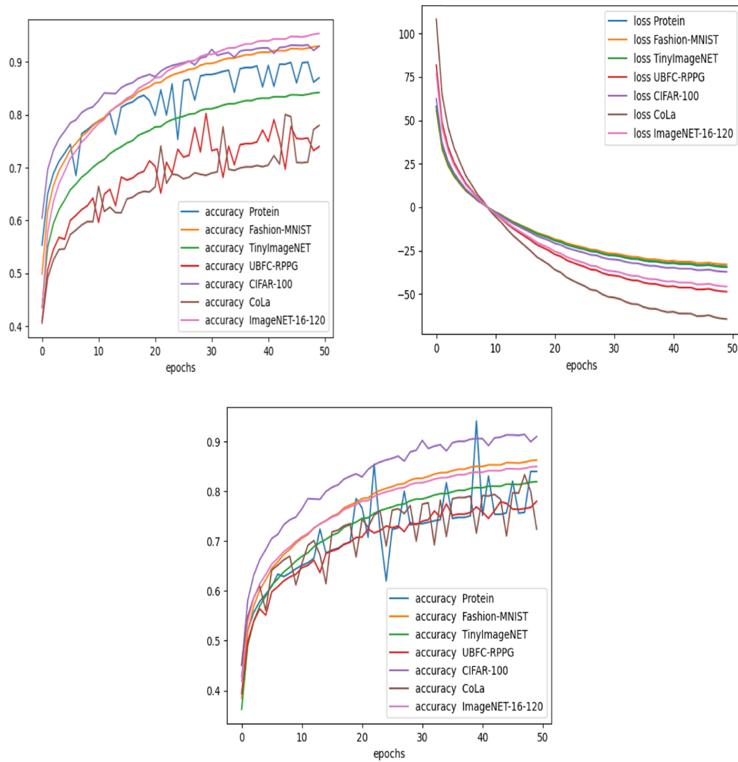

Fig. 5. Accuracy and loss of NAS and adaptation problems

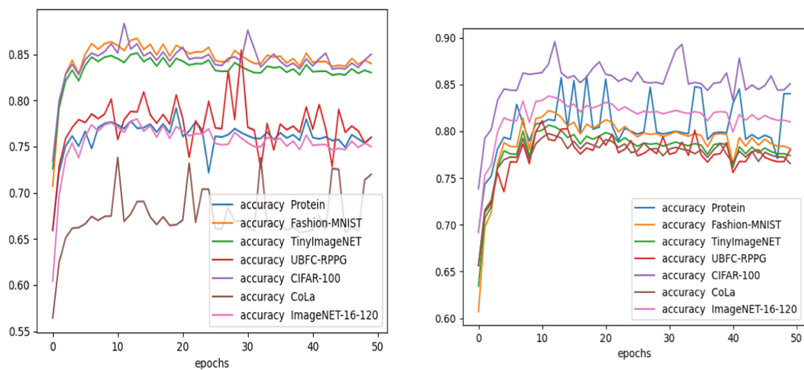

Fig. 6. Accuracy on validation in NAS (left) and adaptation parts

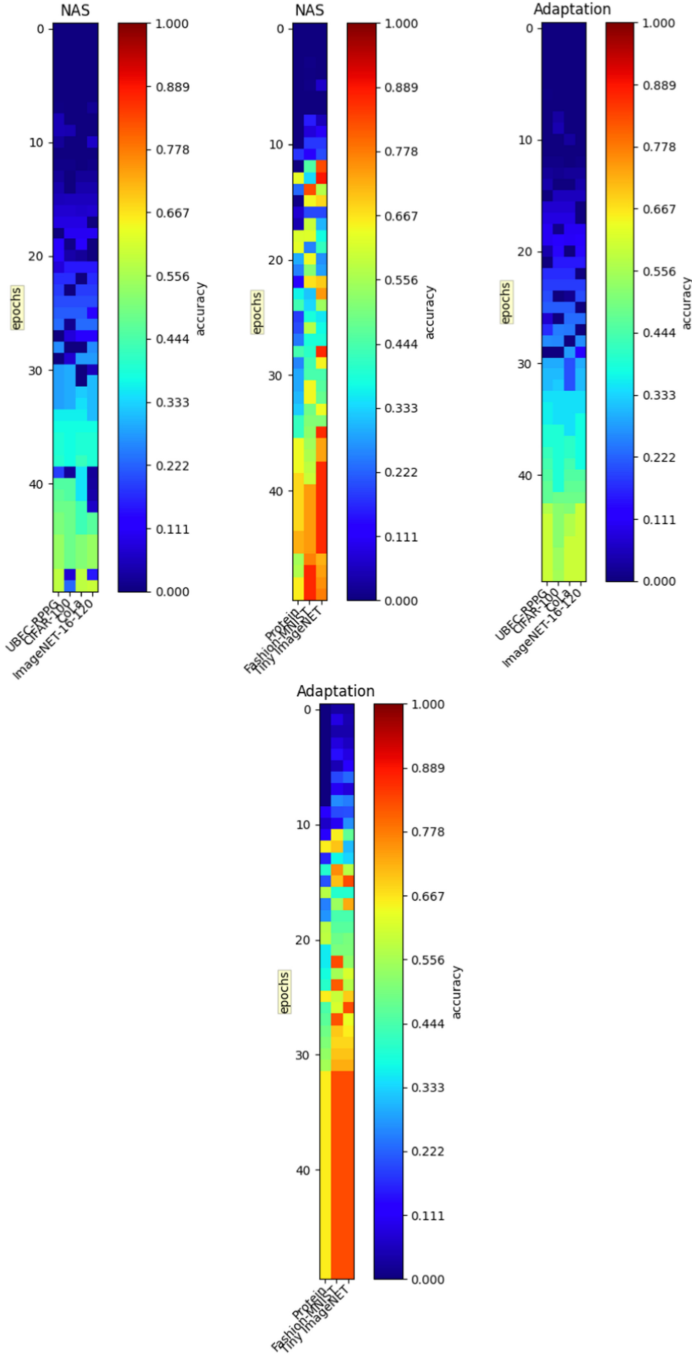

Fig. 7. Average accuracy of first 50 epochs in NAS and adaptation modes

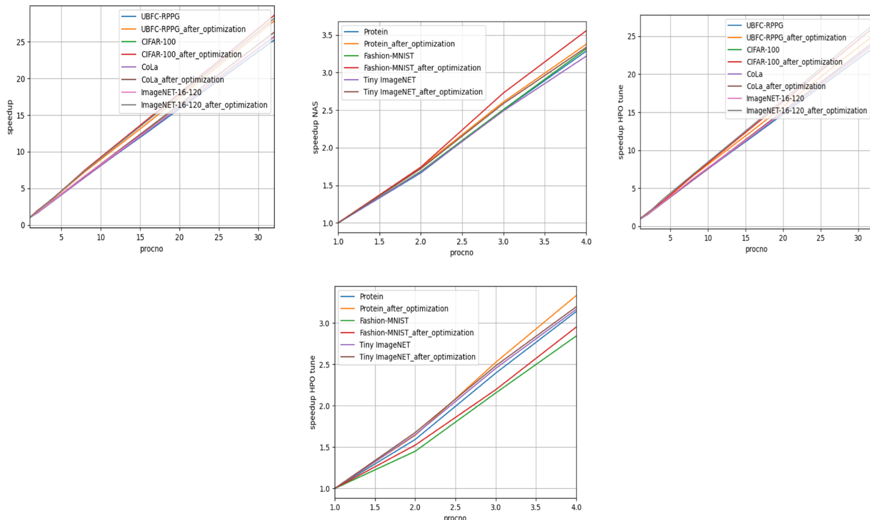

Fig. 8. Comparison of benefits on each speedup.

7 Conclusion

The approach of NAS methods optimization with the "mini-benchmarking" approach was tested in distributed clusters environment and integrated to the Automatic NAS-tuning system described in [13]. To analyze the approach, we built a custom benchmarking dataset that incorporates existing data from NAS-HPO-BENCH and NAS-BENCH-201. The boost of accuracy can be observed in both modes of hyperparameters tuning in all problem scales of benchmarking data. The amount of computational power reduced up to 7%. During NAS tasks, the observable changes may have a various impact, including the improving average accuracy and speedup on each part of the benchmarking dataset. The average speedup among each part is about 5% in NAS tasks and about 3.5% in adaptation tasks. The accuracy boost in NAS tasks and Adaptation tasks can reach even 10%, if the training problems that used for classifying complements each of the data, also in general, it's observed that in adaptation tasks, the difference in accuracy boost from the classification is up to 1.05 more than in NAS problems. It enables more efficient cluster utilization when similar NAS/adaptation tasks are processed, that demonstrated in Fig. 8. Also, It seems possible that using such an approach allows creating more sophisticated device-dependant tuners using collected information about the last generation.

Acknowledgements. Reported study was funded by RFBR according to research project №20-07-01053. The research is carried out using the equipment of the shared research facilities of HPC computing resources at Lomonosov Moscow State University.

References

1. Bergstra, J.S., et al.: Algorithms for hyper-parameter optimization. In: Advances in Neural Information Processing Systems (2011). https://doi.org/10.5555/2986459.2986743
2. Amado, N., Gama, J., Silva, F.: Parallel implementation of decision tree learning algorithms. In: Brazdil, P., Jorge, A. (eds.) EPIA 2001. LNCS (LNAI), vol. 2258, pp. 6–13. Springer, Heidelberg (2001). https://doi.org/10.1007/3-540-45329-6_4
3. Liu, H., Simonyan, K., Yang, Y.: DARTS: differentiable architecture search. arXiv preprint arXiv:1806.09055 (2018)
4. Devlin, J., et al.: BERT: pre-training of deep bidirectional transformers for language understanding. arXiv preprint arXiv:1810.04805 (2018)
5. Real, E., et al.: Large-scale evolution of image classifiers. In: Proceedings of the 34th International Conference on Machine Learning-Volume 70. JMLR.org (2017). https://doi.org/10.5555/3305890.3305981
6. Neural Network Intelligence, April 2020. https://github.com/microsoft/nni
7. Miikkulainen, R., Liang, J., Meyerson, E., et al.: Evolving deep neural networks. In: Artificial Intelligence in the Age of Neural Networks and Brain Computing. Elseiver, pp. 293–312 (2019). https://doi.org/10.1016/B978-0-12-815480-9.00015-3
8. CoDeepNEAT implementation base, April 2020. https://github.com/sash-a/CoDeepNEAT
9. Polus cluster specifications, April 2020. http://hpc.cs.msu.su/polus
10. Voevodin, Vl., et al.: Supercomputer Lomonosov-2: large scale, deep monitoring and fine analytics for the user community. J. Supercomput. Front. Innov. **6**(2), 4–11 (2019). https://doi.org/10.14529/jsfi190201
11. Bobbia, S., Macwan, R., Benezeth, Y., Mansouri, A., Dubois, J.: Unsupervised skin tissue segmentation for remote photoplethysmography. Pattern Recogn. Lett. (2017). https://doi.org/10.1016/j.patrec.2017.10.017
12. Khamitov, K., Popova, N.: Research of measurements of ANN hyperparameters tuning on HPC clusters with POWER8. In: Russian Supercomputing Days 2019: Proceedings of International Conference, Moscow, 23–24 September 2019, pp. 176–184 (2019)
13. Khamitov, K., Popova, N., Konkov, Y., Castillo, T.: Tuning ANNs hyperparameters and neural architecture search using HPC. Supercomputing, 536–548 (2020). https://doi.org/10.1007/978-3-030-64616-5_46
14. Dong, X., Yang, Y.: NAS-bench-201: extending the scope of reproducible neural architecture search. arXiv preprint arXiv:2001.00326 (2020)
15. Klein, A., Hutter, F.: Tabular benchmarks for joint architecture and hyperparameter optimization. arXiv preprint arXiv:1905.04970 (2019)
16. Hutter, F., Hoos, H., Leyton-Brown, K.: An efficient approach for assessing hyperparameter importance. In: International Conference on Machine Learning. PMLR (2014)