Regularization approach for accelerating Neural Architecture Search

Mikhail Nikulenkov[1], Kamil Khamitov[1], and assoc. prof. Nina Popova[2]

[1] Lomonosov Moscow State University, Moscow, Russia
✉michael@nikulenkov.com
[2] berserq0123@gmail.com
[3] popova@cs.msu.ru

Abstract. Modern Artificial Neural networks utilizes vast topologies to become applicable to the vast majority of the ML problems. Such topologies have an enormous number of parameters nowadays, which make it easier to use, but harder to train and modify. With such a number of parameters, the usage of semi-automatic tools for building or adapting new models for new problems is vital for the whole industry. It makes it essential to optimize such methods like Neural Architecture Search (NAS) to efficiently utilize computational resources of the GPU and whole cluster. Since modern NAS techniques are widely used to optimize models in different areas or combine many models from previous experiences, it requires a lot of computational power to perform certain hyperparameters optimization routines. Despite the highly parallel nature of many NAS methods, they still need a lot of computational power and time to converge. The key to increasing performance of many NAS methods is boosting performance of the training of the highly-depth and synthesized model, to evaluate the probe of the model and obtain the score for this epoch. It means that demands for parallel implementations to be available in different cluster configurations and utilize as many nodes as possible with high scalability. However, simple approaches when the NAS solving is performed without considering results from previous launches lead to inefficient utilization of computation power. In this article, we introduce a new method that can improve convergence in NEAT-based NAS method using L1/L2 regularization during the evolution step.

Keywords: Hyperparameters Tuning · NNI · HPC · Neural Architecture Search., LASSO

1 Introduction

In modern Artificial neural networks(ANN) applications, the significant increase in complexity of the models was demonstrated. In production we can observe ANN which number hyperparameters exceeded the 100M[1]. For large natural language processing(NLP) models, the number can even exceeds the 200M [1]. Which leads to the problem of hardware utilization. Even for single node unit with modern GPU's/TPU' the computation power is not enough for the effective training and inference. In most cases the pre-trained model is enough, but if you want to build something new from scratch you have to spend hours training it on the one node. Especially if you provide some custom layers or blocks to the ANN that correspond only to your specific domain problem. In the study [3] it was observed that new, fully automatically synthesized architecture may beat existing state-of-the art models in the some specific domain problems. The key problem with such an approach is how to reduce computation power to efficiently perform searches for this architecture with the highest speedup and a certain degree of parallelism to effectively reduce computation time. Some methods like DARTS [2] tries to increase level of parallelism using some assumptions on some certain properties of the source models, which limits applicability of such methods to the certain type of the models. It leads us to generalizations in Neural Architecture Search and so-called hyperparameters tuning. Such a task can be formulated by performing optimization in a vast search space of myriads of parameters. But since ANNs are not a simple graph with certain nodes, such specific Hyperparameters optimization(HPO) (that the 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 problems with certain constraints: the NAS (Neural Architecture search) problem and refining a particular network topology problem. This search methodology and connecting nodes in graphs oblige us to use algorithms with prebuilt constraints to limit search space [3]. But such generic approaches can utilize a lot of computational power, although can demonstrate remarkable results [4], [5]. Another approach is not to limit the search space to improve performance of the key step of the search method – trial on and test micro-batches of the test data, which costs a lot if you have different model candidates in such an approach. Some approaches to reduce such limits are to perform light-transforming of the model before moving to the next generation, but it requires some complicated work. Another approach to remove some connections and use techniques like dropout in order to keep models as smallest as possible. And some novel approach in this field is to use some kind of regularization, either L1 or L2, to restrain model and build some kind model-aggregate that resembles original model and it's behavior and than test it in the trial, which leads to reducing time on the step.

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 deep neural networks(DNN), 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 those which don'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. Integrating SLURM/LSF into this method can be done via plugins system of NNI and other appliances.

3 Techniques for Tuning Hyperparameters

CoDeepNEAT method CoDeepNEAT – co-Evolution version of DeepNEAT (Deep Neural Evolution of Augmenting Topologies) is a evolutionary method for tuning structure-based hyperparameters of ANN. Like the NEAT, this method utilizes an evolutionary approach to build from scratch or modify architectures. The main step of the methods consists of the evolution process in both modules, that represents the building blocks for the resulting networks and blueprints, that represents the graph of how modules should be connected with each other. On each step we build combinations of modules and blueprints; perform certain iterations of training, and then propagate back the score through both populations, and, according to the score, perform selection and other necessary things, important for the evolutionary process. During the step we may perform mutating of the module and blueprints independently which allows to perform it in a parallel manner. Specific mutations like change edges, add links

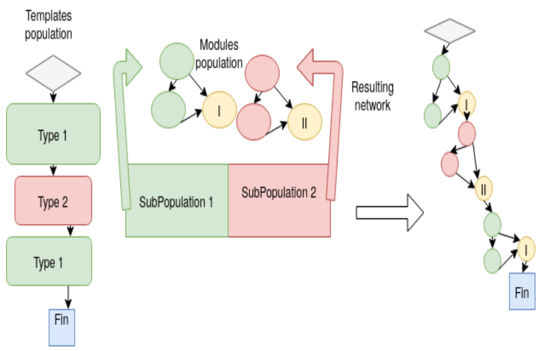

Fig. 1. Evolution scheme of CoDeepNEAT [3]

can be represented in both cases, but some mutations like linking blocks or merging recurrent connections can be block-specific. Before evaluation of the specific network the specific builder process [3] resolves differences in the connected models via inserting specific layers and provides resulting network without any weights. After that it trains the resulted network on a certain number of epochs to obtain the resulting accuracy and then uses its accuracy as measurement for the scoring elements of both populations. CoDeepNEAT proves its convergence rate and applicability compared to its predecessors, since this method has more possibilities for mutations. The most negligible mutations among the population of templates or modules led to a significant change in final networks.

The CoDeepNEAT evolution scheme is demonstrated in Fig. 1.

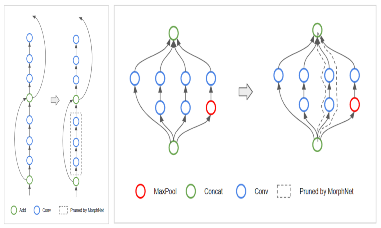

Fig. 2. MorphNet scheme for ResNet-style and Inception-style networks. [7]

4 Usage of regularization in HPO-tuning

In our approach we provide a method that eliminates some CoDeepNEAT drawbacks like high demand of computational resources in certain tasks and areas. Another approach, which utilize idea of limiting search space was described in [5]. Here we present the approach for building some smaller variants of networks from the original one during the initial step of the algorithm. The idea is to perform regularization process on each step of the CoDeepNEAT. The original idea belongs to MorphNET[7]. The method is based on tensors dimensions modification in some types of layers (mostly in convolutional ones) and on convolution sizes optimization to minimize the aimed regularizer loss. The main step can be considered as shrinking and sparsifying convolutions depend on regularizer values. Instead of trying numerous architectures across a large design space, MorphNet starts with an existing architecture for a similar problem and, in one shot, optimizes it for the task at hand, trying to zero some outputs of the layer using penalties. In the Fig. 2 the right side shows how residual connections are removed in ResNet-style and Inception-style networks. The key idea is to change a loss function with a modifying penalty. Penalty added to loss (with weight factor λ) takes the form of $\sum_{all_w} |cost_w * Y_w|$ where $cost_w$ is the cost in terms of the optimized metrics of the channel and Y_w is the scaling factor associated with the channel in the next batch normalization layer(prunning criterion). L1 regularization(LASSO) on weight matrices was introduced to handle problems that layers combined by skipping connections must have the same number of channels and uses it to reduce the number of nonzero weights without or with little effect on the performance (e.g. accuracy) of the deep neural network. Whereas proposed method mainly used for network distillation in this article, we consider it as the post-processing tool, which allows optimizing inference time without significant changes in the loss. Since the resulting network, if we add FLOPS regualizer requires much less power for inference time, we may significantly reduce the training time using the regualrized method for scoring the elements of population. So the step of the method can be described as follows:

- Obtain original resulting network
- Perform the L1 regularization process
- Train regularized model on the training set and propagate the score to the unmodified network

5 Automatic System Architecture

The brief architecture of the NAS system that supports execution of tuning problems using HPC cluster was described in [5]. It consists of four main parts

1. HPO-Tuners, or HPO-methods – kernels that runs on a host and tune the particular model, can be configured independently(CoDeepNEAT, PetriDish, TreeParzenEstimation, Naive Evolution etc)
2. Search space constraints part
   – Storage for modules classifications and corresponding classes in each part of the "benchmarking suite".
   – Classifier that obtains certain set of available modules for the trials.
3. Problems storage – predefined presets for the certain amount of problems, that allows limit search space. [5].

4. Post-processors – regularization methods that can be run on the final generation of population in order to change topology and fit certain constraints of L1/L2 regularization. (MorphNet, modified Morphnet)

The scheme of such system briefly described in Fig. 3. 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 that we obtain after problem reclassification process during mini-benchmark [5]. 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 [4]. Since system supports variety of limitations for search space and tuners that utilize such conditions for better convergence we allow user to tune hyperparameters in the semi-automatic mode with predefined sets of the search spaces or "detected" search space via classification model. Automatic usage of regularization in (Co)DeepNEAT method supported in the configuration files via option. And can be enabled by default parameter for certain classes of problems.

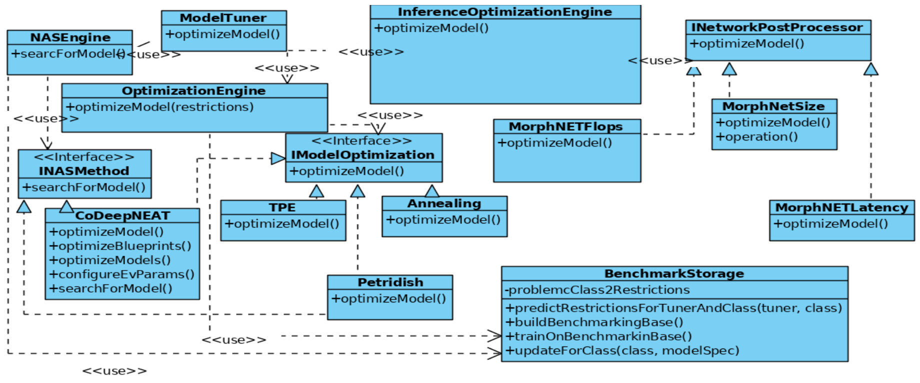

Fig. 3. Unified modelling language(UML) class diagram, that briefly describe system's architecture with Tuners [6].

6 Experiments

6.1 An approach to comparing Neuroevolution methods.

In order to minimize the impact of randomness, which is a built-in feature of evolution algorithms, each group of experiments uses the same pre-generated populations of modules, blueprints and individuals as a starting point of evolution. Given that the number of generations in evolution is fixed and the same in corresponding experiments, a comparison of the proposed approach with regular CoDeepNEAT can be done via analyzing the average accuracy of the last individual generation.

Two groups of the same dataset experiments are compared, the first one using the proposed approach and the second one using the regular CoDeepNEAT method. Populations of individuals belonging to the same group are merged into one mixed population; an average accuracy over the mixed population is calculated. An average time required for evolving a series of generations is calculated over each group of experiments.

Key values for estimating the performance of a proposed method are speedup and accuracy degradation. A speedup is calculated as $\frac{t_a}{t_b}$, where t_a is an average time over a group of CoDeepNEAT experiments and t_b is an average time over a group of experiments of proposed approach. An accuracy degradation is calculated as $acc_a - acc_b$, where acc_a is an average accuracy over a mixed population of CoDeepNEAT experiments and acc_b is an average mixed population accuracy over a group of experiments on proposed approach. A negative value for accuracy degradation means that the proposed method generated better results than the original CoDeepNEAT.

6.2 Experimental Setup and Configuration

Clusters configuration. The following cluster configuration was used for the experimental study of the proposed approach

- 2 nodes of Polus cluster with 2 x IBM POWER8 processors with up to 160 threads, 256GB RAM and 2 x Nvidia Tesla P100 [8].

Datasets. Following datasets were used:

- MNIST
- CIFAR-10
- CIFAR-100 (3 times augmented)
- Tiny ImageNet (100 classes, 4 times augmented)

Evolution configuration. MNIST and CIFAR-10 experiments were done over 7 generations, CIFAR-100 and Tiny ImageNet experiments were done over 4 generations. Each group consists of 5 experiments with a population of 12 individuals.

MorphNet configuration. One epoch of full training set shrinking of each individual was performed, with regularization coefficient set as $\frac{1}{init\_flopn}$, where $init\_flopn$ is a pre-shrinking number of individual's floating point operations required to calculate the result of the input data by the neural network (FLOPN stands for floating points operations number). The FLOPN values of the considered networks vary as $8 \cdot 10^6 - 10^9$. MorphNet alive threshold parameter is used for determining when an activation is alive: the smaller connection weight is considered dead; neurons with all-dead connections are considered redundant and are proposed for removal. This parameter was set to 0.1 for MNIST and CIFAR-10 and to 0.05 for CIFAR-100 and Tiny ImageNet.

Search space. Search space configuration was performed by hand with a trial-and-error method, given the authors' previous experience in solving adjacent problems. Search space for MNIST and CIFAR-10 is composed of: Conv2D layers with $16-68$ filters (kernel sizes vary as $[1,3,5]$) with possible adjacent MaxPooling2D (pool size is 2) or dropout layers (dropout rate varies as $0-0.5$); a pre-output Dense layer of size $32-256$ (with ReLu activation). Search space for CIFAR-100 and Tiny Imagenet is composed of: Conv2D layers with $16-128$ filters (kernel sizes vary as $[2,4,7,9,11,13]$) with possible adjacent MaxPooling2D (pool size is 2) or dropout layers (dropout rate varies as $0-0.5$); a pre-output two layer perceptron with Dense layers of size $32-128$ (with ReLu activation).

6.3 Experimental results

Average time and accuracy on each dataset are presented in figure 4: diagrams show compared time and accuracy of proposed method and original CoDeepNEAT over considered datatests. Time and accuracy of original CoDeepNEAT method can be compared with those of the proposed method: figure 4 shows that proposed method requires less computation time and shows comparable accuracy results.

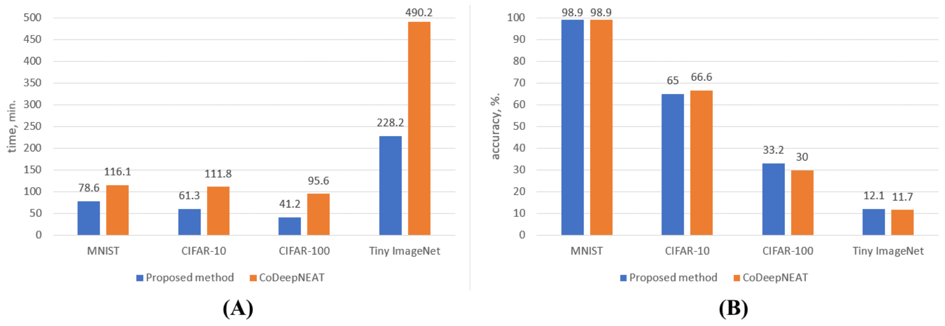

Fig. 4. (A) – time comparison on different datasets. (B) - accuracy comparison on different datasets

Speedup and accuracy degradation on each dataset are presented in table 1. Average speedup of the proposed method relative to CoDeepNEAT is more than 1.9, while accuracy degradation is negligible or even does not take place (in case of negative accuracy degradation values).

Training process of each dataset best architecture is presented in figures 5-8. Validation set loss and accuracy change as number of epochs grows is shown.

Table 1. Speedup and accuracy on different datasets.

	MNIST	CIFAR-10	CIFAR-100	Tiny ImageNet
speedup	1.48	1.82	2.31	2.14
acc. degradation, %	0.02	1.61	-3.16	-0.38

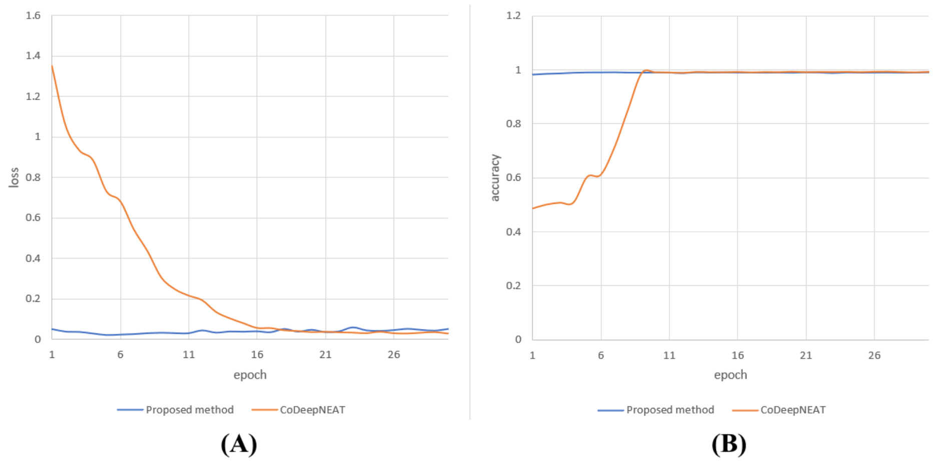

Fig. 5. MNIST training process. (A) – loss, (B) - accuracy.

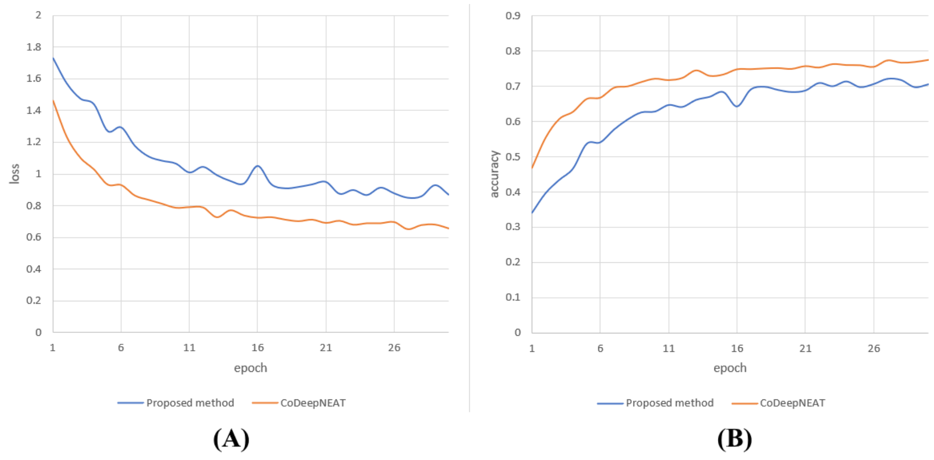

Fig. 6. CIFAR-10 training process. (A) – loss, (B) - accuracy.

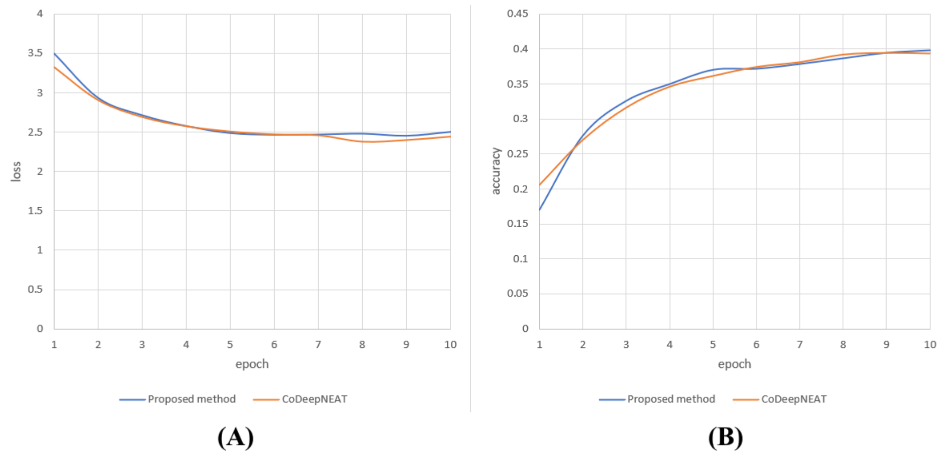

Fig. 7. CIFAR-100 training process. (A) – loss, (B) - accuracy.

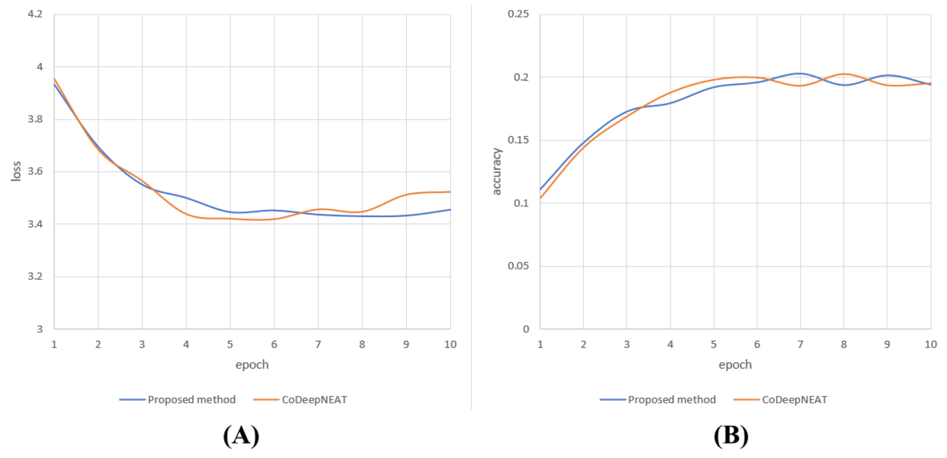

Fig. 8. Tiny ImageNet training process. (A) – loss, (B) - accuracy.

7 Conclusion

We developed and implemented a new type of optimization for Neuroevolution techniques that enables us to reduce the demands of computational power. The proposed technique utilizes L1 regularization for model simplification and distributing loss through this simplified model. In the terms of power reduction, it leads to the more than 1.9 speedup in average Tab. 1. In some complex image classification tasks it grows better than some small one like MNIST. Such an effect can be explained by different search spaces and complexity of the resulting networks (in MNIST/CIFAR-10 the resulting networks have much less graph nodes and types of nodes than in Tiny ImageNet). Such an approach was analyzed via parallel experiments on Polus clusters and in image classification problems we obtained not only negligible results in the terms of accuracy degradation, but also significant in the terms of FLOPS/iteration. At some points like in Tiny ImageNet and CIFAR-100, the overall accuracy degradation is negative it means the resulting networks is better after augmentation than original one.

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. Devlin, Jacob, et al. ”Bert: Pre-training of deep bidirectional transformers for language understanding.” arXiv preprint arXiv:1810.04805 (2018).
2. Liu, Hanxiao, Karen Simonyan, and Yiming Yang. ”Darts: Differentiable architecture search.” arXiv preprint arXiv:1806.09055 (2018).
3. Miikkulainen R., Liang J., Meyerson E., et al. Evolving deep neural networks // Artificial Intelligence in the Age of Neural Networks and Brain Computing. Elseiver 2019. P. 293–312.https://doi.org/10.1016/B978-0-12-815480-9.00015-3
4. Khamitov, Kamil, et al. ”Tuning ANNs Hyperparameters and Neural Architecture Search Using HPC.” Russian Supercomputing Days. Springer, Cham, 2020. https://doi.org/10.1007/978-3-030-64616-5.
5. Khamitov, K., Popova, N. (2021). “Mini-Benchmarking” Approach to Optimize Evolutionary Methods of Neural Architecture Search. In: Voevodin, V., Sobolev, S. (eds) Supercomputing. RuSCDays 2021. Communications in Computer and Information Science, vol 1510. Springer, Cham. https://doi.org/10.1007/978-3-030-92864-3_27
6. Neural Network Intelligence https://github.com/microsoft/nni April 2020
7. A. Poon, D. Narayanan, ”MorphNet: Towards Faster and Smaller Neural Networks”, Google AI Perception, 2019.
8. Polus cluster specifications http://hpc.cs.msu.su/polus April 2020.