# Accelerating Neural Network Training Process on Multi-Core Machine Using OpenMP

Omar T. Mohammed
*Department of Computer Science and Engineering*
*Saint Petersburg Electrotechnical University "LETI"*
Saint Petersburg, Russia
omar.taha.mohammed@gmail.com

Alexey A. Paznikov
*Department of Computer Science and Engineering*
*Saint Petersburg Electrotechnical University "LETI"*
Saint Petersburg, Russia
apaznikov@gmail.com

Sergei Gorlatch
*Department of Mathematics and Computer Science*
*University of Münster*
*Münster*, Germany
gorlatch@uni-muenster.de

2022 III International Conference on Neural Networks and Neurotechnologies (NeuroNT) | 978-1-6654-6776-6/22/$31.00 ©2022 IEEE | DOI: 10.1109/NeuroNT55429.2022.9805549

***Abstract*—Modern machine learning algorithms when applied to some real-world data, such as social networks and web graphs, can be very time-consuming. Despite enormous researches were focusing on making their approaches more scalable, however, their proposed approaches are running sequentially which makes the training run time remain noticeably long. Thus, it is reasonable to split the research among multiple processes. This is where we believe parallel processing can help. In this paper, we develop an OpenMP-based approach for parallelizing neural networks on multi-core CPUs. The novelty of our approach is that it is more general as it mainly covers CPU-based parallel training implementation, we focus on accelerating the training phase of neural networks using OpenMP that divides the work among multiple threads to run in parallel. Our experimental evaluation of a binary classification problem run on the banknote authentication dataset which contains images of banknotes with a machine with 12 cores demonstrates a significant acceleration of the training process compared to related works. We outline some possible approaches for further research concerning parallel optimization of execution time in neural network processes.**

***Keywords—parallel computing, machine learning, neural networks, OpenMP, multithreading, training for supervised learning***

## I. Introduction

Machine learning in general and supervised learning in particular are popular and successful in various application areas [1]. Supervised learning is a task of learning a function that maps an input to an output, based on examples of input-output pairs that are called patterns. Usually, the more complex the problem gets the more examples are required. Training a neural network for both, complex and multidimensional problems requires using many training examples with hundreds of thousands or even millions of patterns. Therefore, training process may take prohibitively long time. Furthermore, to find the optimal neural configuration often requires a specific amount of cross-validation experiments [2]. Hence, one of major challenges in supervised machine learning is that training and testing machine learning models for large, real-world datasets becomes very time-consuming [3].

In this paper, we develop a parallelization approach to accelerate the the training process in the supervised machine learning, and we implement it for multi-core processors (CPU) using the OpenMP (Open Multi Processing) standard.

This research was supported by RSF (project № 22-21-00686).

## II. Related work

Recent attempts to accelerate neural network training have shown good results, especially on GPU (Graphics Processing Units) [4]. An alternative approach for big data for neural network models is based on the MapReduce programming model [5]. However, if we increase the numerous number of parameters, which may be very numerous in the neural network models, training the model becomes a challenging task. [6].

Nickolls et al. [7] and Che et al. [8] employ CPU based parallel approaches such as MPI (Message Passing Interface), PThreads, and OpenMP. Papers [7, 8] use CUDA (Compute Unified Device Architecture) to employ streaming processors connected with external dynamic RAM (Random Access Memory) partitions.

Meng et al. [9] present an OpenMP parallelization and optimization of two new classification algorithms based on graphs and PDE (Partial Differential Equations) techniques, with performance and accuracy advantages over the traditional data classification.

In existing approaches, all training examples are processed one set per training update which is usually called at each iteration. This sequential organization increases the training time of neural networks [10]. A common practical solution is to employ the so-called mini-batch training: at each iteration, the neural network processes a subset of training examples until all examples are processed and then aggregates the training updates of the processed subsets. However, the aggregation cost of the mini-batch training causes an additional latency in the training process in large-scale applications that require large subsets of training examples [11]. J. Duchi et al. [12] introduce a parallel training approach that minimizes this latency.

We present in this paper a parallelization approach that exploits multi-core parallelism of modern CPU. We follow the idea of the mini-batch training, i.e., we process subsets of training examples in parallel, but we exclude some subsets from the aggregation of updates after several iterations depending on the update values.

In the remainder of the paper, Section 2 briefly describes the concept of multi-layer neural networks and presents different approaches to the training in neural networks. Section 3 describes our proposed parallelization approach. In Section 4, we explain our parallel implementation on a multi-core CPU using OpenMP. Section 5 presents our experimental evaluation using a real-world dataset: we compare our approach against the existing parallelization

978-1-6654-6776-6/22/$31.00 ©2022 IEEE

approach [12] for mini-batch training. Section 6 summarizes our findings.

## III. Supervised Learning in Multu-Layer Neural Networks

Fig. 1 shows a simple illustrative example of a neural network architecture often used in supervised learning – the Multi-Layer Perceptron (MLP) [13]. It consists of an input layer, at least one or more hidden layers, and an output layer. Each layer contains a certain number of nodes which are called neurons, all neighboring layers neurons are interconnected [14]. Our approach and experiments in this paper consider significantly more complex structures of neural networks than shown in Fig. 1 by having more hidden layers and neurons.

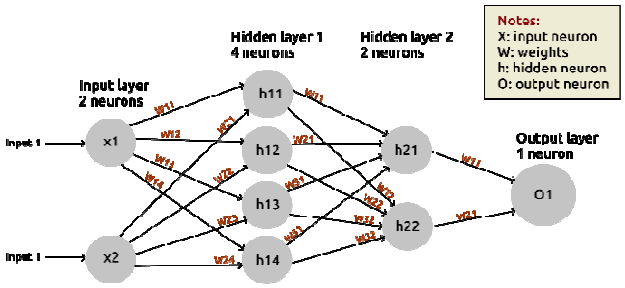


Fig. 1. Example of a 4-layer Multi-Layer Neural Network that has 2 input neurons, two hidden layers with 4 and 2 neurons respectively, and 1 output neuron

We address neural network structures that in terms of complexity are similar to those used, e.g., in Google's very successful Gboard application: it uses a network with 8 layers for the problem of end-to-end speech recognition for a mobile device. A supervised learning algorithm takes a known set of input data (patterns) and known responses to those data and it trains a model to generate proper predictions about the response to arbitrary new data [15].

In the network like the one in Fig. 1, each connection is associated with a weight value that is usually initialized randomly before the training begins. Each neuron calculates the weighted sum of its input data or, in other words, compute the sum of all weights in the previous layer and then transmit it into every neuron of the next layer. This transformation is done by an activation function; a widely used activation function for MLP is the sigmoid function: $sigmoid(x) = 1/(1+e^{-x})$.

Training of an MLP consists in adjusting weights using an optimization algorithm, in order to find a set of weight values that can best map inputs to desired outputs (targets). The training accuracy describes how precise this mapping becomes. The optimization algorithms have two variants: feed-forward and backpropagation. In feed-forward, computations originate from the input layer and proceed to the hidden layers and further to the output layer that finally calculates the output, which is the desired prediction [16].

The difference between the output (prediction) and the target value is called loss. The prediction accuracy of neural networks increases when their loss value decreases. Backpropagation is used to minimize the loss value: it computes the gradient of the loss value with respect to all weight values of a neural network for a single input-output training example to iteratively adjust the weight values [17]

There are three approaches to backpropagation, as follows. First, in Fig. 2 we show the approach called full batch training. A batch consists of training examples processed before the model is updated. Full-batch training trains the entire batch once per iteration as one set, by computing the feedforward for all training examples to get outputs (predictions) and loss values; then it computes the overall loss as the average of the accumulated loss values. For reducing the loss value, neural network uses backpropagation which updates the weight values per each training iteration. The process is repeated until the average loss value does not decrease anymore; this value is called *local minimum*.

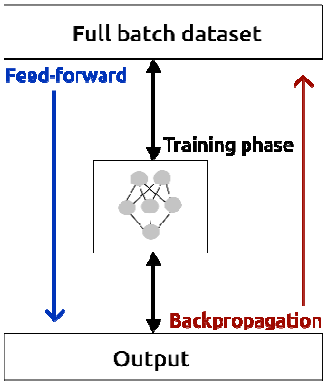


Fig. 2. Full-batch training of a neural network

Second, in Fig. 3 we illustrate the approach called mini-batch training. It divides a batch into subsets of training examples (mini-batches) of size S > 1 and then trains each mini-batch separately by applying feedforward per iteration to produce an output (prediction) and a loss value. Afterwards, backpropagation is applied to minimize the loss which is called gradient of loss; the gradient losses are then averaged for all mini-batches. The process is repeated until the average gradient loss does not decrease anymore: it is called global minimum.

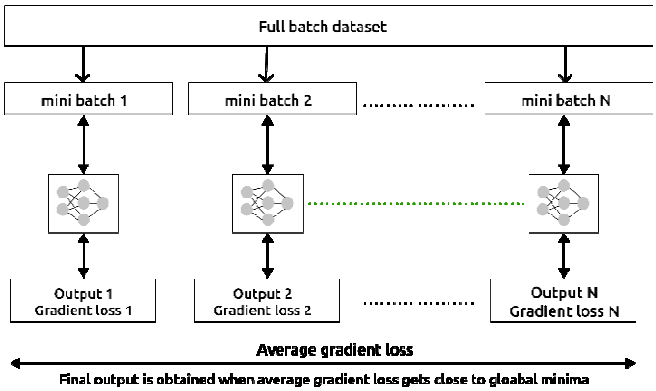


Fig. 3. Mini-batch training of a neural networks

The third approach, called stochastic training, works similarly to mini-batch training, but it performs training on one randomly selected example (one pattern) at a time i.e., the batch size is always equal to one

## IV. Our Parallelization Approach

Our approach is shown in Fig. 4: we parallelize the mini-batch training by distributing the training of the mini-batches across the cores of a multi-core CPU. Our approach differs from the existing mini-batch training the paper [12]: after a several iterations we exclude some mini-batches from averaging their gradient loss, based on their gradient loss values as explained in the following.

Fig. 4 shows that our parallelization approach proceeds in two steps as follows. In the first step, we assign each batch to a separate thread: we run threads in parallel to train the assigned mini-batches and produce a gradient loss with an output value per iteration. We repeat the first step for N iterations. Here, N is a hyperparameter to control the training process [18]. In neural networks, hyperparameter values are usually chosen experimentally. Our experiments show that we obtain a fast training process for N = 100. We save the gradient loss values of all mini-batches in a container which is called error container. Based on the error container, the second step of our approach selects a percentage (determined by another hyperparameter) of the mini-batches with a minimum gradient loss value for further training and excludes the rest. In our experiments, we obtain good training accuracy when we set the percentage hyperparameter value to 75%. Finally, we repeat the first step for M iterations and we average gradient loss values until the average gradient loss value does not decrease anymore, i.e., we obtain the global minimum. Hyperparameter M represents the maximum value of training iterations. In our experiments, it was found that a good value for M is 10000.

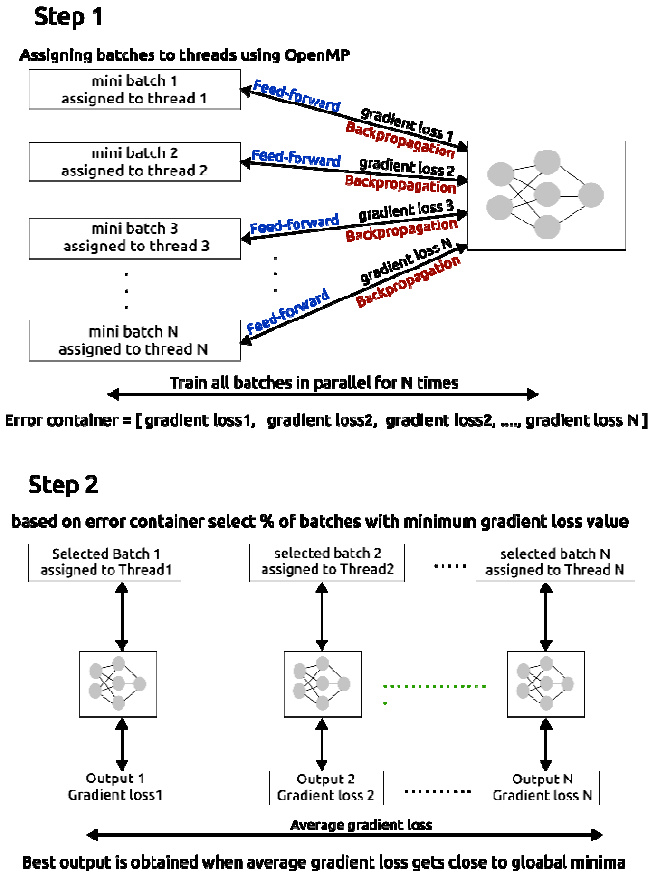


Fig. 4. Our parallelization approach: two-step scheme

## V. Implementation in OpenMP

Fig. 5 shows the implementation of our parallelization scheme described in the previous section: the work is divided between the master thread and several slave threads, according to the general OpenMP paradigm.

In the master thread of our OpenMP C++ program, we first create a new neural network model by initializing the input, the hidden, and the output layers along with their neurons and connecting them by arrows with the automatically generated random weights ranging between -1 and 1. The master thread reads training examples (dataset) as one batch and splits it into K minibatches, where K is the hyperparameter that influences the batch size. In our experiments, the batch size of 7 provides the fastest training process. The master thread then starts multiple concurrent slave threads and assigns to each of them a separate mini-batch with a copy of the original neural network model. Each slave thread performs the training and updates the weight values based on the current copy of the neural network and the mini-batch assigned to that slave.

Once all slaves finish their calculations, they send accumulated updated weight values to the master thread; the master thread then combines them to update the overall weight values in order to minimize the gradient loss of the neural network. This process is repeated until the gradient loss does not decrease anymore or until M is reached.

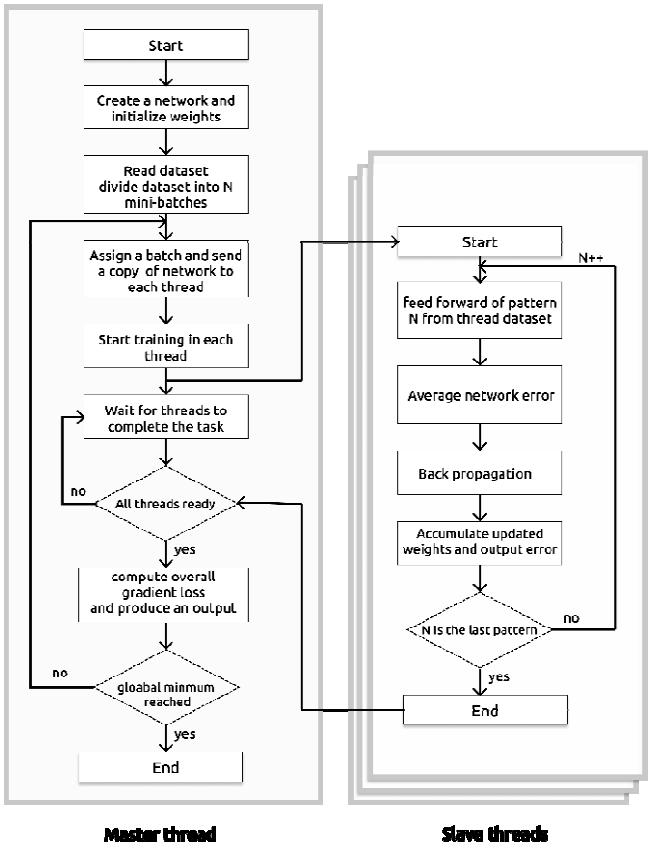


Fig. 5. Our backpropagation learning algorithm in OpenMP parallel implementation. Master thread divides the dataset into mini-batches of equal size. Slaves train the neural network using their subsets training examples. Master then computes the overall weights update (gradient loss) of the neural network

## VI. Experimental Evaluation

For experiments, we use the Banknote Authentication Dataset [19] which is a representative real-world use case for a binary classification representation. The dataset contains 1372 patterns which are images of banknotes with 4 predictor variables. The goal is to predict whether a given banknote is authentic or not, based on some measurements taken from a photograph. For all binary classification problems, the output layer of the neural network consists of one neuron that produce one output (prediction) and its value is a probability ranging between 0 and 1. We choose the binary classification representation for evaluating our approach because of its low computational cost with fewer hyperparameters than other

classifiers and also because of its competitive accuracy measures [20].

We experiment with a dataset that is complex, but not very large, because using a complex and very large dataset would require a machine with more parallel cores than the one available in our experiments.

All experiments are conducted on the following multicore processor: Intel Core i7–8750HQ 2.20 GHz, with 12 Cores and 16 GB RAM.

The structures of neural networks in our study are motivated by [21] and chosen after extensive experiments. We show in Table 1 two different neural network structures: the first network consists of 3 hidden layers with 18 neurons in each layer and overall, 5 layers including the input and output ones; the second network has 4 hidden layers, each having 24 neurons, overall, 6 layers. The complexity of the neural networks used in our experiments is similar to the networks currently used in practice. For example, Google's recent Gboard application for speech recognition uses an on-device neural network with the structure based on paper [22]: it has only 8 layers for solving a very complex problem of streaming end-to-end speech recognition on a mobile device. This application is very successful in practice.

TABLE I. EXPERIMENTAL RESULTS: OUR APPROACH VS.EXISTING MINI-BATCH PARALLELIZATION IN [12].

| *Training Algorithm* | *Threads* | *Hidden Layers* | *Neurons* | *Execution Time* | *Accuracy* |
|---|---|---|---|---|---|
| Proposed approach | 10 | 3 | 18 | 28 ms | 92.4 % |
| Existing mini-batch approach | 10 | 3 | 18 | 35 ms | 94.3 % |
| Proposed approach | 12 | 3 | 18 | 18 ms | 88.7 % |
| Existing mini-batch approach | 12 | 3 | 18 | 24 ms | 93.6 % |
| Proposed approach | 14 | 3 | 18 | 31 ms | 72.2 % |
| Existing mini-batch approach | 14 | 3 | 18 | 35 ms | 94.3 % |
| Proposed approach | 10 | 4 | 24 | 35 ms | 83.3 % |
| Existing mini-batch approach | 10 | 4 | 24 | 42 ms | 72.7 % |
| Proposed approach | 12 | 4 | 24 | 29 ms | 92.4 % |
| Existing mini-batch approach | 12 | 4 | 24 | 38 ms | 97.2 % |
| Proposed approach | 14 | 4 | 24 | 34 ms | 87 % |
| Existing mini-batch approach | 14 | 3 | 18 | 48 ms | 84.1 % |

In Table 1, we compare our proposed approach against the existing mini-batch training approach [12]. In our experiments, we start the training process for both approaches using the same initial generated random weight values of the network. To obtain more precise results, we conduct each experiment several times and then average the computed values. The table contains information on the neural network configurations, batch size, number of used threads, and the measured execution time of the training.

Table 1 demonstrates an obvious trend that using more threads than available processor cores cause some synchronization overhead which lowers the efficiency of parallelization. We also notice that our approach has a faster training process but slightly lower accuracy compared to the mini-batch training approach in [12]. A probable reason is that our approach excludes some training examples during the training process.

Fig. 6 compares the speed-up of two OpenMP-based parallelization approaches: the existing mini-batch training [12] and the approach described in this paper.

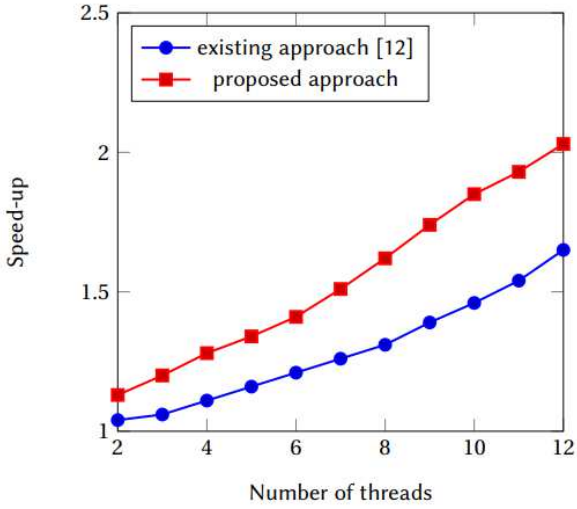


Fig. 6. Speed-up of our parallelization vs. existing minibatch parallelization [12]

We observe in the figure that our proposed approach achieves its best speed-up when it uses a medium batch size with as many threads as available cores. Every core is loaded with the training of the assigned patterns, and a relatively we need short time of synchronization to update the weights of the master thread and to copy them again to the slave threads. In Fig. 6 we see that the fastest training is obtained when using 12 threads on a 12-core CPU.

We also observe that the achieved speed-up is quite modest. We explain this by the general restrictions of the supervised learning based on mini-batch approach [23]: 1) to compute the overall gradient loss update, the master thread needs to combine the gradient loss updates received from all slave threads – this reduction step is an extra work compared to sequential execution; 2) for every batch, the parallel method performs two thread synchronizations – after all slaves complete their gradient loss update, and after the reduction step is done; and 3) since the overall update of the model is not applied until the end of the entire batch (dataset), updates within the batch become increasingly obsolete, as being based on a model that is out of date. We plan to address these restrictions of our approach in future work.

## VII. CONCLUSION

In this work, we propose a parallelization approach to supervised learning of neural networks. Our approach is implemented using OpenMP. Our experimental evaluation

runs on the processor with 12 parallel cores for the banknote authentication dataset that contains images of banknotes for deciding which of them are genuine and which are not. We demonstrated the improvements made by our approach regarding the speed-up of the training process as compared to the existing parallelized mini-batch training.

## REFERENCES


[1] J. Li, Regression and classification in supervised learning, in: Proceedings of the 2nd International Conference on Computing and Big Data,Association for Computing Machinery, New York, NY, USA, 2019, p. 99–104. doi:10.1145/3366650.3366675*)*

[2] S. B. Kotsiantis, Supervised machine learning: A review of classification techniques, in: Proceedings of the 2007 Conference on Emerging Artificial Intelligence Applications in Computer Engineering: Real Word AI Systems with Applications in EHealth, HCI, Information Retrieval and Pervasive Technologies, IOS Press, NLD, 2007, p. 3–24..

[3] O. Alonso, Challenges with label quality for supervised learning, J. Data and Information Quality 6 (2015). URL: https://doi.org/10.1145/2724721.

[4] O. Yadan, K. Adams, Y. Taigman, M. Ranzato, Multi-GPU training of convnets, 2013. arXiv:1312.5853.

[5] C.-T. Chu, S. K. Kim, Y.-A. Lin, Y. Yu, G. Bradski, A. Y. Ng, K. Olukotun, Map-Reduce for machine learning on multicore, in: Proceedings of the 19th International Conference on Neural Information Prcessing Systems, NIPS'06, MIT Press, Cambridge, MA, USA, 2006, p. 281–288o

[6] L. Mai, A. Koliousis, G. Li, A.-O. Brabete, P. Pietzuch, Taming hyper-parameters in deep learning systems, SIGOPS Oper. Syst. Rev. 53 (2019) 52–58. URL: https://doi.org/10.1145/3352020.3352029.

[7] J. Nickolls, I. Buck, M. Garland, K. Skadron, Scalable parallel programming with CUDA: Is CUDA the parallel programming model that application developers have been waiting for?, Queue 6 (2008) 40–53. doi:10.1145/1365490.1365500

[8] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, K. Skadron, A performance study of generalpurpose applications on graphics processors using CUDA, Journal of Parallel and Distributed Computing 68 (2008) 1370–1380. doi:https://doi.org/ 10.1016/j.jpdc.2008.05.014.

[9] Z. Meng, A. Koniges, Y. H. He, S. Williams, T. Kurth, B. Cook, J. Deslippe, A. L. Bertozzi, Openmp parallelization and optimization of graph-based machine learning algorithms, Lawrence Berkeley National Laboratory 9903 (2016).

[10] V. Md, S. Misra, G. Ma, R. Mohanty, E. Georganas,A. Heinecke, D. Kalamkar, N. K. Ahmed, S. Avancha, Distgnn: Scalable distributed training for large-scale graph neural networks, in: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC '21, Association for Computing Machinery, New York, NY, USA, 2021. doi:10.1145/3458817.3480856

[11] L. Ziyin, K. Liu, T. Mori, M. Ueda, On minibatch noise: Discrete-time sgd, overparametrization, and bayes, ArXiv abs/2102.05375 (2021).

[12] S. Chaturapruek, J. C. Duchi, C. Ré, Asynchronous stochastic convex optimization: the noise is in the noise and sgd don't care, in: C. Cortes, N. Lawrence, D. Lee, M. Sugiyama, R. Garnett (Eds.), Advances in Neural Information Processing Systems, volume 28, Curran Associates, Inc.,2015

[13] J. Singh, R. Banerjee, A study on single and multilayer perceptron neural network, in: 2019 3rd International Conference on Computing Methodologies and Communication (ICCMC), 2019, pp. 35–40. doi:10.1109/ICCMC.2019.8819775.

[14] O. T. Mohammed, M. S. Heidari, A. A. Paznikov, Mathematical computations based on a pretrained AI model and graph traversal, in: 2020 9th Mediterranean Conference on Embedded Computing (MECO), 2020, pp. 1–4. doi:10.1109/ MECO49872.2020.9134081.

[15] J. Alcaraz, S. Sleder, A. TehraniJamsaz, A. Sikora, A. Jannesari, J. Sorribes, E. Cesar, Building representative and balanced datasets of openmp parallel regions, in: 2021 29th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP), 2021, pp. 67–74. doi:10.1109/PDP52278.2021.00019.

[16] I. Kholod, A. Rukavitsyn, A. Paznikov, S. Gorlatch, Parallelization of the self-organized maps algorithm for federated learning on distributed sources, The Journal of Supercomputing (2020). doi:10.1007/s11227-020-03509-2.

[17] H. B. Demuth, M. H. Beale, O. De Jess, M. T. Hagan, Neural Network Design, 2nd ed., Martin Hagan, Stillwater, OK, USA, 2014.

[18] R. Liu, S. Krishnan, A. J. Elmore, M. J. Franklin, Understanding and optimizing packed neural network training for hyper-parameter tuning, DEEM '21, Association for Computing Machinery, New York, NY, USA, 2021. doi:10.1145/3462462.3468880.

[19] V. Lohweg, Banknote-authentication dataset, Dataset (2012). URL: https://archive.ics.uci.edu/ml/datasets/banknote+authentication.

[20] K.-A. Toh, Z. Lin, L. Sun, Z. Li, Stretchy binary classification, Neural Netw. 97 (2018) 74–91. doi:10.1016/j.neunet.2017.09.015.

[21] M. Madhiarasan, S. N. Deepa, A novel criterion to select hidden neuron numbers in improved back propagation networks for wind speed forecasting, Applied Intelligence 44 (2016) 878–893.doi:10.1007/s10489-015-0737-z.

[22] Y. He, T. N. Sainath, R. Prabhavalkar, I. McGraw, R. Alvarez, D. Zhao, D. Rybach, A. Kannan, Y. Wu, R. Pang, et al., Streaming end-to-end speech recognition for mobile devices, in: ICASSP 2019 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP), 2019, pp. 6381–6385.

[23] S. Sallinen, N. Satish, M. Smelyanskiy, S. S. Sury, C. Ré, High performance parallel stochastic gradient descent in shared memory, in: 2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS), 2016, pp. 873–882. doi:10.1109/IPDPS.2016.107.