MDPI

Article

Efficient Supervised Learning on Multicores Using OpenMP

Omar Mohammed [1,†]*, Alexey Paznikov [1,] and Sergei Gorlatch [2,†]

[1] Saint Petersburg Electro-Technical University (LETI), ul. Professora Popova 5, 197376 St. Petersburg, Russia
[2] University of Münster, Einsteinstr. 62, 48149 Münster, Germany
* Correspondence: omar.taha.mohammed@gmail.com
† These authors contributed equally to this work.

Abstract: Modern machine learning algorithms, when applied to large, real-world data, such as social networks and web graphs, may be very time-consuming. In this paper, we accelerate the training process of neural networks by developing a parallelization approach for multi-core CPUs and implementing it based on the OpenMP library. Our experimental evaluation is conducted on a binary classification problem with a real-world training dataset: it demonstrates a significant acceleration of the training process as compared to related work.

Keywords: Parallel computing; Machine learning; Neural networks; OpenMP; Multithreading; Training for supervised learning

1. Motivation and related work

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 needed. Training a neural network for complex and multidimensional problems requires using large numbers of training examples with hundreds of thousands or even millions of patterns. Therefore, training process may take prohibitively long time. Furthermore, finding an optimal neural network configuration often requires a certain 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.

Recent attempts to accelerate neural network training have shown good results, especially on GPU (Graphics Processing Unit) [4]. An alternative approach to dealing with big data for neural network models is based on the MapReduce programming model [5]. However, with the increase of the number of parameters which may be very numerous in the neural network models, it become a challenge to train the model [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.

Citation: Mohammed, O.; Paznikov, A.; Gorlatch, S. Title. *Journal Not Specified* **2022**, *1*, 0. https://doi.org/

Received:
Accepted:
Published:

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

2. Supervised learning in multi-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 amount of nodes which are called neurons, and all neurons of neighboring layers 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.

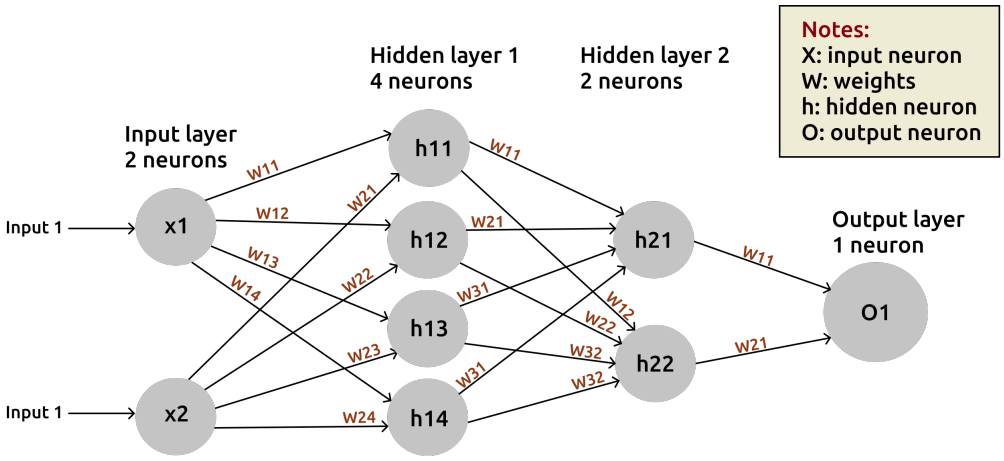

Figure 1. Example of a 4-layer Multi-Layer Neural Network that has 2 input neurons, two hidden layers with 4 and 2 neurons respectively, and 1 output neuron.

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 on a mobile device. A supervised learning algorithm receives 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. The job of each neuron is to compute the weighted sum of its inputs 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*.

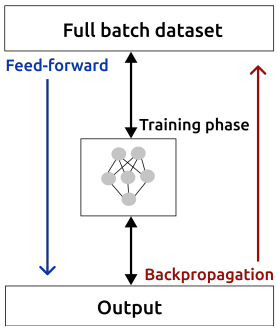

Figure 2. Full-batch training of a neural network.

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 feed-forward 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*.

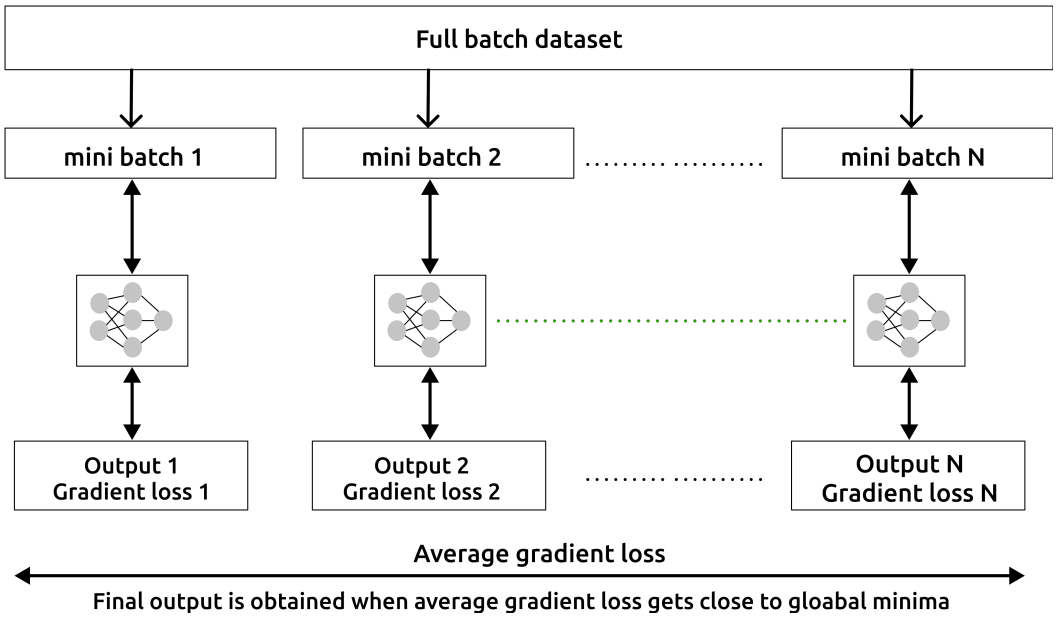

Figure 3. Mini-batch training of a neural networks.

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.

3. 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 [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.

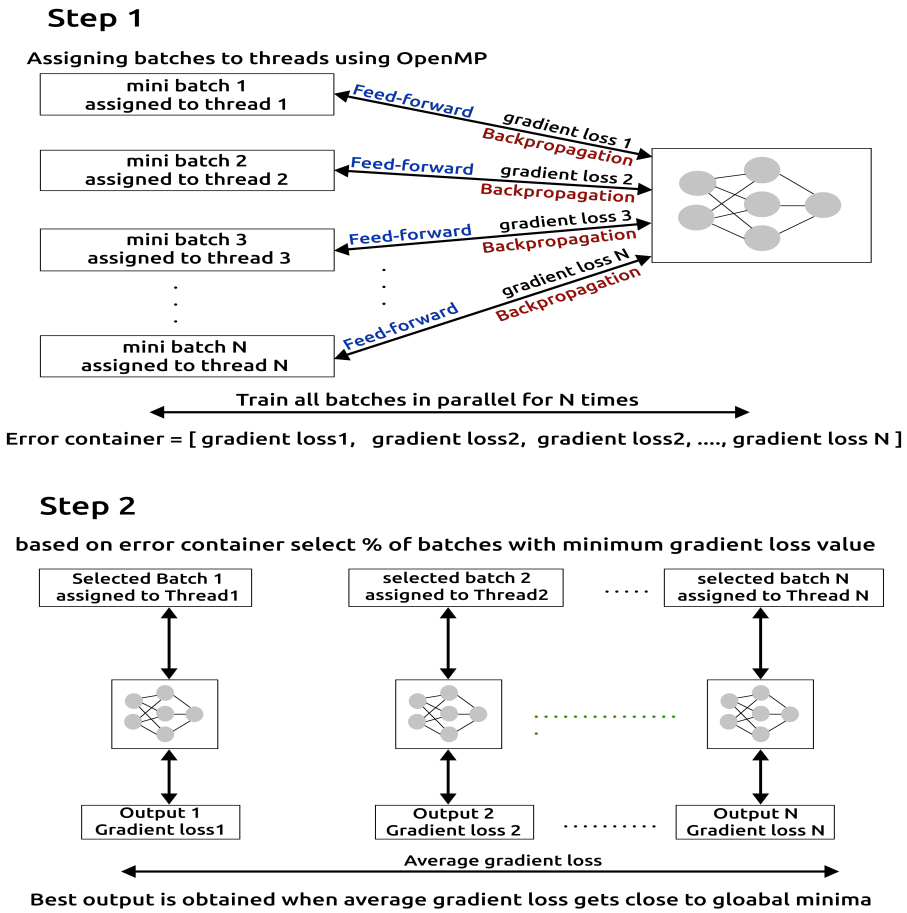

Figure 4. Our parallelization approach: two-step scheme.

4. 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.

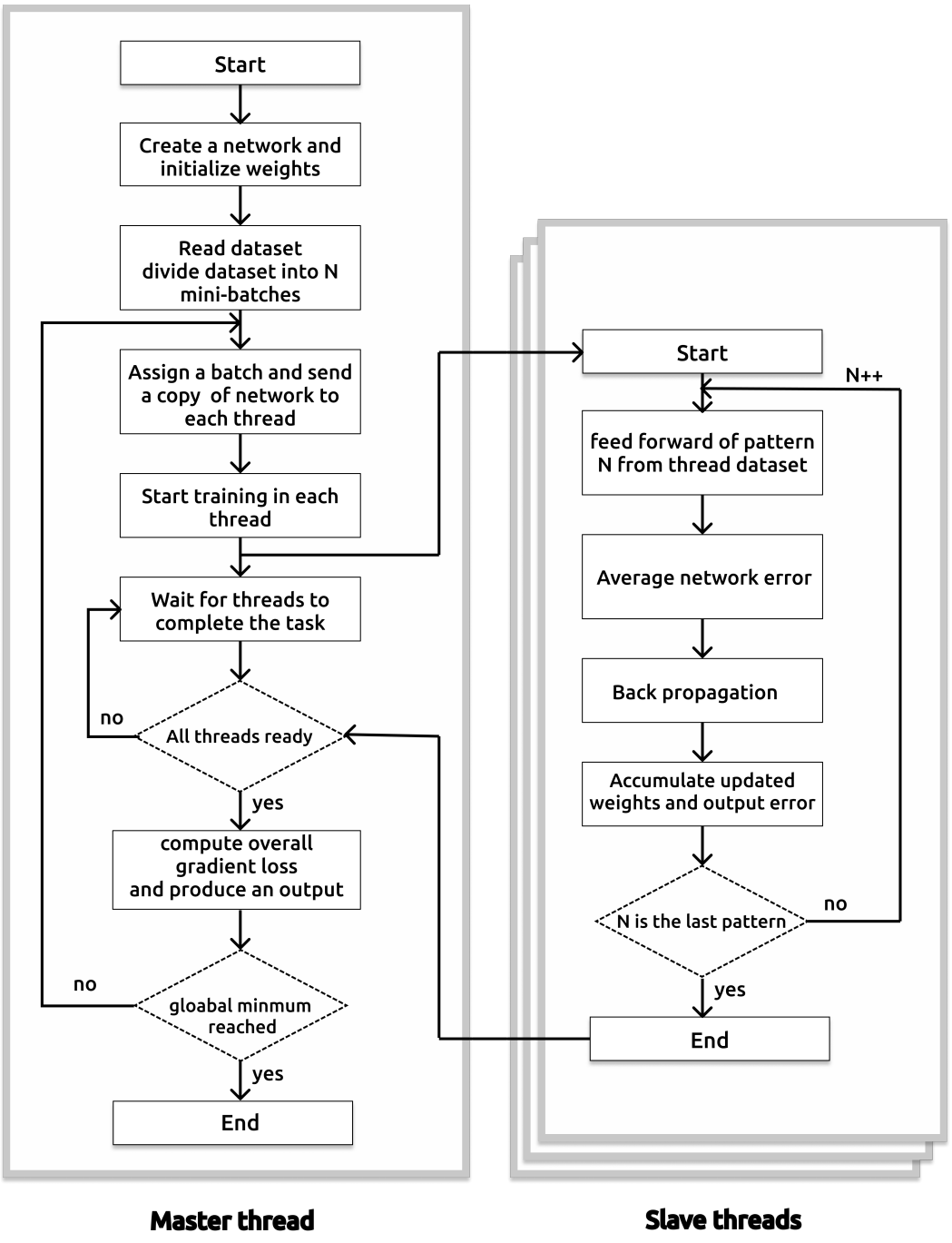

Figure 5. Our parallel implementation of the backpropagation learning algorithm in OpenMP. Master thread divides the dataset into mini-batches of equal size. Slaves train the neural network using their subsets training examples. Master then computes the overall weights update (gradient loss) of the neural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

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 mini-batches, 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. As soon as all slaves finish their computations, they send accumulated updated weight values to the master thread; the master thread 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.

5. 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 produces 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.

Table 1. Experimental results: our approach vs. existing mini-batch parallelization in [12].

Training Algorithms	Batch Size	Threads	Hidden Layers	Neurons	Iterations	Execution Time	Accuracy
proposed approach	7	10	3	18	10000	28 ms	92.4 %
existing mini-batch approach	7	10	3	18	10000	35 ms	94.3%
proposed approach	7	12	3	18	10000	18 ms	88.7 %
existing mini-batch approach	7	12	3	18	10000	24 ms	93.6%
proposed approach	7	14	3	18	10000	31 ms	72.2 %
existing mini-batch approach	7	14	3	18	10000	35 ms	94.3%
proposed approach	7	10	4	24	10000	35 ms	83.3 %
existing mini-batch approach	7	10	4	24	10000	42 ms	72.7%
proposed approach	7	12	4	24	10000	29 ms	92.4 %
existing mini-batch approach	7	12	4	24	10000	38 ms	97.2%
proposed approach	7	14	4	24	10000	34 ms	87 %
existing mini-batch approach	7	14	3	18	10000	48 ms	84.1%

All experiments are conducted on the following multi-core 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.

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 causes 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.

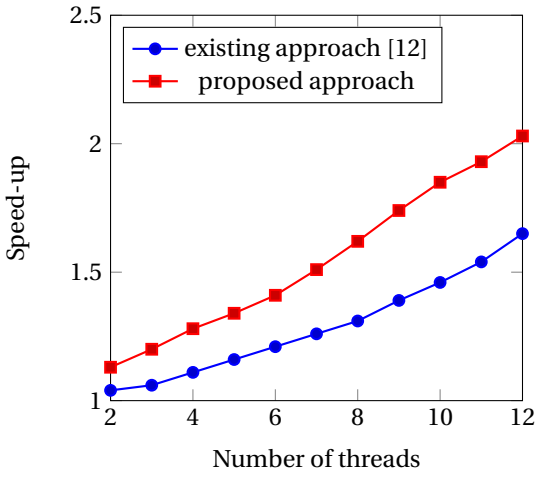

Figure 6. Speed-up of our parallelization vs. existing mini-batch parallelization.

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 short synchronization time is needed for updating the weights of the master thread and copying them back 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.

6. Conclusions

In this paper, 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.

Author Contributions: Conceptualization, O.M. and S.G.; methodology, O.M.; software, O.M.; validation, O.M., S.G. and A.P.; formal analysis, O.M. And S.G; investigation, O.M.; resources, O.M. and A.P; data curation, O.M.; writing—original draft preparation, O.M.; writing—review and editing, S.G.; visualization, O.M.; supervision, S.G. and A.P; project administration, A.P.; funding acquisition, A.P. All authors have read and agreed to the published version of the manuscript.

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

Institutional Review Board Statement: Not applicable

Informed Consent Statement: Not applicable

Data Availability Statement: Banknote Authentication Dataset at https://archive.ics.uci.edu/ml/datasets/banknote+authentication.

Conflicts of Interest: The authors declare no conflict of interest.

Abbreviations

The following abbreviations are used in this manuscript:

CPU	Central Processing Unit
OpenMP	Open Multi-Processing
GPU	Graphic Processinh Unit
MPI	Massage Passing Interface
RAM	Random Access Memory
CUDA	Compute Unified Device Architecture
PDE	Partial Differential Equations
MLP	Multi-Layer Perceptron

References

1. Li, J. Regression and Classification in Supervised Learning. In Proceedings of the Proceedings of the 2nd International Conference on Computing and Big Data; Association for Computing Machinery: New York, NY, USA, 2019; p. 99–104. https://doi.org/10.1145/3366650.3366675.
2. Kotsiantis, S.B. Supervised Machine Learning: A Review of Classification Techniques. In Proceedings of the 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. Alonso, O. Challenges with Label Quality for Supervised Learning. *J. Data and Information Quality* **2015**, *6*.
4. Yadan, O.; Adams, K.; Taigman, Y.; Ranzato, M. Multi-GPU Training of ConvNets, 2013, [arXiv:cs.LG/1312.5853].
5. Chu, C.T.; Kim, S.K.; Lin, Y.A.; Yu, Y.; Bradski, G.; Ng, A.Y.; Olukotun, K. Map-Reduce for Machine Learning on Multicore. In Proceedings of the Proceedings of the 19th International Conference on Neural Information Processing Systems; MIT Press: Cambridge, MA, USA, 2006; NIPS'06, p. 281–288.
6. Mai, L.; Koliousis, A.; Li, G.; Brabete, A.O.; Pietzuch, P. Taming Hyper-Parameters in Deep Learning Systems. *SIGOPS Oper. Syst. Rev.* **2019**, *53*, 52–58.
7. Nickolls, J.; Buck, I.; Garland, M.; Skadron, K. Scalable Parallel Programming with CUDA: Is CUDA the Parallel Programming Model That Application Developers Have Been Waiting For? *Queue* **2008**, *6*, 40–53. https://doi.org/10.1145/1365490.1365500.
8. Che, S.; Boyer, M.; Meng, J.; Tarjan, D.; Sheaffer, J.W.; Skadron, K. A performance study of general-purpose applications on graphics processors using CUDA. *Journal of Parallel and Distributed Computing* **2008**, *68*, 1370–1380. https://doi.org/https://doi.org/10.1016/j.jpdc.2008.05.014.
9. Meng, Z.; Koniges, A.; He, Y.H.; Williams, S.; Kurth, T.; Cook, B.; Deslippe, J.; Bertozzi, A.L. OpenMP Parallelization and Optimization of Graph-Based Machine Learning Algorithms. *Lawrence Berkeley National Laboratory* **2016**, *9903*.
10. Md, V.; Misra, S.; Ma, G.; Mohanty, R.; Georganas, E.; Heinecke, A.; Kalamkar, D.; Ahmed, N.K.; Avancha, S. DistGNN: Scalable Distributed Training for Large-Scale Graph Neural Networks. In Proceedings of the Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis; Association for Computing Machinery: New York, NY, USA, 2021; SC '21. https://doi.org/10.1145/3458817.3480856.
11. Ziyin, L.; Liu, K.; Mori, T.; Ueda, M. On Minibatch Noise: Discrete-Time SGD, Overparametrization, and Bayes. *ArXiv* **2021**, *abs/2102.05375*.
12. Chaturapruek, S.; Duchi, J.C.; Ré, C. Asynchronous stochastic convex optimization: the noise is in the noise and SGD don't care. In Proceedings of the Advances in Neural Information Processing Systems; Cortes, C.; Lawrence, N.; Lee, D.; Sugiyama, M.; Garnett, R., Eds. Curran Associates, Inc., 2015, Vol. 28.
13. Singh, J.; Banerjee, R. A Study on Single and Multi-layer Perceptron Neural Network. In Proceedings of the 2019 3rd International Conference on Computing Methodologies and Communication (ICCMC), 2019, pp. 35–40. https://doi.org/10.1109/ICCMC.2019.8819775.
14. Mohammed, O.T.; Heidari, M.S.; Paznikov, A.A. Mathematical Computations Based on a Pre-trained AI Model and Graph Traversal. In Proceedings of the 2020 9th Mediterranean Conference on Embedded Computing (MECO), 2020, pp. 1–4. https://doi.org/10.1109/MECO49872.2020.9134081.
15. Alcaraz, J.; Sleder, S.; TehraniJamsaz, A.; Sikora, A.; Jannesari, A.; Sorribes, J.; Cesar, E. Building representative and balanced datasets of OpenMP parallel regions. In Proceedings of the 2021 29th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP), 2021, pp. 67–74. https://doi.org/10.1109/PDP52278.2021.00019.

16. Kholod, I.; Rukavitsyn, A.; Paznikov, A.; Gorlatch, S. Parallelization of the self-organized maps algorithm for federated learning on distributed sources. *The Journal of Supercomputing* **2020**. https://doi.org/10.1007/s11227-020-03509-2.
17. Demuth, H.B.; Beale, M.H.; De Jess, O.; Hagan, M.T. *Neural Network Design*, 2nd ed.; Martin Hagan: Stillwater, OK, USA, 2014.
18. Liu, R.; Krishnan, S.; Elmore, A.J.; Franklin, M.J. Understanding and Optimizing Packed Neural Network Training for Hyper-Parameter Tuning; Association for Computing Machinery: New York, NY, USA, 2021; DEEM '21. https://doi.org/10.1145/3462462.3468880.
19. Lohweg, V. Banknote-authentication dataset. *Dataset* **2012**.
20. Toh, K.A.; Lin, Z.; Sun, L.; Li, Z. Stretchy Binary Classification. *Neural Netw.* **2018**, *97*, 74–91. https://doi.org/10.1016/j.neunet.2017.09.015.
21. Madhiarasan, M.; Deepa, S.N. A Novel Criterion to Select Hidden Neuron Numbers in Improved Back Propagation Networks for Wind Speed Forecasting. *Applied Intelligence* **2016**, *44*, 878–893. https://doi.org/10.1007/s10489-015-0737-z.
22. He, Y.; Sainath, T.N.; Prabhavalkar, R.; McGraw, I.; Alvarez, R.; Zhao, D.; Rybach, D.; Kannan, A.; Wu, Y.; Pang, R.; et al. Streaming end-to-end speech recognition for mobile devices. In Proceedings of the ICASSP 2019 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP), 2019, pp. 6381–6385.
23. Sallinen, S.; Satish, N.; Smelyanskiy, M.; Sury, S.S.; Ré, C. High Performance Parallel Stochastic Gradient Descent in Shared Memory. In Proceedings of the 2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS), 2016, pp. 873–882. https://doi.org/10.1109/IPDPS.2016.107.