Parallelization of a Neural Network Algorithm for Handwriting Recognition: Can we Increase the Speed, Keeping the Same Accuracy

D. Todorov*, V. Zdraveski*, M. Kostoska* and M. Gusev*
* Ss. Cyril and Methodius University in Skopje, Faculty of Computer Science and Engineering, Skopje, North Macedonia
E-mail: david.todorov@students.finki.ukim.mk, { vladimir.zdraveski, magdalena.kostoska, marjan.gushev }@finki.ukim.mk

***Abstract*—This paper examines the problem of parallelizing neural network training using the back-propagation neural network, as a breakthrough example in the field of deep learning. The challenge of our solution is to twist the algorithm in such a way so it can be executed in parallel, rather than sequentially. In this paper, we test validity of a research hypothesis if the speed can be increased by parallelizing the back-propagation algorithm and keep the same accuracy. For this purpose, we developed a use-case of a handwriting recognition algorithm and conducted several experiments to test the performance, both in execution speed and accuracy. At the end, we examine just how much it benefits to write a parallel program for a neural network, with regards to the time it takes to train the neural network and the accuracy of the predictions. Our handwriting problem is that of classification, and in order to implement any sort of solution, we must have data. The MNIST dataset of handwritten digits provides necessary data to solve the problem.**

***Index Terms*—message passing interface, neural network, handwriting recognition, multilayer perceptron, parallel processing, distributed processing**

I. INTRODUCTION

We live in a world where everyone has a smartphone at their disposal and can write anything that comes to mind on it. Even so, that world is still, and will continue to be filled with handwritten notes, texts and descriptions. The prime example are schools where students write their notes in paper notebooks, rather than on their smartphones. Some students are naturally good at organizing their handwritten notes, but some prefer them to be in a digital format in which they could store them in a cloud, manipulate them etc. Instead of having to rewrite all of their notes, with an app for handwriting recognition one could just scan or take a picture of a page of a notebook, feed it to a piece of software that would be able to process the image and give the written text as output. Said output could then be processed and stored as per the user's desire.

Using machine learning (ML), researchers have actively been developing new and refining old ways of recognising handwritten text. One of, if not the biggest breakthrough in the field of ML is the multilayer perceptron algorithm, more commonly called a neural network (NN).

Of course, computer scientists would like to speed up and optimize the process, if even just a little bit. The idea for optimizing the training process of the NN is to run it in parallel on multiple CPU cores, or even (if the hardware is present) on multiple workstations. Naturally, we also have to check the accuracy of the neural network trained in parallel, and how it compares to the accuracy of the sequentially trained NN.

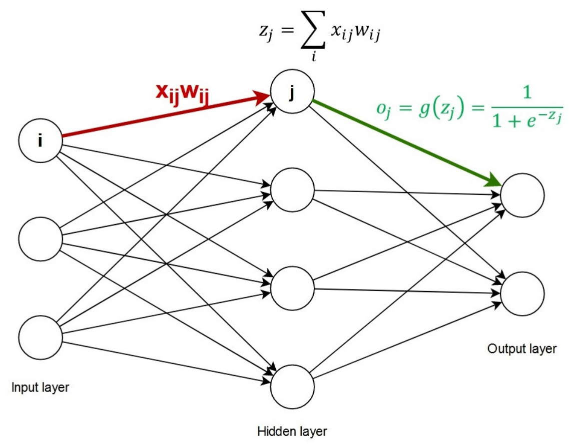
The hypothesis we make is loosely based on the No-Free-Lunch Theorem, in the sense that there can't be a one-size-fits-all solution to all problems. More concretely, the assumption is that there will have to be a trade-off between execution speed and prediction accuracy. This hypothesis is validated by measuring the time the algorithm takes to finish training the NN sequentially, in contrast to the parallel execution. After the different training sessions, the prediction accuracy is checked on the test set. Our expectation is that it will be slightly lower for the parallel implementation, due to the fact that the algorithm will not converge in a conventional sense as it would if it ran sequentially.

There have been multiple approaches to parallelizing a NN. Petchick et al. [1] explains four approaches in order of level of granularity, from training session parallelism, to weight parallelism. Training session parallelism consists of no communication between the processes, providing zero overhead. Weight parallelism is the finest grained solution. With it, the input from each synapse is calculated in parallel for each node, and the net input is summed via some suitable communications scheme. Our solution is based on the exemplar parallelism approach which provides little communication overhead and is very suitable to our experimental environment.

In Section II, we start by examining some related scientific papers regarding the problems of NN, handwriting recognition, and parallelization, and how they relate to this paper. Then, a basic overview of our solution is presented in Section III, first by going over existing knowledge of standard NN structure, followed by a bird's eye view of the parallel architecture. The implementation is discussed in Section IV providing an explanation of a standard sequential implementation for the NN classification problem, a parallel implementation, and by giving details on the specific environment being used to run the tests and provide our results. In Section V we compare the results generated by the different implementations, and in Section VI we asses the impact and worth of our solution.

II. RELATED WORK

The invention of neural networks has been one of the biggest breakthroughs in AI and ML. Heckt-Nielsen [2]



$z_j = \sum_i x_{ij} w_{ij}$
$x_{ij} w_{ij}$
i
j
$o_j = g(z_j) = \frac{1}{1+e^{-z_j}}$
Input layer
Hidden layer
Output layer

Fig. 1. Backpropagation NN structure

presents a survey of the basic theory of the back-propagation NN architecture covering architectural design, performance measurement, function approximation capability, and learning. One of, if not the most, useful applications of neural networks is in pattern recognition. Pao [3] has elaborated the nature of the pattern-recognition task and adaptive pattern recognition (and its applications) as one of the most useful applications of AI.

One application in which pattern recognition would be right at home is in the recognition of handwritten text. Graves and Schmidhuber [4] combine two recent innovations in NN, multidimensional recurrent NN and connectionist temporal classification [5]. They introduce a globally trained offline handwriting recogniser that takes raw pixel data as input. Unlike competing systems, it does not require any alphabet specific pre-processing, and can therefore be used unchanged for any language.

AI requires huge computational power for processing. Seeing as microprocessor manufacturers have struggled to increase the raw computational power of CPUs, the relevancy of Moore's Law has slowly been fading. This in turn has increased the need for engineers to think of new ways to increase the amount of computation that is possible in a finite period of time. This is where parallel computing comes in with [6] and [7], introducing new paradigms for parallel algorithm design, performance analysis and program construction. Akl [8] surveys existing parallel algorithms, with emphasis on design methods and complexity results. Parallelizable optimization techniques are applied to the problem of pattern recognition and learning in feed-forward neural networks by Kramer [9].

III. Solution overview

A. Neural network structure

Our solution is designed using the standard multilayer perceptron NN, also known as a backpropagation NN shown in Fig. 1. The NN consists of an input layer of nodes (perceptrons), an output layer of nodes, and at least one (usually more) intermediate layer. To keep things simple, we will train NNs with only one hidden layer. The network is trained on a data set consisting of (x_i, y_i) pairs where x_i is a feature vector used as an input, and y_i is the true output, or the ground truth, for that feature vector.

A node in the i^{th} layer of the NN is connected to all nodes in the $(i+1)^{th}$ layer. The connectors that connect a node in a layer to the nodes in the next layer have assigned weights w_{ij}. We denote by x_{ij} the i^{th} input to the node j. A node in one layer takes all of the products $x_{ij}w_{ij}$ that come as inputs from the previous layer, computes their sum z_j and uses it to compute the output (or the input to the next layer).

The NN learns by comparing the computed output with the true output, and adjusting the weights of the nodes accordingly so the computed output is the same as the desired one (if the computed output is already the true output, no adjustment is done). The changes to the weights propagate back through the network (hence the name) up until they reach the input layer. The algorithm stops after a full epoch without change in weights (sometimes we stop the algorithm prematurely to combat the problem of overfitting to the training set). Sometimes the outputs of the problem are not uniformly distributed among the population of the input/output pairs. For this reason we need a BIAS node with its own weight assigned to it, at each non-output layer, that has no inputs going into it. If the population is biased towards a certain output class, the BIAS allows us to skew the evaluation function so that we can better predict the true output.

B. Parallelization

There are multiple strategies to implement a parallel solution for the backpropagation NN. Some of the main strategies are discussed by Pethick, Liddle, Huang and Werstein [1]. We will go with the ”preferred technique” according to Rogers and Skillicorn [10], which is exemplar parallelism, also known training example parallelism. In essence, the training set is split into disjoint subsets and each running process (a thread, microprocessor or separate machine) works on only one subset. The processes need to start with initial states identical to each other, meaning the weights associated with each node need to be the same. Usually this means that every weight at the beginning is set to 1. At the end of the training, the changes are combined and applied together to the neural network by averaging them out. The diagram for this parallel solution approach is given in Fig. 2.

Two advantages come to mind when we think about this approach to parallelizing a NN. First is the small overhead that occurs because of the communication between processes. Namely, the processes only communicate at the beginning to distribute the data, and at the end of the training to average out the weights, so relatively few messages are generated. The second advantage is the speed-up during the training phase. Since the data set is split into n smaller subsets s_i, where $i = 1, 2, ..., n$, the time it takes to go through an epoch is the

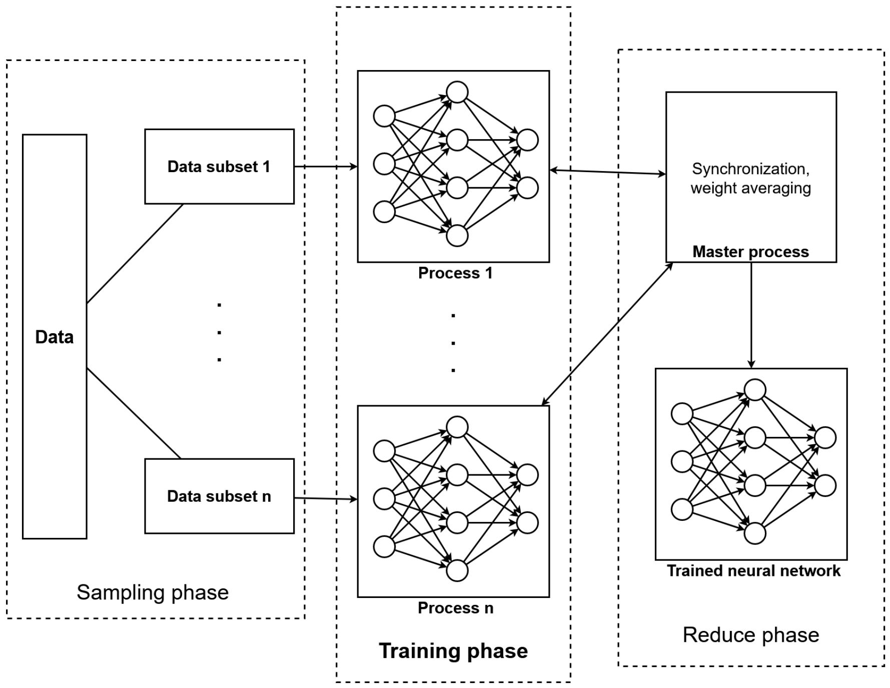


Fig. 2. The parallelization method. The dataset is split equally among the processes, each process trains its own neural network, and at the end all the processes average out their weight vectors.

maximum of the times t_i it takes to iterate through a subset s_i, or $t_{epoch} = max(t_i)$. A disadvantage to this approach is that it does not provide a performance increase at the layer level, meaning that no two nodes in the same process, working on the same subset of data, work in parallel.

IV. Experimental methods

For the purposes of this research, we will conduct two experiments that correspond to two implementations for the handwriting recognition back-propagation NN. The first one will be a classical sequential solution and the other will be a parallel implementation. For both experiments, we will use Python with its scikit-learn library which includes the tool needed for training NNs, MLPClassifier.

For the sake of simplicity, the dataset we will be using to train the NN is the MNIST dataset [11] of handwritten digits which includes 60,000 labeled 28x28 images of handwritten digits in its training set and 10,000 images in its testing set. Only the first 6,000 data points will be used for the training in our experiments. The implementation can easily be extended to use the EMNIST dataset [12] which also includes images of handwritten letters.

A. Sequential implementation

The sequential implementation is executed conventionally, starting by reading the training and test data from csv files in the format $(y, x_1, ..., x_{784})$ where y is the digit that corresponds to the image represented by the pixels denoted by x_i where each digit x_i holds values from 0 to 255 representing the amount of the color black in the pixel, 0 being all white and 255 being all black.

The dataset will be split into training and test data sets, The training subset will be used to train a vast variety of classifiers, each having a different number of neurons in their hidden layer, and each being run with a different number of epochs. The number of neurons in the hidden layer and the number of epochs will vary in the interval from 50 to 100. The test set will be used to determine which of the classifiers has the most optimal accuracy to training time ratio.

The goal is to find a NN model that is easy and fast to train but makes the most accurate predictions possible. Note that sometimes the most accurate classifier is not the best choice because it may take longer to train than it's worth. For this reason, the actual classifier that will be used on the parallel implementation will be the classifier which has most optimal accuracy to training time ratio.

B. Parallel implementation

In development of a parallel implementation, the goal is to take the training set and distribute it evenly among the processes in the pool. The master process, usually the one with $rank = 0$ is responsible to read the data, to determine the number of training examples per process and to scatter the data among the other processes. After each process has their share of the training data, they can begin the training independently of each other. First they initialize their own classifier. Afterwards, each process begins fitting training its NN to its respective subset of the training data.

When the processes are finished, they need to send their weight matrices to the master process whose job is to compute an average for each respective element in the different weight matrices, and produces an all encompassing weight matrix for the neural network. The master process then uses the test data to predict the class of each test example, comparing it with the real class and thus, determining the accuracy of the neural network.

The main tool that will help us write the parallel algorithm is the Python library $mpi4py$ that offers the means necessary to write a parallel program using the Message Passing Interface.

C. Experimental environment

The solution is implemented in such a way that it can be executed on any multiple processor configuration, so long as the data can be split evenly among the processes the algorithm is started with. For example, our training data consists of 6,000 training examples, meaning that the program should be started with a number of processes n, such that 6,000 is divisible by n. For our testing purposes, we will use a system with an i7-9750H six core CPU and 16GB of RAM. The algorithm will be run three times, with $n = 2$, $n = 4$ and $n = 6$ processes respectively.

Out of the family of classifiers described in the sequential implementation, we will select the optimal classifier to use for the parallel implementation. That classifier should provide a good balance between the accuracy of the predictions, and the time it takes to train.

To asses the value of our exercise in parallelization, we introduce an $impact\_factor$ parameter, defined by (1).

$$impact\_factor = \frac{prediction\_accuracy * s_1}{training\_time * s_2} \qquad (1)$$

The parameters s_1 and s_2 represent significance factors. For the purpose of this paper, we will assume equal importance between $prediction\_accuracy$ and $training\_time$ by setting $s_1 = s_2 = 1$. This is not always the case in practice. Usually, the trade off between time and accuracy will vary from situation to situation and so the significance parameters would have to be set accordingly. The $impact\_factor$ parameter defined as such, can provide a numerical representation of the benefit regarding the parallelization of the neural network algorithm.

The $impact\_factor$ parameter defined as such, provides a numerical representation of the benefit regarding the parallelization of the neural network algorithm.

V. Results

A. Sequential implementation

Running the script that contains the sequential algorithm, we observe the time it takes to train all of the individual classifiers.

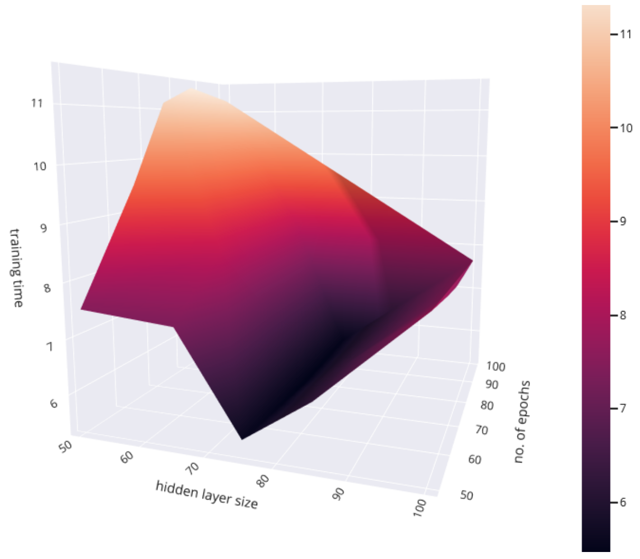


Fig. 3. A 3D mesh presenting the training time with respect to the hidden layer size and number of epochs

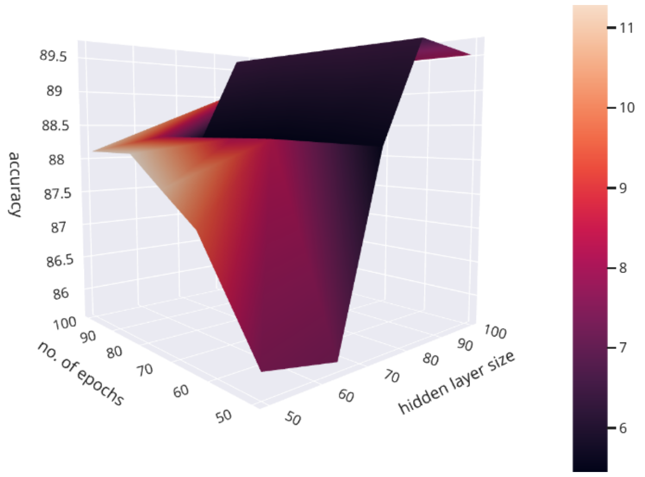


Fig. 4. A 3D mesh presenting the accuracy with respect to the hidden layer size and number of epochs

Fig. 3 presents a mesh of all the classifiers and how the training time reacts almost linearly to different hidden layer sizes and different epoch numbers. Note that the results are not exactly linear due to the nature of how operating systems schedule tasks. Each time the code is executed, the measured time will be different. We conducted five test runs and calculated an average.

When it comes to the accuracy of predicting the test set, the same type of mesh is given in Fig. 4, where the accuracy reacts very differently to epoch numbers and hidden layer sizes, from the training time. As we add more neurons to a classifier's hidden layer and we increase the number of epochs, we can surmise that the time it takes to train it isn't worth the negligible increase in accuracy we get. The numerical representation of these results is given in Table I.

The defined impact factor parameter shows that the best

TABLE I
IMPACT FACTORS FOR DIFFERENT MLP CLASSIFIERS

Hidden layer	Epochs	Time	Accuracy	Impact
50	50	7,2142	85,8985	11,9068
50	65	9,2273	87,2587	9,4564
50	75	10,5781	87,6787	8,2886
50	85	11,0793	88,1088	7,9525
50	100	11,324	88,1088	7,7806
65	50	7,5223	85,6985	11,3925
65	65	8,3173	88,3188	10,6186
65	75	8,2403	88,3188	10,7178
65	85	8,2412	88,3188	10,7167
65	100	8,1574	88,3188	10,8267
75	50	5,4276	88,2288	16,2552
75	65	5,7881	88,2288	15,2429
75	75	5,3966	88,2288	16,3487
75	85	5,5618	88,2288	15,8633
75	100	5,5589	88,2288	15,8714
85	50	6,4337	89,6289	13,9311
85	65	6,5464	89,6289	13,6911
85	75	6,2227	89,6289	14,4034
85	85	6,353	89,6289	14,108
85	100	6,275	89,6289	14,2834
100	50	8,2521	89,4989	10,8454
100	65	8,0113	89,4989	11,1715
100	75	7,8782	89,4989	11,3603
100	85	8,0304	89,4989	11,1449
100	100	8,0716	89,4989	11,088

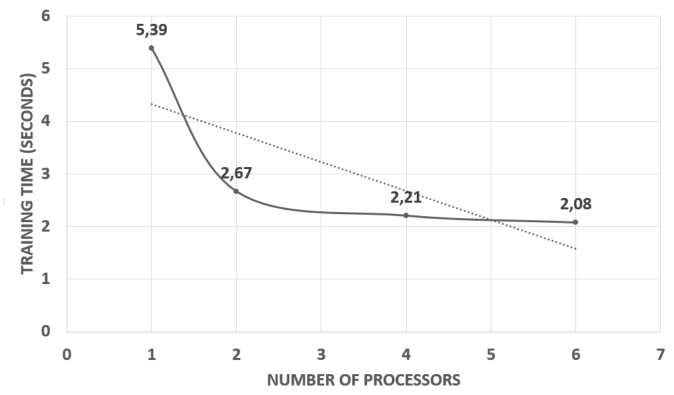


Fig. 5. Training time for the sequential and parallel algorithms

trade-off in terms of accuracy and training time is accomplished using a classifier with 75 neurons in its hidden layer, with a number of 75 epochs. Such a classifier gives a reasonably good accuracy of 88.23 percent using about 5.39 seconds to train, and therefore, it is used in our parallel implementation, to test if and how much of an increase in speed we can gain, whilst keeping the accuracy in check.

B. Parallel implementation

With the parallel algorithm, interesting details come to light about the speed with which the neural network is trained. We are using a classifier identical to the optimal classifier from the sequential implementation (recall that the optimal classifier had 75 neurons in the hidden layer and a maximum of 75 epochs). The change in training time is presented visually in Fig. 5. Our parallel program using 2 cores managed to train the neural network in less than half of that of the sequential

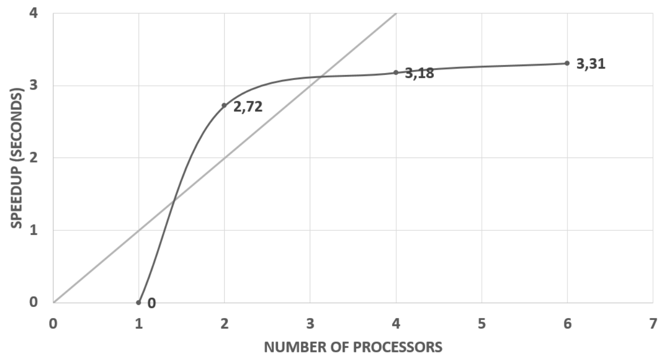


Fig. 6. Speedup versus different numbers of processes

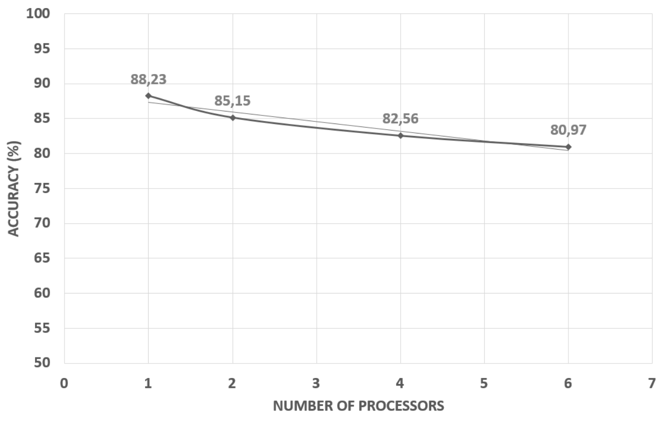


Fig. 7. Prediction accuracy for the sequential and parallel algorithms

algorithm. Using 4 cores, the increase in speed from 2 to 4 isn't as significant as that from 1 to 2, and the increase from 4 to 6 even less so. Increasing the number of processors, will cause negligible increase in speed. The reason for this is the nature of Amdahl's Law [13] (Fig. 6), which concludes that the speedup is limited by the sequential part of the algorithm and not by the number of processors. With all of that taken into account, these results, presented in Table II, are still an improvement over the original sequential implementation.

The same can not be said, however, for the accuracy of the predictions. Although not bad by any means, Fig. 7 presents a slight drop-off from the sequential algorithm's 88%, to this parallel algorithm's 85%, 82% and 80% accuracy, using 2, 4 and 6 cores respectively. This result is in line with the hypothesis we presented in the introduction.

The $impact\_factor$ parameter determines the usefulness of our experiment. Table II also presents the impact factor for each of the executions with 1, 2, 4 and 6 cores. The sequential algorithm's $impact\_factor = 16.37$ while for the parallel algorithm running on 2 cores $impact\_factor = 31.89$, for 4 cores $impact\_factor = 37.36$ and for 6 cores $impact\_factor = 38.93$.

Even though the accuracy of our predictions went down using the parallel training, the higher impact of the parallel implementation tells us that the experiment was indeed worth-

TABLE II
IMPACT FACTORS FOR DIFFERENT NUMBERS OF PROCESSES - NO COMMUNICATION OVERHEAD

No. of processes	Training time	Accuracy	Impact factor
1	5,39	88.23	16.369
2	2.67	85.15	31.891
4	2.21	82.56	37.357
6	2.08	80.97	38.928

TABLE III
IMPACT FACTORS FOR DIFFERENT NUMBERS OF PROCESSES - WITH COMMUNICATION OVERHEAD

No. of processes	Training time	Accuracy	Impact factor
1	5,39	88.23	16.369
2	13.67	85.15	6.229
4	13.21	82.56	6.250
6	13.08	80.97	6.190

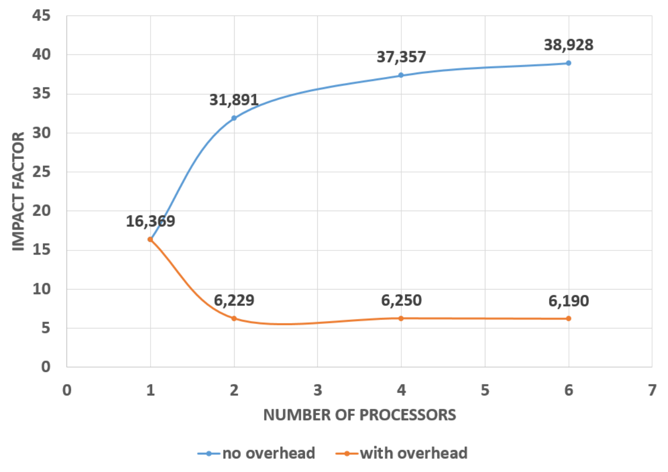


Fig. 8. The change of the Impact Factor for different numbers of processes

while. We still however don't get a clear enough picture of the impact of our solution, until we take into consideration the communication overhead that occurs during the parallel execution. After the master process reads the data, it needs to distribute it evenly among the other processes. Scattering the data takes around 11 seconds, and so, when we inject that into the calculation, Table III and Fig. 8 give much greater insight. The 2 core parallel $impact\_factor = 6.23$ which is a significant drop-off from the previously calculated 31.89. A minor increase is observed for the 4 core implementation ($impact\_factor = 6.25$), while for the 6 core implementation, the impact factor actually decreases after taking the scattering time into account ($impact\_factor = 6.19$). This leads to the conclusion that after a certain point, parallelizing a neural network stops being useful.

VI. CONCLUSION

As with everything in life, every benefit comes with a price. In our case we wanted to provide an increase in speed for the training process of the neural network intended for predicting handwritten digits. Although we achieved that speed up, it came at the price of accuracy with the predictions. In the end, these results prove that the hypothesis we presented in the introduction can be considered as satisfied.

The algorithm of training neural networks is sequential in its nature, so the end results make sense taking everything into consideration. The experiment provided us with an interesting insight into the world of neural networks and parallel computing and how, sometimes, you can't have your cake and eat it too. In order to accomplish something, some sacrifices also need to be made, whether those be complexity, resources, or accuracy.

This experiment proves useful as a jumping-off point for further research. Priorities in said research would include refining the computing of the final weight matrix of the trained NN, rather than just averaging the weight matrices of the separate NNs trained in the individual processes. We would also like to explore finer grained solutions where the processes communicate more often (during or after each epoch, rather than just at the end of the training to computer the weight matrix).

Lastly something that would likely provide an interesting read, is the exploration of different ML algorithms. Perhaps Naive Bayes classifiers, Decision trees, Linear and Quadratic discriminant analysis (LDA and QDA), etc. could prove themselves to be better suited for parallel execution than NNs.

REFERENCES

[1] M. Pethick, M. Liddle, P. Werstein, and Z. Huang, "Parallelization of a backpropagation neural network on a cluster computer," in *International conference on parallel and distributed computing and systems (PDCS 2003)*, 2003.
[2] R. Hecht-Nielsen, "Theory of the backpropagation neural network," in *Neural networks for perception*. Elsevier, 1992, pp. 65–93.
[3] Y. Pao, *Adaptive pattern recognition and neural networks*. Reading, MA (US); Addison-Wesley Publishing Co., Inc., 1989.
[4] A. Graves and J. Schmidhuber, "Offline handwriting recognition with multidimensional recurrent neural networks," *Advances in neural information processing systems*, vol. 21, pp. 545–552, 2008.
[5] A. Graves, S. Fernández, F. Gomez, and J. Schmidhuber, "Connectionist temporal classification: labelling unsegmented sequence data with recurrent neural networks," in *Proceedings of the 23rd international conference on Machine learning*, 2006, pp. 369–376.
[6] V. Kumar, A. Grama, A. Gupta, and G. Karypis, *Introduction to parallel computing*. Benjamin/Cummings Redwood City, CA, 1994, vol. 110.
[7] I. Foster, *Designing and building parallel programs*. Addison-Wesley, 2003.
[8] S. G. Aki, *The design and analysis of parallel algorithms*. Old Tappan, NJ (USA); Prentice Hall Inc., 1989.
[9] A. H. Kramer and A. Sangiovanni-Vincentelli, "Efficient parallel learning algorithms for neural networks," in *Advances in neural information processing systems*, 1989, pp. 40–48.
[10] R. Rogers and D. Skillicorn, "Strategies for parallelizing supervised and unsupervised learning in artificial neural networks using the bsp cost model," *Queens University, Kingston, Ontario, Tech. Rep*, 1997.
[11] Y. LeCun and C. Cortes, "MNIST handwritten digit database," 2010. [Online]. Available: http://yann.lecun.com/exdb/mnist/
[12] G. Cohen, S. Afshar, J. Tapson, and A. Van Schaik, "Emnist: Extending mnist to handwritten letters," in *2017 International Joint Conference on Neural Networks (IJCNN)*. IEEE, 2017, pp. 2921–2926.
[13] G. M. Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities," in *Proceedings of the April 18-20, 1967, spring joint computer conference*, 1967, pp. 483–485.