SPMD-based Neural Network Simulation with Golang

Daniela Kalwarowskyj and Erich Schikuta[0000−0002−4126−4243]

University of Vienna
Faculty of Computer Science, RG WST
A-1090 Vienna, Währingerstr. 29, Austria
dkalwarowskyj@yahoo.com
erich.schikuta@univie.ac.at

Abstract. This paper describes the design and implementation of parallel neural networks (PNNs) with the novel programming language Golang. We follow in our approach the classical Single-Program Multiple-Data (SPMD) model where a PNN is composed of several sequential neural networks, which are trained with a proportional share of the training dataset. We used for this purpose the MNIST dataset, which contains binary images of handwritten digits. Our analysis focusses on different activation functions and optimizations in the form of stochastic gradients and initialization of weights and biases. We conduct a thorough performance analysis, where network configurations and different performance factors are analyzed and interpreted. Golang and its inherent parallelization support proved very well for parallel neural network simulation by considerable decreased processing times compared to sequential variants.

Keywords: Backpropagation Neuronal Network Simulation · Parallel and Sequential Implementation · MNIST · Golang Programming Language

1 Introduction

When reading a letter our trained brain rarely has a problem to understand its meaning. Inspired by the way our nervous system perceives visual input, the idea emerged to write a mechanism that could "learn" and furthermore use this "knowledge" on unknown data. Learning is accomplished by repeating exercises and comparing results with given solutions. The neural network studied in this paper uses the MNIST dataset to train and test its capabilities. The actual learning is achieved by using backpropagation. In the course of our research, we concentrate on a single sequential feed forward neural network (SNN) and upgrade it into building multiple, parallel learning SNNs. Those parallel networks are then fused to one parallel neural network (PNN). These two types of networks are compared on their accuracy, confidence, computational performance and learning speed, which it takes those networks to learn the given task.

The specific contribution of the paper is twofold: on the one hand, a thorough analysis of sequential and parallel implementations of feed forward neural network respective time, accuracy and confidence, and on the other hand, a feasibility study of Golang [8] and its tools for parallel simulation.

2 Related Work and Baseline Research

In the literature a huge number of papers on parallelizing neural networks can be found. An excellent source of references is the survey by Tal Ben-Nun and Torsten Hoefler [1]. However, only few research was done on using Golang in this endeavour.

In the course of our work on parallel and distributed systems [11,2,9] we developed several approaches for the parallelization of neural networks. In [6], two novel parallel training approaches were presented for face recognizing back-propagation neural networks. Further, we differentiate between topological data parallelism and structural data parallelism [10], where the latter is focus of the presented approach here.

3 Parallel Neuronal Networks

Go, often referred to as Golang [8], is a compiled, statically typed, open source programming language developed by a team at Google and released in November 2009. It is distributed under a BSD-style license, meaning that copying, modifying and redistributing is allowed under a few conditions.

Built-in support for concurrency is one of the most interesting aspects of Go, offering a great advantage over older languages like C++ or Java. One major component of Go's concurrency model are goroutines, which can be thought of as lightweight threads with a negligible overhead, as the cost of managing them is cheap compared to threads. If a goroutine blocks, the runtime automatically moves any blocking code away from being executed and executes some code that can run, leading to high-performance concurrency [8]. Communication between goroutines takes place over channels, which are derived from "Communicating Sequential Processes" found in [5]. A Channel can be used to send and receive messages from the type associated with it. Since receiving can only be done when something is being sent, channels can be used for synchronization, preventing race conditions by design.

Another difference to common object oriented programming languages can be found in Go's object oriented design. Its approach misses classes and type-based inheritance like subclassing, meaning that there is no type hierarchy. Instead, Go features polymorphism with interfaces and struct embedding and therefore encourages the composition over inheritance principle.

For the parallelization of neural network operations we apply the classical Single-Program Multiple-Data (SPMD) approach well known from high-performance computing [3]. It is a programming technique, where several tasks execute the same program but with different input data and the calculated output data is merged to a common result. Thus, based on the fundamentals of single feed forward neural network we generate multiple of these networks and set them up to work together in parallel manner.

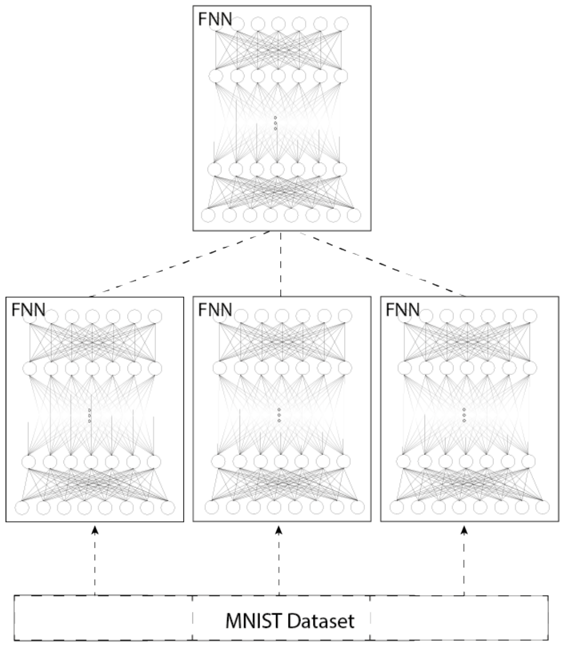

Fig. 1. Design of a Parallel Neural Network

The parallel-design is visualized in figure 1. On the bottom it shows the dataset which is divided into as many slices as there are networks, referred to as child-networks (CN). Each child-network learns only a slice of the dataset. Ultimately the results of all parallel child-networks are merged to one final parallel neural network (PNN). The combination of those CNs can be done in various ways. In the presented network the average of all weights, calculated by each parallel CN by a set number of epochs, is used for the PNNs weights. For the biases the same procedure is used, e.g. averaging all biases for the combined biases value.

4 Performance Evaluation

For our analysis, we use the MNIST dataset which holds handwritten numbers and allows supervised learning. Using this dataset the network learns to read handwritten digits. Since learning is achieved by repeating a task, the MNIST dataset has a "training-set of 60,000 examples, and a test-set of 10,000 examples" [7] . Each dataset is composed of an image-set and a label-set, which holds the information for the desired output and makes it possible to verify the networks output. All pictures are centered and uniform by 28x28 pixels. First, we start the training with the training-set. When the learning phase is over the network is supposed to be able to fulfill its task . To evaluate it's efficiency it is tested by running the neural network with the test-set since the samples of this set are still unknown. It is important to use foreign data to test a network since it is more qualified to show the generalization of a network and therefore its true efficiency. We are aware that MNIST is a rather small data set. However, it was chosen on purpose, because it is used in many similar parallelization approaches and allows therefore for relatively easy comparison of results.

4.1 Network Configurations

Number of Neurons. Choosing an efficient number of neurons is important, but it is hard to identify. There is no calculation which helps to define an effectively working number or range of neurons for a certain configuration of a neural network. Varying the number of neurons between 20 to 600 delivered great accuracy.

Number of Networks. To evaluate the performance of PNNs in terms of accuracy, PNNs with different amounts of CNs are composed and trained. The training runs over 20 epochs with a learning rate of 0.1 and a batchsize of 50. All CNs are built with one hidden layer consisting of 256 neurons. On the hidden layer the ReLU-function and on the output layer the Softmax-function is used. After every epoch, the networks are tested with the test-dataset. The results are visualized in figure 2.

Figure 2 illustrates a clear loss in accuracy of PNNs with a growing number of CNs. The 94.5% accuracy, for example, is reached by a PNN with 2 CNs after only one epoch, while a PNN with 30 CNs achieves that after 12 epochs.

Since the provided PNNs are built by using averaging of weights and biases it also seemed interesting to compare the average accuracy of the CNs with the resulting PNN, to grade the used combination function. The results are illustrated in figure 3.

It shows that the efficiency of an average function grows with the number of CNs. The first graph drawn with 2 CNs shows, that the resulting PNN is performing worse than the average of the CNs, it has been built from. By growing the number of CNs to 10, the average of CNs approximates towards the PNN. The last graph of this figure shows that a PNN composed of 20 CNs outperforms the average of its CNs after 200 epochs, and after 300 epochs levels with it. It has to be noted that the differences in accuracy are very small, as it is only a range of 0.1 to 0.2 percent. Overall it can be said that this combination function is working efficiently.

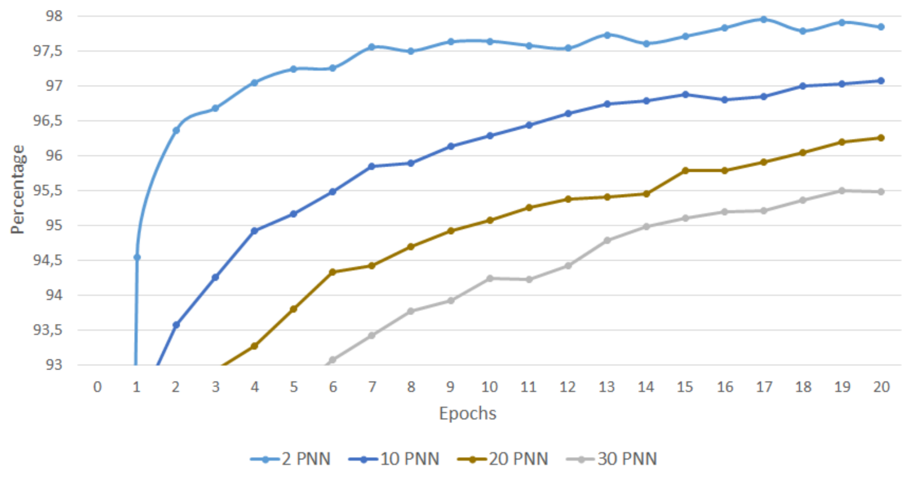

Fig. 2. Accuracy of PNNs, built with different amount of CNs, over 20 epochs

4.2 Comparing the Performances

Time. Time is the main reason to have a network working in parallel. To test the effect of parallelism on the time required to train a PNN, the provided neuronal network is tested on three systems. The first system is equipped with 4 physical and 4 logical cores, an Intel i7-3635QM processor working with a basic clock rate of 2.4GHz, the second system holds 6 physical cores and 6 logical cores working with 2.9GHz and an Intel i9-8950HK processor and last the third system works with an AMD Ryzen Threadripper 1950X with 16 physical and 16 logical cores, which work with a clock rate of 3.4GHz. The first, second and third systems are referred to as 4 core, 6 core and 16 core in the following.

In figure 4 the benefit in terms of time using parallelism is clearly visible. The results illustrated show the average time in seconds needed by each system for training a PNN consisting of one CN per goroutine.

The time in figure 4 starts on a high level and decreases with an increasing amount of goroutines for all three systems. Especially in the range of 1 to 4 goroutines, a formidable decrease in training time is visible and only starts to level out when reaching a systems physical core limitation. This means that the 4 core starts to level out after 4 goroutines, the 6 core after 6 goroutines and the 16 core after 16 goroutines, even though all systems support hyper threading. After reaching a systems core number the average time necessary for training a neural network decreases further with more goroutines. This should be due to the ability to work in parallel and in concurrency as one slot finishes and a waiting thread can start running immediately, without waiting for the rest of the running threads to be finished. All three systems show high time savings by parallelizing the neural networks. While time requirements decreased in every system, the actual time savings differ greatly as the 16 core system decreased

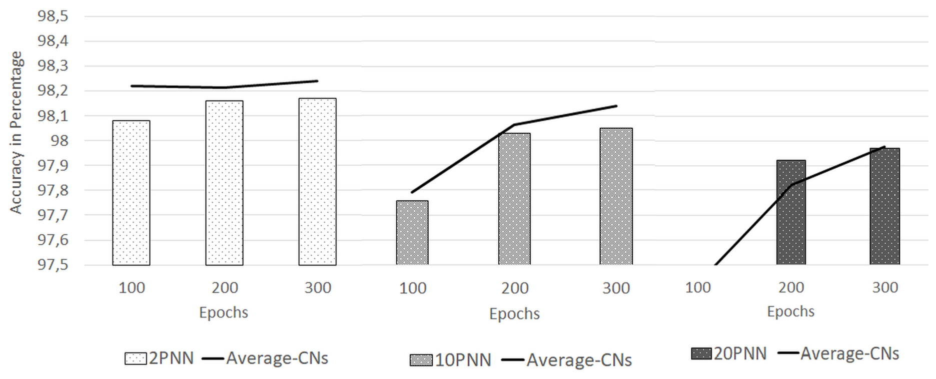

Fig. 3. Comparison of the average accuracy of all CNs, out of which the final PNN is formed, with PNNs accuracy

91 percent on average from 1 goroutine to 64 goroutines. In comparison, the 4 core system only took 65 percent less time. As the 16 core system is a lot more powerful than the 4 core system, it can perform an even greater parallel task and therefore displays a positive effect of parallelism upon time requirements.

Accuracy and Confidence of Networks. In this section the performance in terms of accuracy and confidence is compared between a PNNs and an SNN. For the test, illustrated by figure 5, both types of networks have been provided with the same random network to start their training. They have the exact same built, except that one is trained as SNN and the other is cloned 10 times to build a PNN with 10 CNs.

In figure 5 the SNN performs better than the PNN in both accuracy and confidence. While the SNNs accuracy and confidence overlap after 8 epochs, the PNN has a gap between both lines at all times. This concludes that the SNN is "sure" about its outputs, while the PNN is more volatile. The SNNs curve of confidence is a lot steeper than the PNNs and quickly approximates towards the curve of accuracy. Both curves of accuracy start off almost symmetric upwards the y-axis, but the PNN levels horizontally after about 90 percent while the SNN still rises until about 94 percent. After those points both accuracy curves run almost horizontally and in parallel towards the x-axis. The gap stays constantly until the end of the test. Even small changes within the range of 90 to 100 percent are to be interpreted as significant.

5 Findings and Conclusion

This paper presents and analyses PNNs composed of several sequential neural networks. The PNNs are tested upon time and accuracy and compared to an

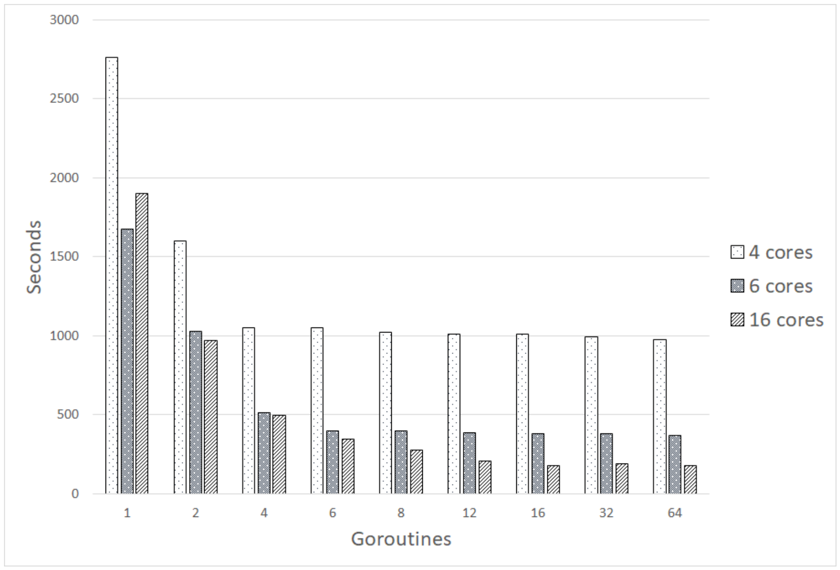

Fig. 4. Time in seconds, that was needed to train a PNN with a limited amount of one Goroutine per composed CN.

SNN. Summing up, PNNs proved to be very time efficient but are still lacking in terms of accuracy. As there are plenty of other optimizations, e.g. adjusting learning rates [4], a PNN proved to be more time efficient than an SNN. However, until the issue of accuracy has been taken care of, the SNN surpasses the PNN in practice.

We close the paper with a final word on the feasibility of Golang for parallel neural network simulation: Data parallelism proved to be an efficient parallelization strategy. In combination with the programming language Go, a parallel neural network implementation is coded as fast as a sequential one, as no special efforts are necessary for concurrent programming thanks to Go's concurrency primitives, which offer a simple solution for multithreading.

References

1. Ben-Nun, T., Hoefler, T.: Demystifying parallel and distributed deep learning: An in-depth concurrency analysis. ACM Computing Surveys (CSUR) **52**(4), 1–43 (2019)
2. Brezany, P., Mueck, T.A., Schikuta, E.: A software architecture for massively parallel input-output. In: Waśniewski, J., Dongarra, J., Madsen, K., Olesen, D. (eds.) Applied Parallel Computing Industrial Computation and Optimization. pp. 85–96. Springer Berlin Heidelberg, Berlin, Heidelberg (1996)

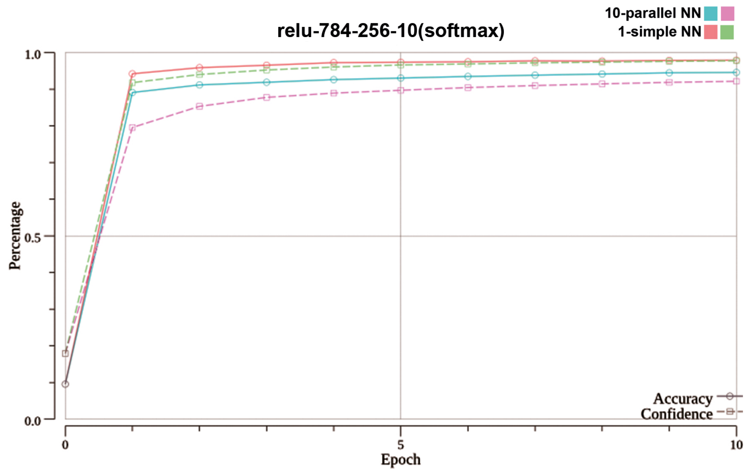

Fig. 5. Compare Accuracy and Confidence of a PNN composed of 10 CNs and an SNN with one Hidden Layer which holds 256 Neurons

3. Darema, F.: The spmd model: Past, present and future. In: European Parallel Virtual Machine/Message Passing Interface Users' Group Meeting. pp. 1–1. Springer (2001)
4. Goyal, P., Dollár, P., Girshick, R., Noordhuis, P., Wesolowski, L., Kyrola, A., Tulloch, A., Jia, Y., He, K.: Accurate, large minibatch sgd: Training imagenet in 1 hour. arXiv preprint arXiv:1706.02677 (2017)
5. Hoare, C.A.R.: Communicating sequential processes. In: The origin of concurrent programming, pp. 413–443. Springer (1978)
6. Huqqani, A.A., Schikuta, E., Ye, S., Chen, P.: Multicore and gpu parallelization of neural networks for face recognition. Procedia Computer Science **18**(Supplement C), 349 – 358 (2013), 2013 International Conference on Computational Science
7. LeCun, Y., Bottou, L., Bengio, Y., Haffner, P.: Gradient-based learning applied to document recognition. Proceedings of the IEEE **86**(11), 2278–2324 (1998)
8. Meyerson, J.: The go programming language. IEEE Software **31**(5), 104–104 (Sept 2014)
9. Schikuta, E., Weishaupl, T.: N2grid: neural networks in the grid. In: 2004 IEEE International Joint Conference on Neural Networks (IEEE Cat. No.04CH37541). vol. 2, pp. 1409–1414 vol.2 (2004)
10. Schikuta, E.: Structural data parallel neural network simulation. In: Proceedings of 11th Annual International Symposium on High Performance Computing Systems (HPCS'97), Winnipeg, Canada (1997)
11. Schikuta, E., Fuerle, T., Wanek, H.: Vipios: The vienna parallel input/output system. In: Pritchard, D., Reeve, J. (eds.) Euro-Par'98 Parallel Processing. pp. 953–958. Springer Berlin Heidelberg, Berlin, Heidelberg (1998)