Go for Parallel Neural Networks

David Turner(✉) and Erich Schikuta(✉)

Faculty of Computer Science, University of Vienna, RG WST, 1090 Vienna, Austria
davidturner@gmx.at, erich.schikuta@univie.ac.at

Abstract. Training artificial neural networks is a computationally intensive task. A common and reasonable approach to reduce the computation time of neural networks is parallelizing the training. Therefore, we present a data parallel neural network implementation written in Go. The chosen programming language offers built-in concurrency support, allowing to focus on the neural network instead of the multi-threading. The multi-threaded performance of various networks was compared to the single-threaded performance in accuracy, execution time and speedup. Additionally, two alternative parallelization approaches were implemented for further comparisons. Summing up, all networks benefited from the parallelization in terms of execution time and speedup. Splitting the mini-batches for parallel gradient computation and merging the updates produced the same accuracy results as the single-threaded network. Averaging the parameters too infrequently in the alternative implementations had a negative impact on accuracy.

Keywords: Neural network simulation · Backpropagation · Parallelization · Go programming language

1 Introduction

Parallelization is a classic approach for speeding up execution times and exploiting the full potential of modern processors. Thus, we present in this paper a data parallel implementation of the training phase of artificial neural networks and show the feasibility of the Go language as implementation framework. Go is a fairly new programming language that makes concurrent programming easy. With the optimal number of threads, Go programs can fully exploit modern multi-core processors. Still, not every algorithm has to profit from multi-core execution, as parallel execution might add a non-negligible overhead. This can also be the case for data parallel neural networks, where accuracy problems usually occur, as the results have to be merged. The authors of [5] present a parallel backpropagation algorithm additionally dealing with the accuracy problem by using a MapReduce and Cascading model. In [3], two novel parallel training approaches are presented for face recognizing backpropagation neural networks. The authors use the OpenMP environment for classic CPU multithreading and CUDA for parallelization on GPU architectures. Aside from that, they differentiate between structural data parallelism and topological data parallelism.

Most work will just differentiate between data parallelism and model parallelism. [8] offers a comparison of different parallelization approaches on a cluster computer. The results differ depending on the network size, data set sizes and number of processors. Besides parallelizing the backpropagation algorithm for training speed up, alternative training algorithms like the Resilient Backpropagation described in [10] might lead to faster convergence. One major difference to standard backpropagation is that every weight and bias has a different and variable learning rate. A detailed comparison of both network training algorithms is given in [9] in the case of spam classification. The authors conclude, that resilient backpropagation can be a promising choice for training neural networks for time-sensitive machine learning applications.

The paper is structured as follows: In the next section we introduce the Go programming language and its characteristics. In Sect. 3 we lay out the fundamentals and techniques of the artificial neural network model and our parallelization approach. Accuracy, performance and speedup analysis is presented in Sect. 4. The lessons learned form our approach are listed in Sect. 5. The paper closes with the conclusion of our work done.

2 The Go Programming Language

Go, often referred to as Golang, 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. It is designed to be expressive, concise, clean and efficient [1]. Hence, Go compiles quickly and is as easy to read as it is to write. That being said, 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 runnable code, leading to high-performance concurrency [7]. Communication between goroutines takes place over channels, which are derived from "Communicating Sequential Processes" found in [2]. 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.

3 Fundamentals

Artificial Neural Networks are networks made of artificial neurons. The first artificial neuron was invented in 1943 by Warren McCulloch and Walter Pitts, known as the McCulloch-Pitts-Neuron and described in [6]. It is the simplest form of an artificial neuron, only accepting binary input. If the summed input is larger than a certain threshold value, the neuron's output is 1, else 0. Today's neurons have weighted, real number inputs and use an activation function instead of a threshold value. Although many different kinds of ANNs exist, artificial neurons are typically organized in layers. The first and the last layer are also called input and output layer. If the signal in a network moves in only one direction, that is to say from the input to the output layer, the network is called a *feedforward neural network*, also known as *multilayer perceptron (MLP).*

Forwardpropagation. To calculate an output in the last layer, the input values need to get propagated through each layer. This process is called forward propagation and is done by applying an activation function on each neuron's corresponding input sum. The input sum z for a neuron k in the layer l is the sum of each neuron's activation a from the last layer multiplied with the weight w:

$$z_k^l = \sum_j (w_{kj}^l a_j^{l-1} + b_k^l) \tag{1}$$

Backpropagation. For proper classification the network obviously has to be trained beforehand. In order to do that, a cost function, telling us how well the network performs, like the cross entropy error with expected outputs e and actual outputs x,

$$C = -\sum_i e_i log(x_i) \tag{2}$$

has to be defined. The aim is to minimize the cost function by finding the optimal weights and biases with the gradient descent optimization algorithm. Therefore, a training instance gets forward propagated through the network to get an output. Subsequently, it is necessary to compute the partial derivatives of the cost function with respect to each weight and bias in the network:

$$\frac{\partial C}{\partial w_{kj}} = \frac{\partial C}{\partial z_k} \frac{\partial z_k}{\partial w_{kj}} \tag{3}$$

$$\frac{\partial C}{\partial b_k} = \frac{\partial C}{\partial z_k} \frac{\partial z_k}{\partial b_{kj}} \tag{4}$$

As a first step, $\frac{\partial C}{\partial z_k}$ needs to be calculated for every neuron k in the last layer L:

$$\delta_k^L = \frac{\partial C}{\partial z_k^L} = \frac{\partial C}{\partial x_k^L} \varphi'(z_k^L) \tag{5}$$

In case of the cross entropy error function, the error signal vector δ of the softmax output layer is simply the actual output vector minus the expected output vector:

$$\delta^L = \frac{\partial C}{\partial z^L} = x^L - e^L \tag{6}$$

To obtain the errors for the remaining layers of the network, the output layer's error signal vector δ^L has to be back propagated through the network, hence the name of the algorithm:

$$\delta^l = (W^{l+1})^T \delta^{l+1} \odot \varphi'(z^l) \tag{7}$$

$(W^{l+1})^T$ is the transposed weight matrix, $\odot$ denotes the Hadamard product or entry-wise product and φ' is the first derivative of the activation function.

Gradient Descent. Knowing the error of each neuron, the changes to the weights and biases can be determined by

$$\Delta w^l_{kj} = -\eta \frac{\partial C}{\partial w^l_{kj}} = -\eta \delta^l_k x^{l-1}_j \tag{8}$$

$$\Delta b^l_k = -\eta \frac{\partial C}{\partial b_k} = -\eta \delta^l_k \tag{9}$$

The constant η is used to regulate the strength of the changes applied to the weights and biases and is also referred to as the learning rate, x^{l-1}_j stands for the output of the j^{th} neuron from layer $l-1$. The changes are applied by adding them to the old weights and biases.

3.1 Parallelism

A fundamental distinction is made between two parallelization techniques. The approach to parallelism used in the implementation of this work is often referred to as data parallelism, since all threads or goroutines train identical networks concurrently on a different subset of the data, see Fig. 1. This stands in contrast to model parallelism, where the model gets split instead of the data. While model parallelism produces better results on very large neural networks, data parallelism profits from large data sets [8] and performs better on networks where all weights fit into memory.

The difficulty with data parallelism is the combination of the parameters computed by each network copy. One way to merge the weights and biases is by averaging them after each thread finished network training on its subset of the training data. However, this is not recommended, as each individually trained network might find a different optimum with diverging weights and biases. In that case, the combined neural network could perform even worse on the whole data set than the individual networks, which were trained only on a subset of the training examples. Better accuracy can be achieved by synchronizing the individual networks from time to time during training. Therefore, the parameters of all neural network copies are averaged periodically, i.e. after a specified number of mini-batches or after a few minutes. The resulting parameters are then broadcasted back to the copies.

Another technique is merging the parameter updates instead of the weights and biases themselves. This can happen asynchronously or synchronously. The asynchronous variant has an increased speedup, as the threads do not wait for each

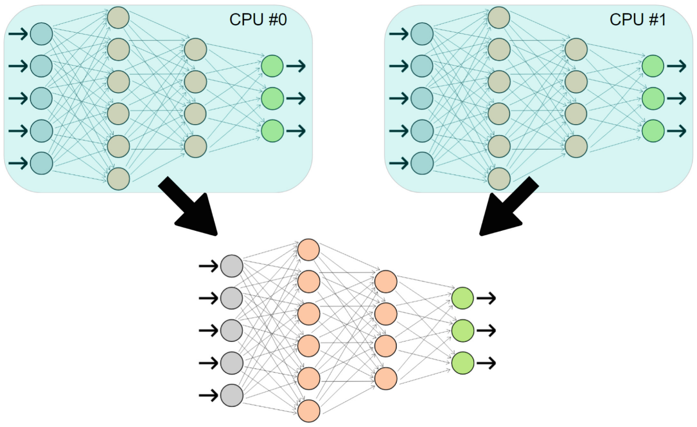

Fig. 1. Data parallelism. Identical networks train on different subsets of the data

other. In each iteration, a network instance requests the current parameters from a separate thread, computes the gradients for a mini-batch and communicates them to the parameter thread, which applies the updates [12]. By the time one thread finishes its mini-batch, the parameters might have already been updated several times. This is called gradient staleness and usually results in accuracy loss, as the computed gradient is no longer parallel to the steepest descent direction at the current weights and biases [11]. The implementation presented in the following section is a synchronized version. Each thread computes the gradients for a subset of the mini-batch. When all gradients of the mini-batch are computed by the network copies and accumulated, one copy updates its weights and biases accordingly and broadcasts its new parameters to the other copies. Splitting the mini-batches and synchronizing after every mini-batch update produces exactly the same accuracy results of the network as the sequential algorithm would do. This comes at the cost of a decreased speedup because of the frequent synchronization. Evidently, the algorithms speedup profits from larger mini-batch sizes.

4 Performance Evaluation

4.1 Data Set

The chosen MNIST database contains a training set of 60000 examples and a test set of 10000 examples of handwritten digits [4]. It is a widely used data set for benchmarking classification algorithms. The training data was further split into 50000 training examples and 10000 validation examples. The network trains on the 50000 examples, while the evaluation and tuning of the parameters use the validation set for better generalization. The final performance evaluation is done on the test set.

4.2 Alternative Implementations

After 30 epochs on a training set with 50000 examples, a learning rate of 0.25 and a mini-batch size of 256, a network with three hidden layers (440-240-120) classified 98.31% of the test set correctly. For validation a separate subset of the MNIST training set comprising 10000 examples was used, meaning that parameter finding and training of the network was done completely without the test set. Still, the frequent synchronization can be a bottleneck in terms of execution time. Therefore, two further versions were implemented to measure the performance loss resulting from it. The first alternative implementation synchronizes after every epoch simply by averaging the weights and biases, the second one averages the weights and biases after the specified number of training epochs. Unlike the first two implementations, the last one mentioned shuffles the data only once before training instead of every epoch, meaning that the network will train in the same order and on the same batches for the configured number of epochs.

4.3 Environment

The tests were carried out on three different systems. The first system has a dual Intel Xeon X5570 quad-core processor setup and 24 GB memory. The processors work with a basic clock rate of 2,93 GHz per physical core and support hyper-threading (thus 16 logical cores in total). Each processor has an 8 MB L3-Cache. The second system utilizes an AMD Ryzen Threadripper 1950X hexadeca-core processor and 32 GB memory. The processor comprises 32 logical cores, a 32 MB L3-Cache and works with a basic clock rate of 3,4 GHz per physical core. The third system has an Intel Core i7-5820K hexa-core processor and 16 GB memory. The processor has a 15 MB L3-Cache and also supports hyper-threading. The basic clock rate is 3,3 GHz. The first system will be further referred to as 8-Core, the second one as 16-Core and the third one as 6-Core.

4.4 Performance

Each test was performed with the default source (=goSeed(1)) of the *math/rand* package, a fixed learning rate of 0.25 and 15 epochs. The networks were trained on 50000 training set examples. As described earlier, *implementation 1* synchronizes after every mini-batch, *implementation 2* after every epoch and *implementation 3* at the end of the training (depending on the configured number of epochs).

Accuracy. Implementation 2 & 3 can not keep up with the first version's accuracy when multithreaded because the parameters get synchronized too infrequently. This happens regardless of the network size. Shuffling the data and averaging the weights and biases after every epoch improves the overall accuracy of implementation 2 a bit, but does not affect the accuracy loss curve when

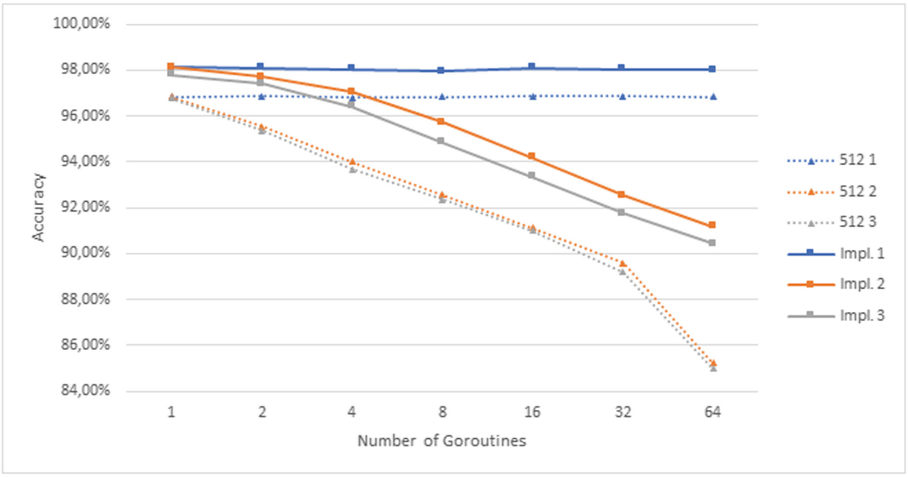

Fig. 2. 784-800-10 network accuracy on the test data set with a mini-batch size of 128 (solid line) and a mini-batch size of 512 (dotted line)

compared to implementation 3. Decreasing the number of neurons from 800 to 450 and 100 increases the error rate in general, with the smallest network performing worst. As seen in Fig. 2, all three implementations profit from a smaller mini-batch size and drop in accuracy with a larger size, although networks with more than one hidden layer generalize better with slightly larger mini-batch sizes. Splitting 800 neurons into three hidden layers (440-240-120) produces a bit more irregular results but generally improves the accuracy. Splitting 450 as well as 100 hidden neurons (250-135-65/55-30-15) affects the curves in a similar way, although in those cases implementation 1 does not profit from the deeper networks in each instance because of the irregularities, which get stronger the smaller the deep network is. Figure 3 shows a comparison between a network with 450 hidden neurons in one hidden layer and a network with 450 hidden neurons split in three layers. Especially implementation 2 produces noticeably better results with the deeper network. The accuracy results do not differ between the systems, since all three computers performed the tests with the same seed.

Execution Time. Shuffling and synchronizing after every epoch in implementation 2 produces marginal additional execution time overhead compared to implementation 3. While the difference can be seen in networks with 100 neurons in the hidden layers, where training time is short in general, it vanishes in larger ones, where the training takes considerably more time. Both versions outperform implementation 1 in every test case, although the results converge with a greater mini-batch size as seen in Fig. 4. Besides impacting the error rate, splitting neurons into three hidden layers also reduces the training time significantly in comparison to networks with the same number of hidden neurons in one hidden layer. Figure 5 shows the execution times of the two 450 hidden

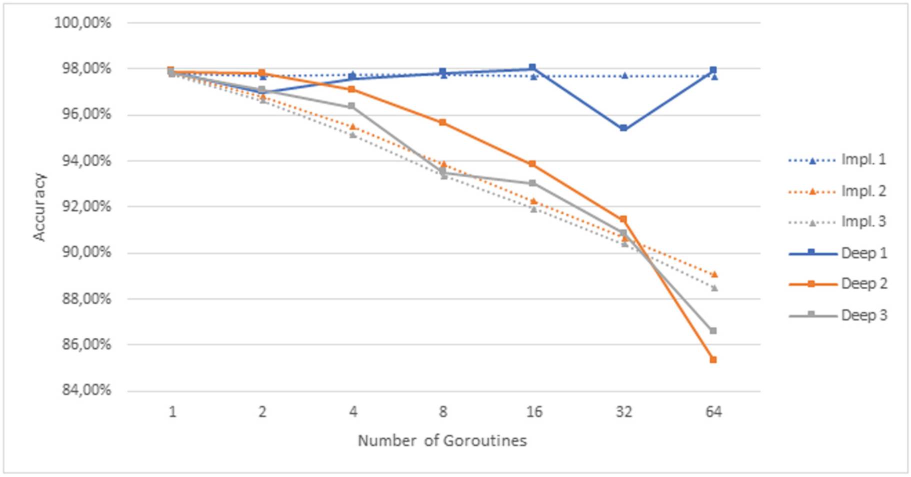

Fig. 3. 784-250-135-65-10 network accuracy (solid line) on the test data compared to a 784-450-10 network (dotted line) with a mini-batch size of 256

neuron networks compared in Fig. 3. Interestingly, the 16-core machine and the 8-core machine generally both perform best when run with eight goroutines. The 6-Core system runs fastest with four goroutines, although it should be stated, that the tests were only carried out with powers of two. Most of the time, the 16-Core and the 6-Core system already outperformed the 8-Core systems peak performance with only two goroutines, showing that the performance per core had a greater impact on the training time than the number of cores.

Speedup. The three test systems achieve different speedup results. While the 16-Core machine performs best in networks with 100 or 450 hidden neurons, the 8-Core machine shows a greater speedup than the others in networks with 800 hidden neurons.

All three systems show better speedup values when training smaller networks, although the 8-Core machine scales equally and sometimes even better in networks with 800 hidden neurons than in networks with 450 hidden neurons. This can be seen in Fig. 6 and compared with the results of the 6-Core and the 16-Core system visualized in Figs. 7 and 8. Figure 6 also shows the exceptional results of the 8-Core system in networks with 800 hidden neurons, where the speedup of implementation 1 and 2 increases even with 64 goroutines. This happens regardless of the mini-batch size. Although having lower peak values, the 8-Core machine also scales better than the other systems in networks with 450 hidden neurons. While the speedup from implementation 1 with up to eight goroutines and a mini-batch size of 512 is comparable with the speedup from implementation 2 and 3, it decreases with a lower mini-batch size as already mentioned earlier and shown in Fig. 4. Networks with three hidden layers achieve a better speedup than networks with the same

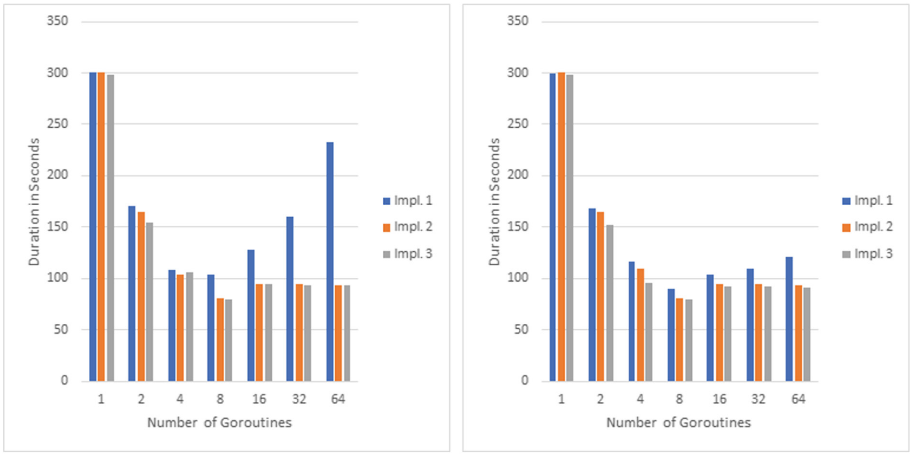

Fig. 4. 8-Core 784-100-10 network execution time with a mini-batch size of 128 on the left and a mini-batch size 512 on the right

number of neurons in one hidden layer. Overall, the 8-Core system scales best but runs slowest with its two quad-core processors.

5 Lessons Learned and Future Work

5.1 Parallelization

Data parallel neural networks can achieve a great speedup compared to single threaded networks. Implementation 1, where the mini-batches get split into equal parts for each thread, probably has limited possible applications, despite the good accuracy rates. If large mini-batches are usable on the chosen data set and not too many threads are used, it can be a valuable choice. Otherwise, particularly with a higher number of threads, parameter averaging with more frequent synchronization than in implementation 2 and 3, or alternative strategies, for example asynchronous ones, should be chosen. In general, data parallelism is a simple and effective parallelization strategy, preferably for large data sets.

5.2 Deep Neural Networks

The used hyper-parameters, which were mostly chosen by trial and error, like the learning rate, are probably not the best ones in each instance. Still, the results were satisfying, especially for networks with three hidden layers. The initial purpose was to see if networks executed faster when the neurons from one hidden layer were split into more hidden layers. Surprisingly, the deep networks, although not being that deep in fact, proved to be not only noticeably faster but also more accurate. 800 hidden neurons were split to 440 neurons in the first

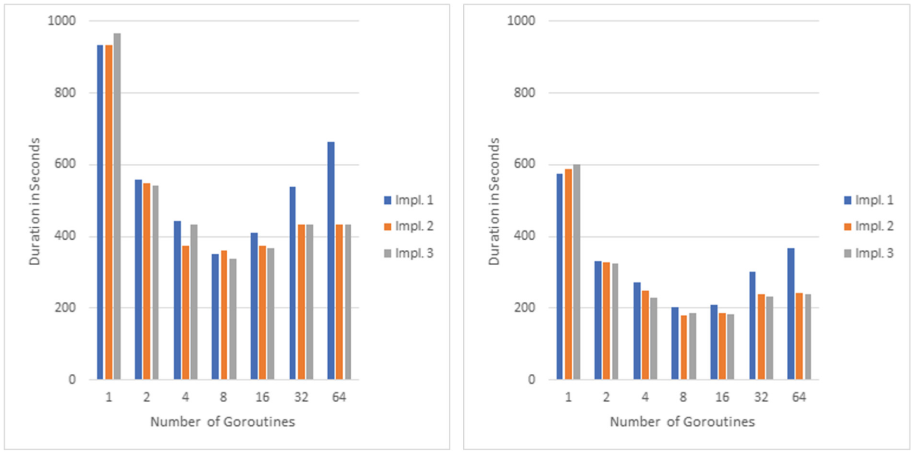

Fig. 5. 16-Core 784-450-10 network (left) execution time compared to a 784-250-135-65-10 network (right) with a mini-batch size of 256

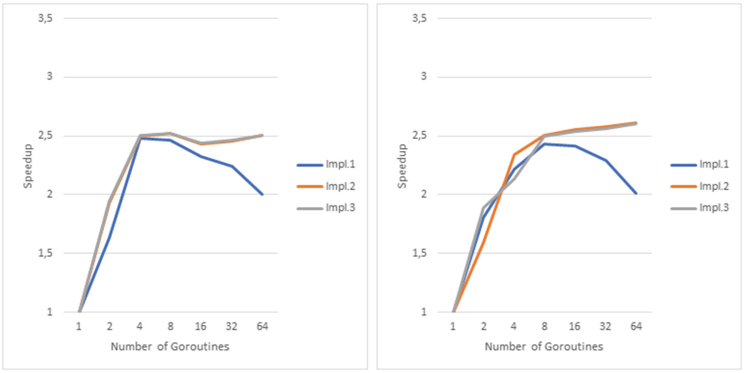

Fig. 6. 8-Core 784-450-10 network (left) speedup compared to a 784-800-10 network (right) with a mini-batch size of 512

hidden layer, 240 in the second and 120 in the third. 450 and 100 hidden neurons were split in a similar way (250-135-65/55-30-15). Distributing the neurons this way seems to be a good choice.

5.3 Matrix Multiplication

Unfortunately, the implementation presented in this paper does not exploit the full potential of mini-batches. The features of each training instance are stored in a vector and multiplied with the weight matrices. The same happens with the error in the backpropagation phase. Instead of doing these series of matrix-vector multiplications in a less efficient for-loop, it is highly recommendable to

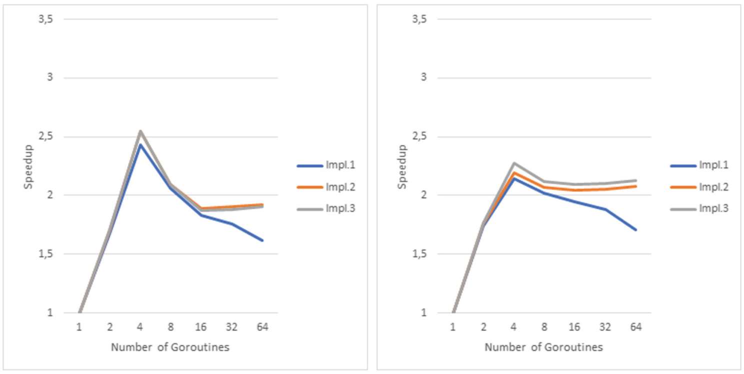

Fig. 7. 6-Core 784-450-10 network (left) speedup compared to a 784-800-10 network (right) with a mini-batch size of 512

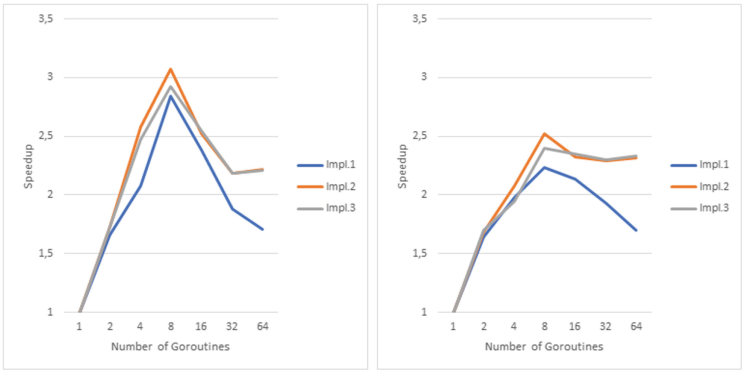

Fig. 8. 16-Core 784-450-10 network (left) speedup compared to a 784-800-10 network (right) with a mini-batch size of 512

save the features of all mini-batch examples in a weight matrix and to perform a more efficient matrix-matrix multiplication for each mini-batch. Particularly GPUs should profit greatly from this variation.

6 Conclusion

Data parallelism proved to be an efficient parallelization strategy. In combination with the programming language Go, a parallel neural network implementation is programmed 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. Further work will focus on comparing the parallel performance of neural networks written in Go with implementations written in other languages. Necessary to that end is the improvement of the Go implementation, starting by catching up on the missed opportunities mentioned above.

References

1. Google: The go programming language (2018). https://golang.org/doc/. Accessed 06 Jan 2018
2. Hoare, C.A.R.: Communicating sequential processes. In: Hansen, P.B. (ed.) The Origin of Concurrent Programming, pp. 413–443. Springer, New York (1978). https://doi.org/10.1007/978-1-4757-3472-0_16
3. Huqqani, A.A., Schikuta, E., Ye, S., Chen, P.: Multicore and GPU parallelization of neural networks for face recognition. Procedia Comput. Sci. **18**(Supplement C), 349–358 (2013). 2013 International Conference on Computational Science
4. LeCun, Y., Cortes, C., Burges, C.J.: The MNIST database of handwritten digits (1998). http://yann.lecun.com/exdb/mnist/. Accessed 06 Jan 2018
5. Liu, Y., Jing, W., Xu, L.: Parallelizing backpropagation neural network using mapreduce and cascading model. Comput. Intell. Neurosci. **2016**, 11 (2016)
6. McCulloch, W.S., Pitts, W.: A logical calculus of the ideas immanent in nervous activity. Bull. Math. Biophys. **5**(4), 115–133 (1943)
7. Meyerson, J.: The go programming language. IEEE Softw. **31**(5), 104 (2014)
8. Pethick, M., Liddle, M., Werstein, P., Huang, Z.: Parallelization of a backpropagation neural network on a cluster computer. In: International Conference on Parallel and Distributed Computing and Systems (PDCS 2003) (2013)
9. Prasad, N., Singh, R., Lal, S.P.: Comparison of back propagation and resilient propagation algorithm for spam classification. In: Fifth International Conference on Computational Intelligence, Modelling and Simulation, pp. 29–34, September 2013
10. Riedmiller, M., Braun, H.: A direct adaptive method for faster backpropagation learning: the RPROP algorithm. In: IEEE International Conference on Neural Networks, pp. 586–591. IEEE (1993)
11. Sato, I., Fujisaki, R., Oyama, Y., Nomura, A., Matsuoka, S.: Asynchronous, data-parallel deep convolutional neural network training with linear prediction model for parameter transition. In: Liu, D., Xie, S., Li, Y., Zhao, D., El-Alfy, E.S.M. (eds.) ICONIP 2017. LNCS, vol. 10635, pp. 305–314. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-70096-0_32
12. Zhang, W., Gupta, S., Lian, X., Liu, J.: Staleness-aware async-SGD for distributed deep learning. In: Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence, pp. 2350–2356. AAAI Press (2016)