Go Concurrent Speedup
Artificial Neural Network for
MNIST Digit Recognition

Karv
(`karvprime.github.io`, Twitter @KarvPrime)

October 31, 2018

Contents

Abstract

The main question was how much speedup could be generated when using concurrent goroutines. A simple concurrent feedforward network for MNIST [1] digit recognition with the programming language Go [2–5] was created to find the answer. The first findings when using a Lenovo Yoga 2 showed a speedup of 252% when utilizing 4 goroutines. When testing with a Banana Pi M3 there were more convincing results: 320% with 4 goroutines, and 432% with 8 goroutines. Another observation when using more goroutines than available cores only creates an almost negligible slowdown with the Lenovo. The Banana Pi had a total speedup of 498% when utilizing 16 goroutines, even though there were only 8 threads available, due to pipelining.

Keywords

Go, Feedforward, Backpropagation, Artificial Neural Network, MNIST, Character Recognition, Parallel Computing, Goroutines, Speedup

1 Introduction

Neural networks and artificial intelligence are becoming more and more important not only in research, but also in daily used technology. Due to higher amounts of data these artificial agents have to analyze there is the need for a larger throughput and highly efficient neural networks. The programming language Go looks promising for developing such a highly efficient agent, as the language itself has been made not only for highly efficient parallelization but also with fast development in mind. The main question is if Go is suitable for a highly efficient parallelization of neural networks. The main objective is the creation of an efficient parallelized neural network. There is a possibility that Go could lead to a higher parallelization efficiency/speedup than other programming languages. As Go is a young programming language[I], the literature about this specific topic is very sparse to almost nonexistent. There are, in example, tertiary sources like websites comparing the general throughput of Go in comparison to web languages like NodeJS, PHP and Java[II]. Other literature is related to parallelization speedup. There are also some neural networks realized in Go. No sources for a better comparison of parallelization techniques has been found. The scope of this work is to find out the speedup when using multiple goroutines with a neural network while maintaining a high and sustainable classification accuracy. A working MNIST digit recognition system has been created for testing the speedup with up to sixteen cores. The network and parameters have been optimized, but due to only negligible improvements with more than 100 hidden layer nodes this amount has not been exceeded. The execution time for one epoch has been sped up from 856.57-1271.73 (median 1005.80) seconds with 1 goroutine to only 171.50-221.38 (median 201.82) seconds with 16 goroutines with a Banana Pi M3. The Lenovo Yoga 2 showed a less significant speedup with 137.29-146.01 (median 142.33) for 1 goroutine to 55.10-64.62 (median 56.49) with 4 goroutines. Additional goroutines exceeding the maximum thread limit brought further speedup due to pipelining with the Banana Pi, but a negligible speed loss for the Lenovo Yoga.

[I]First stable version of Go: 28.03.2012 [6]

[II]https://www.toptal.com/back-end/server-side-io-performance-node-php-java-go

2 Data and Methods

The following data and methods have been used to gain insight on the speedup possibilities.

2.1 Choosing the data and parallelization method

Different approaches of which data could be used have been evaluated. Weather data, crime rates, etc. all seemed to be a good fit, but with the possibility of very inconclusive outputs. Finally the "Hello World!" of neural networks has been chosen: The MNIST dataset [1]. With this ready to use dataset the development process sped up as the convolutional part was already done.

Exemplar parallelism [7] has been chosen as parallelization technique. Within the workers the learning method is stochastic gradient descent [8], but due to combining the data in the main connectome, it behaves like a mini-batch update [9].

2.2 Basic structure

First a functional code for a basic neural network has been prepared. With this code it is also possible to define simple multi-layer feedforward networks. From that stable basis more functionality has been added (i.E. different activation functions) to ease up the future development. Then the parallelization of the neural network has been implemented. There were additional challenges with avoiding possible race conditions. Go was very helpful with its built in race detector which can be utilized with the "-race" flag. It was easy to spot any race conditions and therefore the development sped up in the area deemed to take the most time. Afterwards the possibility to input and compute large datasets has been implemented. A batch file functionality for ease of testing as well as data output functionality have been added too. Afterwards the code and neural network have been optimized for a balance of speed, memory usage and training quality. Shuffling of the training data has been implemented to prevent any unwanted behavior that comes from repeated data. The elastic net regularization [10] has been chosen to get better results and more stability for the neural network.

2.3 Implementation of the basic program structure

The program is structured into different parts which will be explained in this section. The root folder contains the folders "Batchfile", "Data", and "Core". The code will be walked through in more detail in the following subsections.

2.3.1 Batchfile

This folder contains the Batchfiles, which will be processed by the program.

The batchfile is used in the following way:
Each line in the batch file represents one action which the software will execute.
ResultFile:<Filepath> $\rightarrow$ Path to the result output
NetworkFile:<Filepath> $\rightarrow$ Path to the network creation file
PersistenceFile:<Filepath> $\rightarrow$ Path to the persistence
TrainFile:<Filepath> $\rightarrow$ Path to the training data
TestFile:<Filepath> $\rightarrow$ Path to the test data
PreProcessing:<PreProcessing> $\rightarrow$ [None, MeanSubstraction, Proportional]
Parallel:<Integer> $\rightarrow$ Number of goroutines
WorkerBatch:<Integer> $\rightarrow$ Batch size for each worker before result merge
LearningRate:<Float> $\rightarrow$ Rate of learning
Lambda:<Float> $\rightarrow$ Lambda value for elastic net regularization
MinWeight:<Float> $\rightarrow$ Minimum weight for connectome initialization
MaxWeight:<Float> $\rightarrow$ Maximum weight for connectome initialization
TargetColumnsStart:<Integer> $\rightarrow$ First field that is a target value
TargetColumnsEnd:<Integer> $\rightarrow$ First field that isn't a target value
Train:<Integer> $\rightarrow$ Train the network x times
Test:<Integer> $\rightarrow$ Run a test x times1

2.3.2 Core

Herein lies the main program to start the Brain.

Brain provides the main neural network functionality and contains the following subfolders:

- Connection: Mapping connections between the layers.
- Layer: Implementation of the singular layers.
    - Activation: Activation functions for the neuron layers.
        * Interface: Interface for activation functions.
    - Layout: Translation of the layout files for the single layers.
- PreProcessing: Pre processing functions for the neural network.
    - Interface: Interface for pre processing functions.

FileHandler is responsible for handling different files, as well as reading and writing contents from and to files and contains the following subfolder:

- ReadWriter: The implementation for operating system level reading, writing, and counting lines.

Logging creates Logfiles to tackle eventual errors and/or possible inconsistencies. Currently there is no setting for error severity, so every console log output will be written to the current logfile.

Persistence is responsible for reading and writing the weight map of the neural networks main connectome.

WorkerPool has a name which is a little bit misleading. Here the workers are created, but also their networks, which will get assigned to them. It also provides the main connectome to the core program. Due to the lack of a better name and for creating the network "pool", this naming stuck for now.

Contains the following subfolder:

- Worker: The working class for seizing the means of computation.

2.3.3 Data

All standard data input and output files are stored here.

Contains the following subfolders:

- Log: Logging data

- Networks: Networks which can be created

- Persistence: Persistent memory

- Result: Results from learning and testing

- Test: Test files

- Train: Training files

Logs are all log outputs written by the program. If any bugs or inconsistencies occur the error will be seen in the console output as well as in the log. Results are the learning durations as well as the correct and incorrect guesses, and the error rate, of the finished task in CSV format. Test contains all the test files. Training contains all training files. Network contains different network layouts.

The network batch file is used in the following way:
Each line represents one layer in the network.
<Activation>,<Neurons>

Current activation functions:

- Identity

- Logistic

- TanH

- ReLU [11]

- LeakyReLU [12]

- ELU [13]

- SoftMax

In example "SoftMax, 10" (without quotation marks) creates one SoftMax layer with 10 neurons. Currently a bias neuron will be added to every but the last layer.

2.3.4 Program sequence

The inner workings of the implementation are held very simple and can be divided into the sections "Startup", "Initialization", "Preparation", and "Training / Testing".

Startup
First NeuroNet.go creates a new Core instance and starts it with providing the path to the batchfile. Currently this is hardcoded[III].

Initialization
Then Core.go initializes the main program functionality. It finds the root path and initializes the logging functionality. The software then loads and reads the batch file. The file handlers for the other files (data, result, persistence, and network) as well as the worker pool get initialized and fills the needed attributes (weight range, etc.) with standard values to prevent the software interrupting if the user doesn't provide these. The core then runs over the provided batch lines, where it sets attributes or starts training or testing according to the tasks provided.

Preparing Training and Testing - core.runMode()
First the basic values for result file are provided. It gets filled with additional values on the way (i.E. test results or time needed for training). The data and output channels for the workers are created and the worker pool gets initialized. First the worker pool creates the main connectome[IV]. The worker pools main responsibility is to create the networks with their layers and the workers while providing each worker its own network. It returns the networks, the connectome, setup time and any possible error back to the core function and get assigned to the core struct. After that process a state connectome gets created to later remember the state for calculating the differences after each worker training cycle. The data reader gets initialized,

---

[III]There are multiple directions to go from here. Either the software waits for a user input, or it gets the path from a settings file, or only a single Batchfile on a certain path is permitted. The best option would be to provide a user interface where it's possible to change the batch path and save it in a settings file to prevent unnecessary retyping, but to also give the user the ability to provide multiple test files. This falls under the section "future work".

[IV]In theory the worker pool isn't responsible for creating the connectome. But as it creates the networks for the workers the decision has been made that it can also provide the connectome, if it already has all the needed information. The naming should be changed as it is ambiguous - it isn't a worker pool, but it creates the workers, the networks with layers, and the main connectome.

reads the data, puts it into an [][]float64 array[V] and gets the length to provide the user with that information when training or testing starts. When training the data also gets shuffled to prevent any problems which could occur to training with repeated patterns. If there is already a persistence file, the data gets loaded. If there isn't one, a new connectome with deterministic random weights will be created. Now the real training or testing commences (more details about these processes will follow in the next part "Training and Testing"). Afterwards the user gets provided with the final results: How much time did the software need for that epoch and in total, as well as the correct and incorrect recognitions when testing. When a training has been concluded, the provided persistence will also save the connectomes current training status.

Training and Testing - core.run()
First the user gets informed the the mode (training or testing) has been started. A flag is set to true for checking if the main loop is still running. The currentLine, activeWorkers, correct and incorrect variables as well as the data slice length get all set to 0, while the timeElapsed gets set to a 0.0 double type. A state will be created to save the current state of the connectome to calculate the weight differences when training. Then the main loop will start.

- First the starting time will be saved as it is needed for the speedup calculations. The networks for the workers will be initialized with the connectomes current status.

- Then a loop assigns the worker batches to the available workers and increments the activeWorkers when the loop gets finished.

  - If there are still lines, get the length of the slice, which is either the user set worker batch size, or the remaining lines (when exceeding the total lines). Then send the slice to the data channel[VI].

  - Otherwise set running (for the main loop) to false to stop execution when finished and break out of the current loop.

- If there are activeWorkers, the next loop will run as long as there are any activeWorkers left. But first, if in training mode, save the weights

---

[V]First the file is read line by line; but for shuffling the whole dataset the whole data is needed at the beginning which could eat away memory when used for larger inputs.

[VI]It's faster to assign batches of data to the singular workers instead of only single lines. As there are no changes occuring within the original data array, slices have been used. These are pointers to the original array fields, and speeds up the process even further.

of the main connectome to a new state to use them as comparison for the next loop.

  – Get the id of the next worker that finished its work from the output channel. So the software will wait until all workers are finished. If the mode is training, add the weighed difference of the old connectome state and the new worker connectome state to the main connectome state.

- Add the time since start to the timeElapsed and show the user this information in the terminal, along with the finished and remaining lines.

- When testing also get the correct and incorrect classifications and inform the user about the current state.

After the main loop the user gets informed about the conclusion of the current training or testing mode. The function returns currentLine, timeElapsed, correct, incorrect to save that data in the persistence and result files.

2.4 The math

Different activation functions have been tested to get a high accuracy, although this is not the purpose of this work. At the end it has been concluded that the best way is to start without any data normalization for the data to be put into the input layer. But before any activation function runs over the layer, the data is normalized by dividing each value by the size of the layer (including the bias) to minimize the risk of exploding gradients [14].

The following activation functions have been used:

- Input layer: Identity
- Hidden layer: ELU [13]
- Output layer: SoftMax

Variables are aas followed:

- η = learning rate
- t = target
- x = neuron value before activation
- φ = activation function
- δ = error
- w = weight

2.4.1 Activation functions

Identity
The simplest activation function is the identity function. It simply states that the value stays the same, as in equation (1).

$$\varphi(x) = x \tag{1}$$

So the derivation simply is 1 as shown in equation (2).

$$\varphi'(x) = 1 \tag{2}$$

Exponential Linear Unit

In comparison to ReLU and leaky ReLU, ELU "[...] speeds up learning in deep neural networks and leads to higher classification accuracies." [13] and therefore has been chosen over the other options. Equation (3) shows the math, where alpha is a positive value that can be freely chosen. Caution: The math has been updated since version 3 of their paper[VII]. The code has already be changed accordingly, but the test results stem from the following equations.

$$\varphi(x) = \begin{cases} x & \text{if x} \geq 0 \\ \alpha * (e^{x} - 1) & \text{if x} < 0 \end{cases} \tag{3}$$

The derivation for training is shown in equation (4).

$$\varphi'(x) = \begin{cases} 1 & \text{if x} \geq 0 \\ \varphi(x) + \alpha & \text{if x} < 0 \end{cases} \tag{4}$$

SoftMax

The SoftMax function gives us a classification of the likelihood that the current input represents a certain number. The math is straightforward and shown in equation (5).

$$\varphi_i(\overrightarrow{x}) = \frac{e^{x_i}}{\sum\limits_{j=1}^{J} e^{x_j}} \tag{5}$$

But with this equation, exploding or vanishing gradients [15] can become a problem due to the high likelihood of numbers exceeding[VIII] the maximum value of float64[IX]. For the SoftMax activation there is a little "trick". It is

---

[VII]The equality sign has been swapped. This leads to a change in the derivation: If x = 0, then it becomes α instead of 1. No change happens if $\alpha = 1$. As $\alpha = 0$ The mistake of using the wrong equation happened in many places.

[VIII]When calculating the power of e by each number and sum them up ($\sum\limits_{j=1}^{J} e^{x_j}$) it tends to exceed...

[IX]... 1.7976931348623157e+308 (which will be shown with math.MaxFloat64), so Go sets this "immeasurable" number to +Inf which will become NaN if this value gets used for any calculation.

possible to add a scalar, as shown in (6), without changing the value of the softmax function [16].

$$\varphi_i(\overrightarrow{x}) = \frac{e^{x_i + S}}{\sum\limits_{j=1}^{J} e^{x_j + S}} \tag{6}$$

So, instead of using softmax(x), softmax(z) - with a scalar value of the negative x maximum - has been used, as in equation (7).

$$z_i = (x_i - max_i(x_i)) \tag{7}$$

If we use the maximum, we push the calculation into the negative number spectrum. So, instead of having values ranging over]-∞ , ∞[, they've been shifted to]0, 1][X], as in (8).

$$\varphi_i(\overrightarrow{x}) = \frac{e^{z_i}}{\sum\limits_{j=1}^{J} e^{z_i}} \tag{8}$$

Equation (9) derivation for training is a little bit more complicated.

$$\varphi'_i(\overrightarrow{x}) = \frac{\partial \varphi_i(\overrightarrow{x})}{\partial x_j} = \begin{cases} \varphi_i(\overrightarrow{x}) * (1 - \varphi_j(\overrightarrow{x})) & i = j \\ \varphi_i(\overrightarrow{x}) * (0 - \varphi_j(\overrightarrow{x})) & i \neq j \end{cases} \tag{9}$$

Mathematicans use (10) to shorten the equation to (11).

$$\delta_{ij} = \begin{cases} 1 & i = j \\ 0 & i \neq j \end{cases} \tag{10}$$

$$\varphi'_i(\overrightarrow{x}) = \frac{\partial \varphi_i(\overrightarrow{x})}{\partial x_j} = \varphi_i(\overrightarrow{x}) * (\delta_{ij} - \varphi_j(\overrightarrow{x})) \tag{11}$$

[X]As computers work with floating point arithmetic the value could evaluate to zero though when getting smaller than machine precision allows.

But in the end it comes down to the same function as the logistic derivation, shown in equation (12). As the result of the derivation is a diagonal matrix [17] there is no need to calculate the whole matrix.

$$\varphi'_i(\overrightarrow{x}) = (1 - \varphi_i(\overrightarrow{x})) * \varphi_i(\overrightarrow{x}) = (1 - x) * x \tag{12}$$

2.4.2 Elastic Net Regularization

The elastic net regularization [10] has been used for weight updates.

It's a combination of the lasso regression (13) and the ridge regression (14).

$$L^1 = \lambda |w| \tag{13}$$

$$L^2 = \lambda w^2 \tag{14}$$

The elastic net (15) is simple.

$$ElasticNet = \lambda |w| + \lambda w^2 \tag{15}$$

Computational optimization (16)

$$ElasticNet = \lambda(|w| + w^2) \tag{16}$$

For the derivation (17) the signum function (18) is needed.

$$|w| = w * sgn(w) \tag{17}$$

$$sgn(w) = \begin{cases} 1 & \text{w} > 0 \\ 0 & \text{w} = 0 \\ -1 & \text{w} < 0 \end{cases} \tag{18}$$

Which leads to (19).

$$ElasticNet' = \lambda(sgn(w) + 2w) \tag{19}$$

2.4.3 Loss function

Quadratic loss has been chosen as loss function. Although the classification of handwritten digits - as the name says - is a classification problem and therefore cross entropy loss should show better results. Different loss functions will be implemented in the future of this work.

$$L = -\frac{1}{2}\sum_{i}^{k}\left((t_i - \varphi(x_i))^2 + \lambda\left(|w_i| + w_i^2\right)\right) \tag{20}$$

The derivative is the logistic equation, therefore the loss function is equation (21).

$$L' = -\sum_{i}^{k}\left((t_i - \varphi(x_i)) + \lambda\left(\frac{1}{2}sgn(w_i) + w_i\right)\right) \tag{21}$$

2.4.4 Forward and backward propagation

All the previous information is needed to understand the forward and backward propagation methods.

Forward
After setting the inputs and targets, the first layer of the neural network gets activated. Then for each layer, the next neuron gets excited with the product of the activated value and the weight of the connection between the neurons (22).

$$x_j = \varphi(x_i) * w_{ij} \tag{22}$$

Backward
When learning, equation (23) is used to calculate the error.

$$\delta = t - \varphi(x) \tag{23}$$

The formula for the weight update, with learning rate and the regularization in (24).

$$\Delta w = \eta * (\delta * \varphi(x) + \lambda(sgn(w) + w)) \tag{24}$$

16

The final equation in (25).

$$w_{ij}{}^{+} = w_{ij} - \Delta w^{XI} \tag{25}$$

2.5 Choosing the parameters

ELU alpha = 0.5 (currently hardcoded)
The alpha value for ELU has been hardcoded as there was no incentive to do otherwise in the current software iteration.

workerBatch: 100
The worker batch has been chosen to merge the single neural networks as often as possible, but without losing too much performance due to context switching.

Minimum/Maximum weight (starting weights): [-0.1; 0.1]
As the weights are usually getting smaller, when learning occurs, the starting values have to be chosen to be 0.1 instead of 1. This led to the best outcome.

LearningRate: 0.8
The learning rate has been set to 0.8, as this led to the best outcome.

Lambda: 0.0000001
The multiplicator for the elastic net has been set to this value, as it provided the highest accuracy for the training and test set. As it is hard to tell if either L1 or L2 regularization is the best, there is only one lambda for setting both methods, to achieve a balance between the two methods.

3 Results/Evaluation

For testing the neural network, two available systems have been chosen: The Lenovo Yoga 2 laptop, as it is a dual core consumer product which utilizes threading and a turbo mode for higher workloads, with 64 Bit Linux. The Banana Pi M3, as it is a well known octa core home server, with no threading,

---

[XI]Some sources add the update to the old weight and subtract the elastic net - all calculations seem to be inverted. To prevent any confusion: This depends on how the error gets calculated. Here (output - target) has been used, while others may use (target - output).

and without data distortion due to turbo mode kicking in, and 32 Bit Linux. Both systems have a standard CPU frequency of 1.80 GHz, although the minimum and maximum values differ.

There are stark differences in computation speed as well as speedup between the Intel and the ARM architecture. As RISC and CISC lost their meaning to describe newer architectures, it is not possible to draw the conclusion here[XII], although the main effect could come from the smaller - and therefore faster access rates - of the Intel L1 and L2 caches, or the lack of an L3 cache in the ARM architecture. Further research would be needed.

[XII]Some of the differences could also be caused by higher bit-level parallelism in the 64 bit system, different general purpose register sizes, a higher expense of doing 64 bit calculations on a 32 bit system, different out of order execution approaches, other effects, or a combination of all of these.

3.1 Benchmark

When using pprof for checking the total cpu usage of the code parts with BenchBatch[XIII], it can be seen in Figure 1 that thinking and training takes up about 96% of the total time. Thinking takes about 40% of the time, training takes about 56%. Thinking is the forward propagation, training is the backward propagation. Due to that high amount of cpu usage heavy optimizations were made in these code parts, as these had the greatest effects.
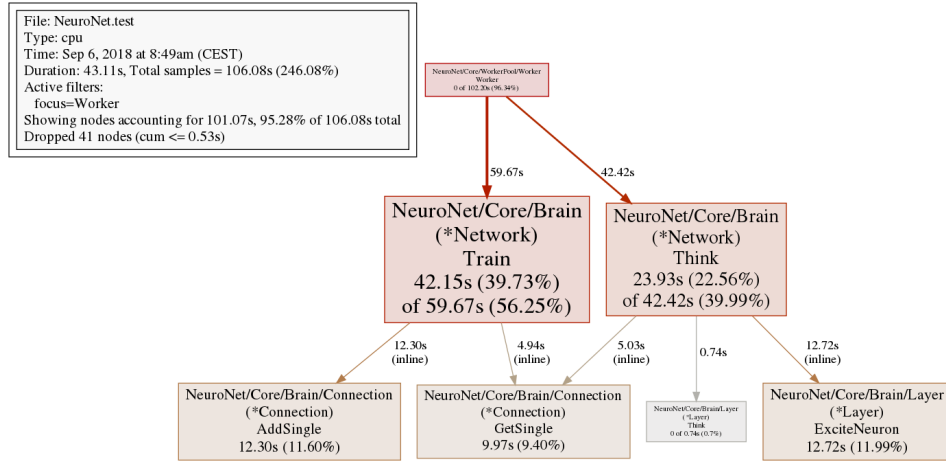

Figure 1: pprof worker CPU profile

The utility functions only play a marginal role. Even though they wouldn't need any optimization, they've been optimized for general code quality reasons. In example the garbage collector (mallocgc) is hardly used in the utility functions, and almost never in the main code part. As strings are only converted when needed, these parts of the code - even though they're not impacting the measurements - have been highly optimized. Maybe there's still room for further optimization, but for the general purpose this goal has been exceeded.

[XIII]BenchBatch utilizes 4 cores, uses a worker batch of 100 lines, and processes 1 training with 60.000 MNIST lines as well as 1 test with 10.000 MNIST lines

3.2 Test systems

The final tests were made with a "Lenovo Yoga 2 Pro Multimode Ultrabook" as well as a "Banana Pi M3". The system specifications[XIV]:

Specifications of the Lenovo: Intel(R) Core(TM) i7-4500U CPU @ 1.80GHz, Dual Core (4 threads)
Min CPU: 800 MHz, Max CPU: 3.0 GHz
32 KiB L1 cache, 256 KiB L2 cache, 4 MiB L3 cache
2x4096 DIMM @ Clockspeed 1600 MHz
64 Bit Linux Ubuntu 18.04.1 LTS

Specifications of the Banana Pi M3:
A83T ARM Cortex-A7 octa-core CPU @ 1.80 GHz, Octa Core (8 threads), 4800 BogoMIPS
ARMv7 Processor rev 5 (v7l)
Min CPU: 480 MHz, Max CPU: 1.8 GHz
512 KiB L1 cache, 1 MiB L2 cache
2GB LPDDR3
32 bit (armv7l) Linux Ubuntu 16.04.5 LTS, MATE Desktop Environment 1.12.1

[XIV]Useful Linux terminal commands for getting information about a system:
Caches and RAM: lshw -C memory
RAM: dmidecode –type 17
Make and model: dmidecode | grep -A3 'ˆSystem Information'
CPU information: cat /proc/cpuinfo
Thread count: nproc
CPU usage (press 1 for seeing each CPU): top
Memory usage: vmstat -s
Short memory usage: free -m
CPU stats: lscpu | grep MHz
32/64 Bit: uname -m

3.2.1   Lenovo Yoga 2

The 252% speedup generated with 4 goroutines on the Lenovo Yoga 2 when
utilizing more than 1 processor is clearly visible in Figure 2. It is also visible
that using more goroutines than processors slows the execution time down
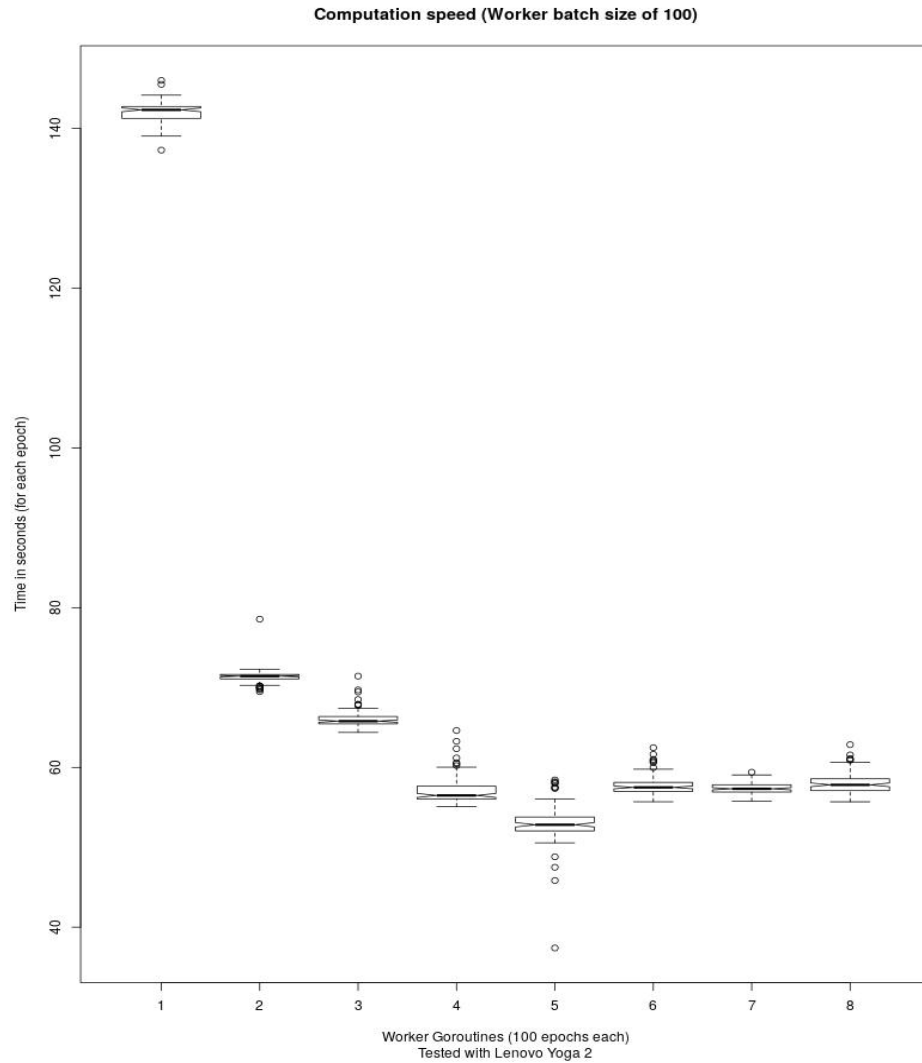only by an almost negligible amount.

Figure 2: Lenovo Yoga 2 speedup

Parallelization speedup comes at a price. Although very small, there is a slight decrease in recognition rates when utilizing more goroutines as shown in Figure 3.
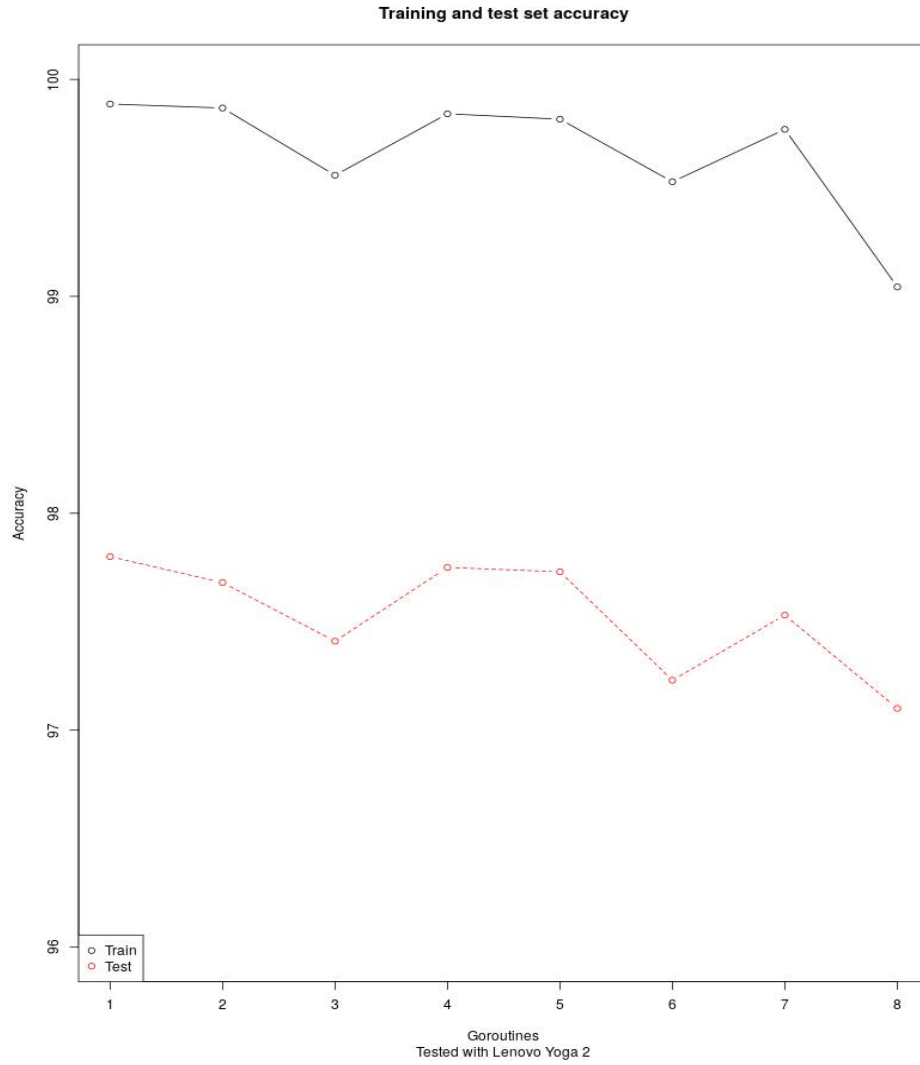

Figure 3: Lenovo Yoga 2 accuracy

3.2.2   Banana Pi M3

When looking at the results of the Banana Pi M3 in Figure 4, it is apparent that utilizing multiple cores leads to an even greater benefit than with the Lenovo. It was possible to generate a 320% speedup with 4 goroutines, and - due to pipelining - it was even possible to generate over 498% speedup when using more goroutines than there were threads available.
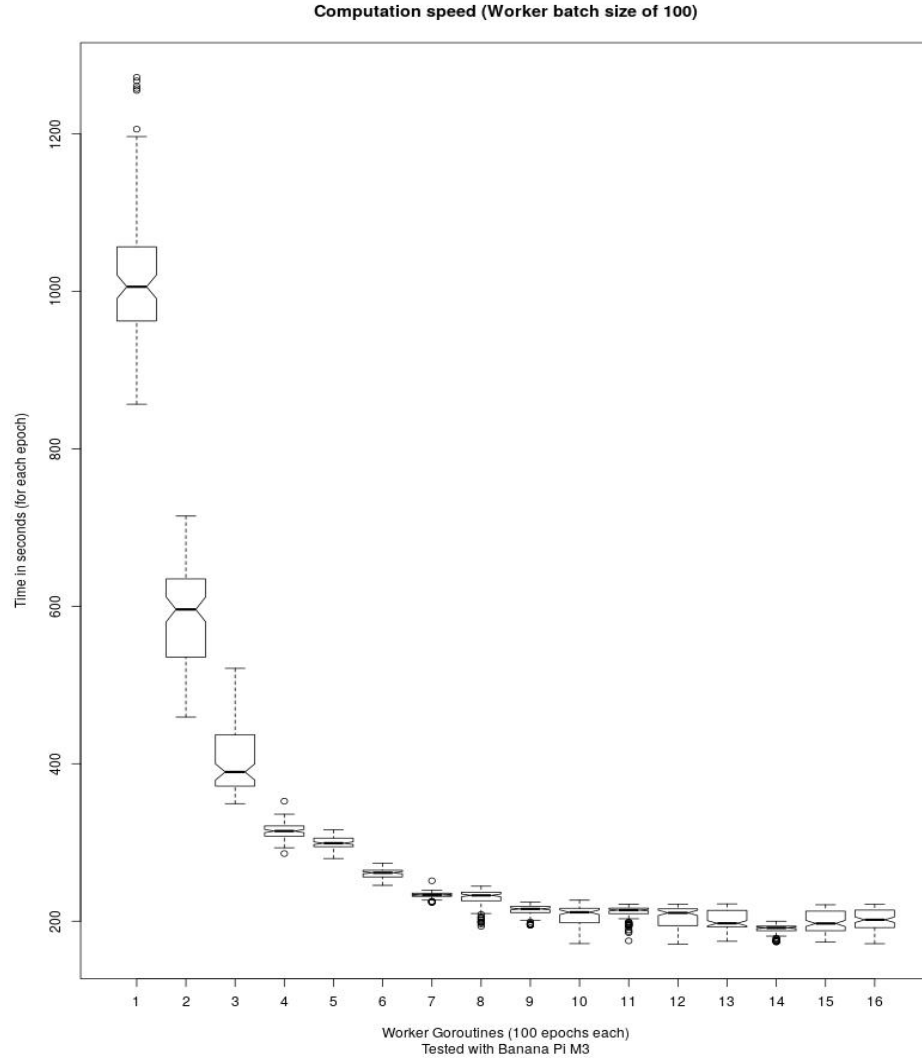
Figure 4: Banana Pi M3 speedup

The training and test set accuracies look promising too. A 99.26% training set accuracy and 97.14% test set accuracy with only one core has been accomplished. The accuracy does not get lower when utilizing more cores, even though quality differences in the recognition rate can occur. In Figure 5 it is clearly visible that recognition rate drops can occur at any time.
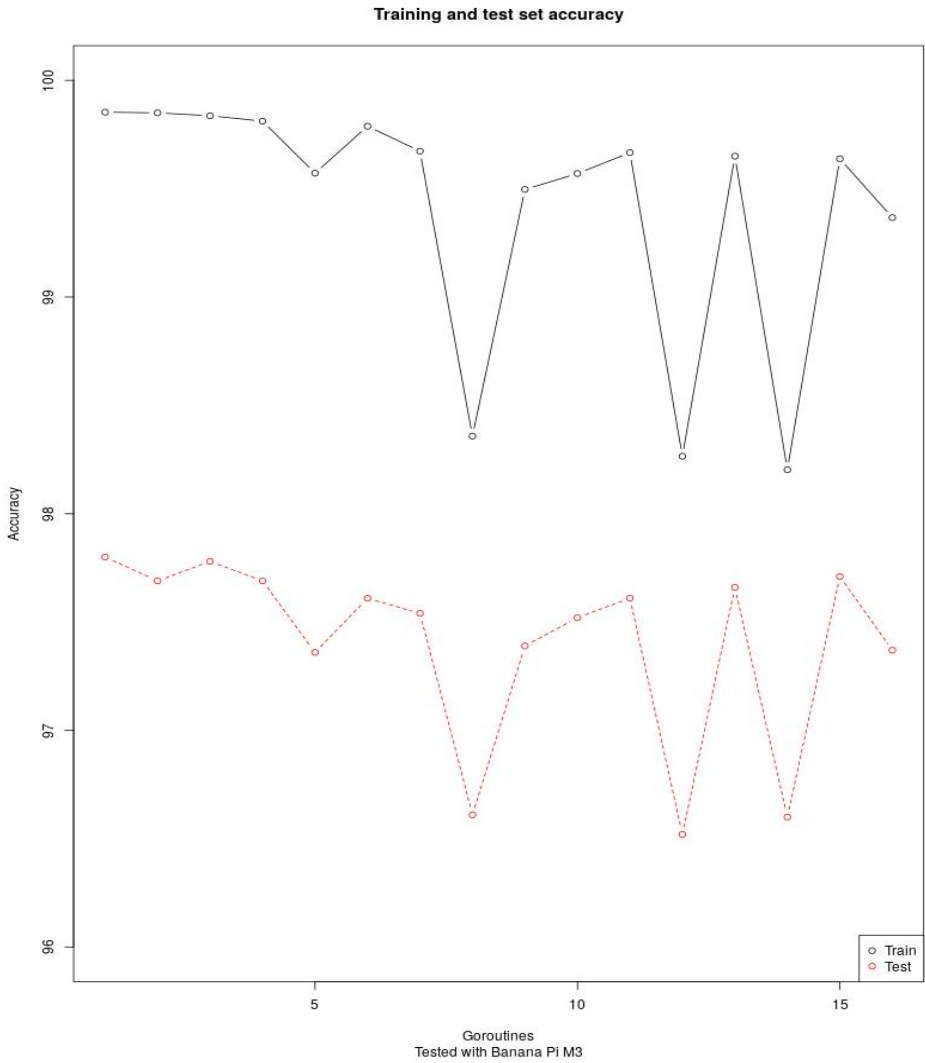

Figure 5: Banana Pi M3 accuracy

3.3 Accuracy growth depending on goroutines

When only one goroutine is usedFigure 6 with the Banana Pi, the neural network starts with a very high recognition accuracy after the first epoch and has a very good learning rate.
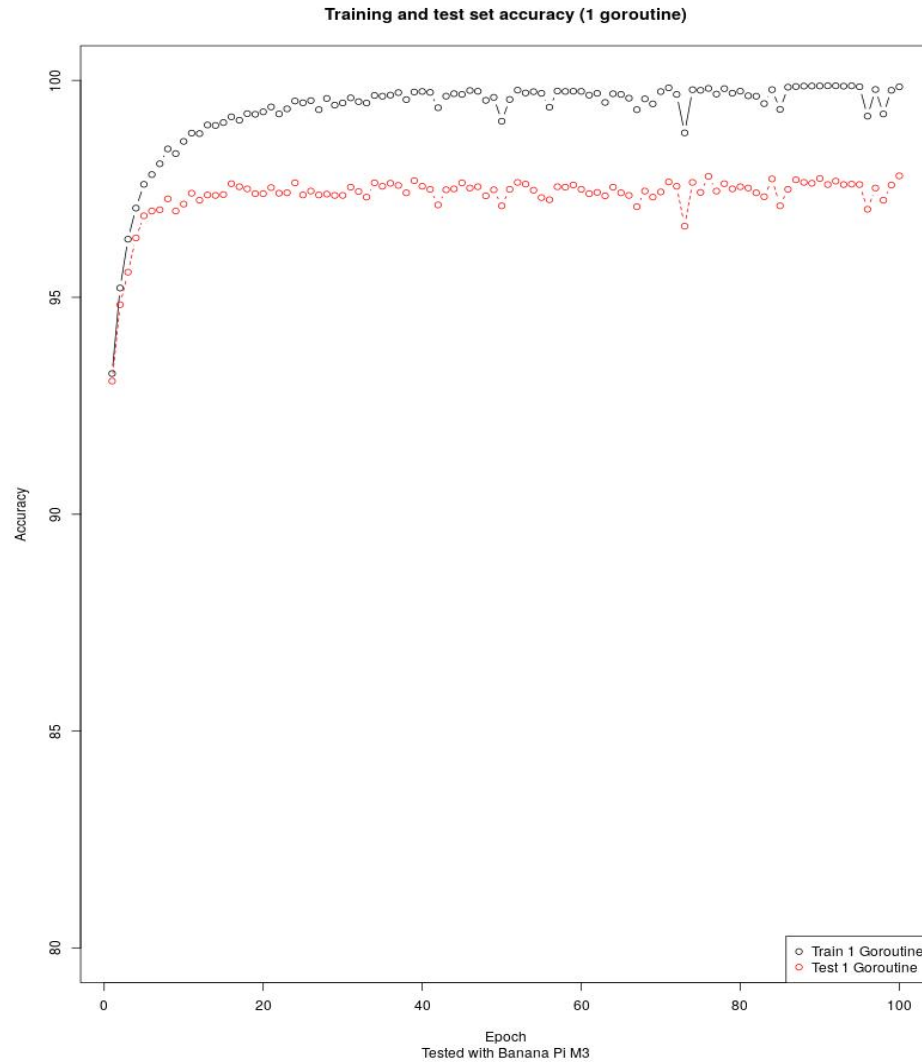

Figure 6: Accuracy growth with 1 Goroutine

25

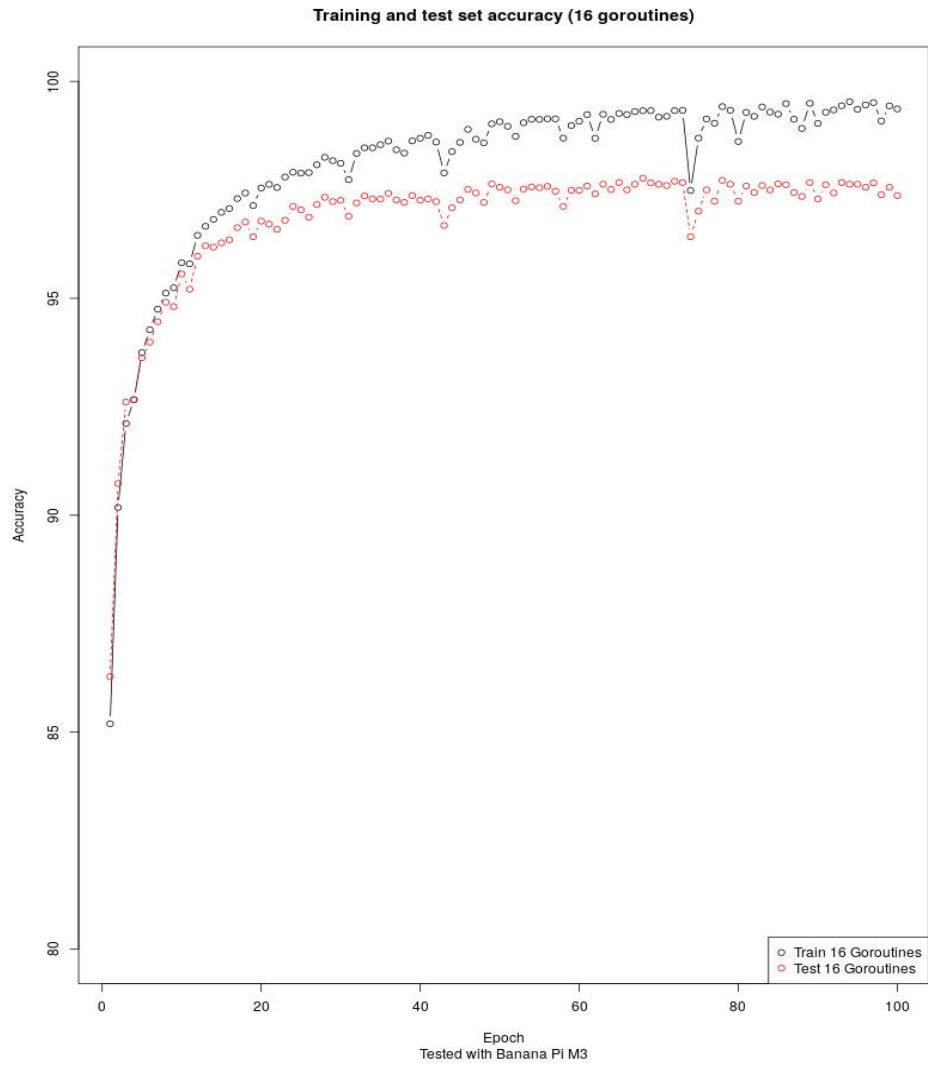With 16 goroutinesFigure 7 the recognition accuracy starts lower and the network takes longer to learn.

Figure 7: Accuracy growth with 16 goroutines

Recognition rates in a nutshell

1 Goroutine, accuracy > 90%/95%/99%:
93.24% accuracy after 1 epoch, 1040 seconds
95.22% accuracy after 2 epochs, 2006 seconds
99.03% accuracy after 15 epochs, 15608 seconds

16 goroutines, accuracy > 90%/95%/99%:
90.18% accuracy after 2 epochs, 392 seconds
95.12% accuracy after 8 epochs, 1583 seconds
99.02% accuracy after 49 epochs, 9628 seconds

To reach a higher accuracy with more goroutines more epochs and training samples are needed. But the speedup allows to train it in shorter time - or, to look at it from another perspective - to compute more inputs in a much shorter timespan.

4   Conclusion and future work

The final part of this work is to look at what has been learned about the "do's and don'ts of implementing neural networks", Go as a language, the drawn conclusion, and possible future work.

4.1   Lessons learned

There are certain roads to victory and many paths to development hell. The latter leads to a steeper learning curve and should therefore be preferred when trying to understand the implications of certain design decisions - but under normal circumstances the beaten path is the quicker route. These recommendations for other coders shall make implementing neural networks a little bit easier and shine a light on which thought processes are good and which are impractical to do.

4.1.1   Arrays instead of structs

Do not use one struct instance per neuron, connection, etc. as it has a large overhead. The compiler is able to optimize the usage of arrays. The first iteration of the neural network took hours for just one epoch on the Lenovo, while the array version takes less than a minute.

4.1.2 Only save when necessary

Only save and load data when needed. In the context of the neural network: Save either batchwise or after every epoch. Try to hold the data in the memory as long as possible[XV]. Memory is cheap, but not the execution time.

4.1.3 Machine readable is better than human readable

The conversion of data to XML, JSON, or any other human readable format takes a higher amount of computation time, memory, and disk space, than machine readable formats. If a human readable format is needed, it should only be created, if a human wants to read it and there is a need for them to do so. Sifting through millions of weights and updates is not something a human should do. But, depending on the use case, the human readable format can be created, when:
- The process is finished and the results shall be shown.
- An error occurs and the data is necessary to fix it.

If human entities want to access data while the process is running (in real time, or stepwise for debugging) there are different approaches:
- Create only one file every few epochs which can be accessed by multiple human entities. Do NOT create it for every entity that accesses the file.
- Duplicate the machine readable results and parse them on a different system. For snapshots a simple ID can be given to every file.

4.1.4 Parallelization and context switches

It takes time to store states of threads. Data has to be shoved around between CPU caches. If applicable give a worker as much data as possible, with one drawback in mind: More data merges mean higher fluctuations and slower computation - less data merges can lead to a more stable convergence and faster computation [18], as well as a higher level of generalization [14]. All while being able to perform online learning due to the singular workers performing stochastic gradient descent.

[XV]This is also true for other projects like websites or software. Every database/datafile access takes time. When going for a highly optimized software, only save changes, load data when changes in the database occur, and - if possible - only load it once at system startup or lazy load only when needed, depending on the use case.

4.1.5 Struct packing

In Go it is possible to pack structs. That means organizing the data types in a way so that they waste the least amount of memory. The principle for this work was "Memory is cheap, but even though students lack the money to buy some, there is no need to overdo it". But one late evening (at 7 o'clock in the morning) these principles had been thrown over board. So structs have been packed. If you're out of funds for additonal memory, there's a good explanatory article[XVI] for further reading about that topic.

4.1.6 Use slices instead of appending

Do not loop through lines of data to append it to a batch line by line. Use the slice functionality of Go - which passes them by reference - if applicable. The following data has been taken from the old model with using the updated weights for the error calculation Figure 8.


```
// Set new weights
neurons := network.layers[i].GetNeurons()
for x, output := range network.layers[i-1].GetNeurons() {
    for y := 0; y < network.layers[i].GetSize(); y++ {
        currentWeight := network.connections[i-1].GetSingle(x, y)

        // Get new delta
        errorValue[x] += neurons[y] * currentWeight

        // Signum
        if currentWeight > 0 {
            currentWeight += 1
            currentWeight += 0.5
        } else if currentWeight < 0 {
            currentWeight -= 1
            currentWeight -= 0.5
        }

        // Add weight with elastic net regularization
        network.connections[i-1].AddSingle(x, y, network.learningRate*neurons[y]*output-currentWeight*network.elasticNetMultiplicator)

        // Get new delta
        errorValue[x] += neurons[y] * network.connections[i-1].GetSingle(x, y)
        network.connections[i-1].AddSingle(x, y, network.learningRate*(neurons[y]*output-network.lambda*currentWeight))
    }
}
```

Figure 8: Code changes from wrong to correct code

In example the code in Figure 9 takes 129.68 seconds for 1 training and 1 testing with the MNIST dataset, 4 cores, and a worker batch setting of 100, as shown in Figure 10.

[XVI]https://medium.com/@felipedutratine/how-to-organize-the-go-struct-in-order-to-save-memory-c78afcf59

```
for ; activeWorkers < core.parallel; activeWorkers++ {
    var dataBatch [][]float64
    dataAvailable = false
    for i := 0; i < core.workerBatch; i++ {
        if currentLine < core.lines {
            // Append input to worker batch
            dataBatch = append(dataBatch, core.data[currentLine])
            dataAvailable = true
            currentLine++
        } else {
            running = false
            break
        }
    }

    // Sending input
    if dataAvailable {
        core.dataChannel <- dataBatch
    } else {
        break
    }
}
```

Figure 9: Old code, appending to a new slice
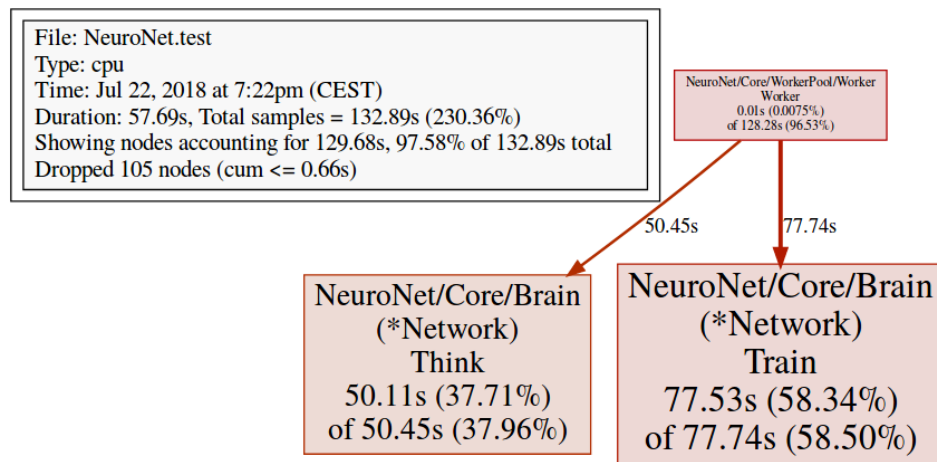
Figure 10: CPU profile before code changes

In comparison when utilizing slices instead of making a slice and appending the data lines within core.run() to send the worker batches to the workers

as shown in Figure 11 saved about 17 seconds on the Lenovo, as is visible in Figure 12.

```
for ; activeWorkers < core.parallel; activeWorkers++ {
    if currentLine < core.lines {
        if currentLine + core.workerBatch < core.lines {
            length = core.workerBatch
        } else {
            length = core.lines - currentLine
        }
        core.dataChannel <- core.data[currentLine:currentLine+length]
        currentLine += length
    } else {
        running = false
        break
    }
}
```

Figure 11: New code, using the slice functionality
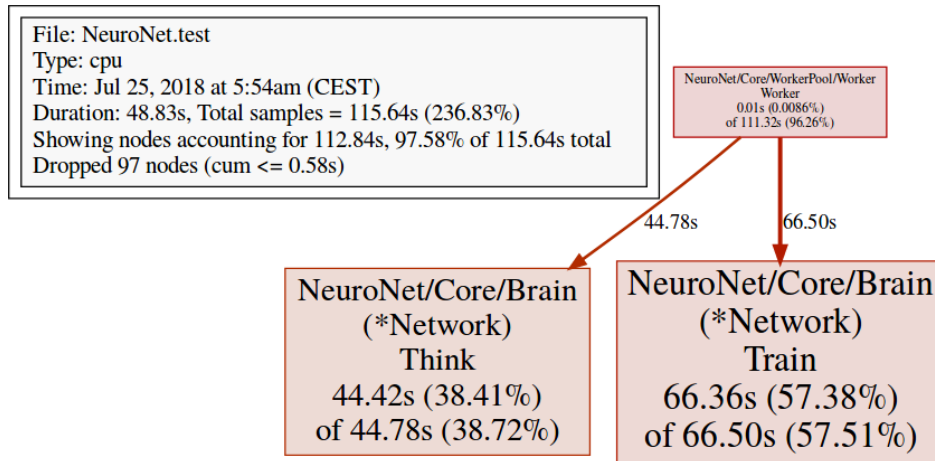
Figure 12: CPU profile after code changes

But when comparing the old speedup in Figure 13 with the new speedup in Figure 14, the computation speed is on average a little slower. The assumption is that - due to code optimizations and therefore less workload - the CPU chose to use a lower clock speed, so computation took longer. This assumption is untested, but context switching can be ruled out with a high certainty as the possible speedup of the new code has been tested with 4

cores: If context switching would have any negative impact, it should have shown in this test.
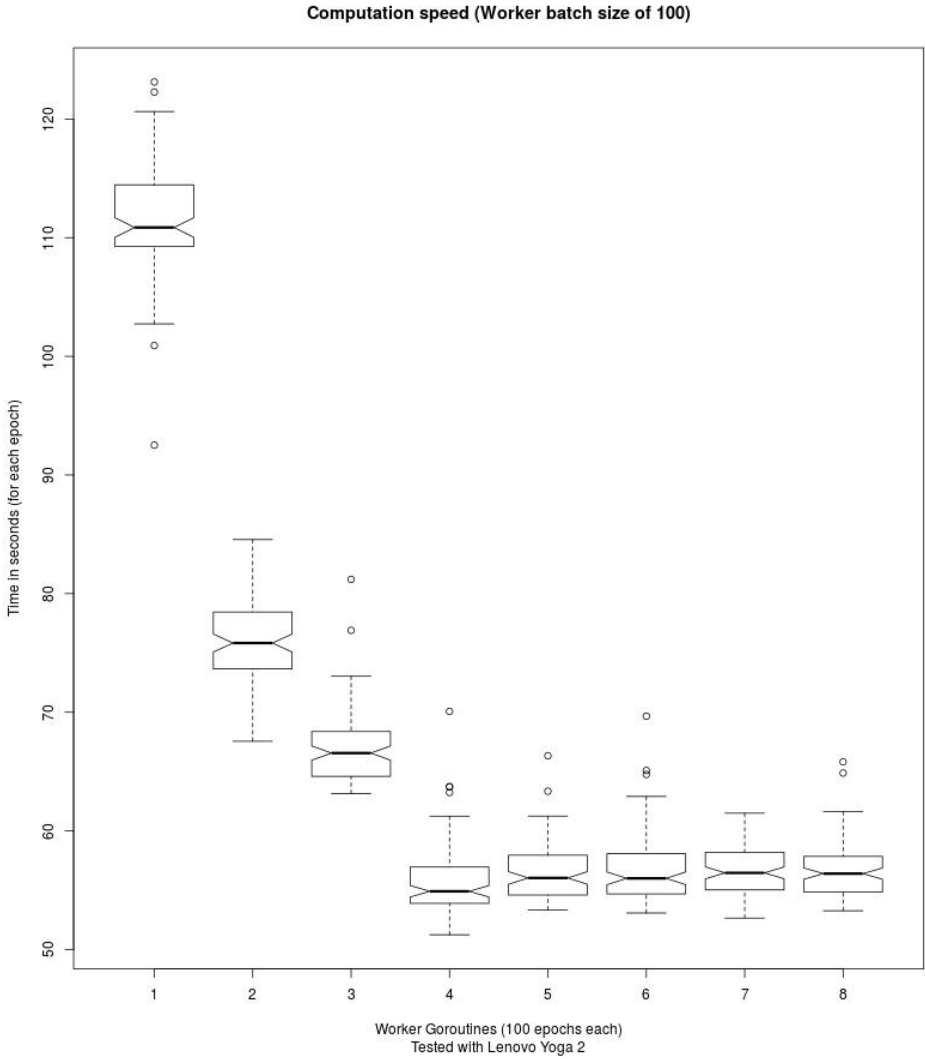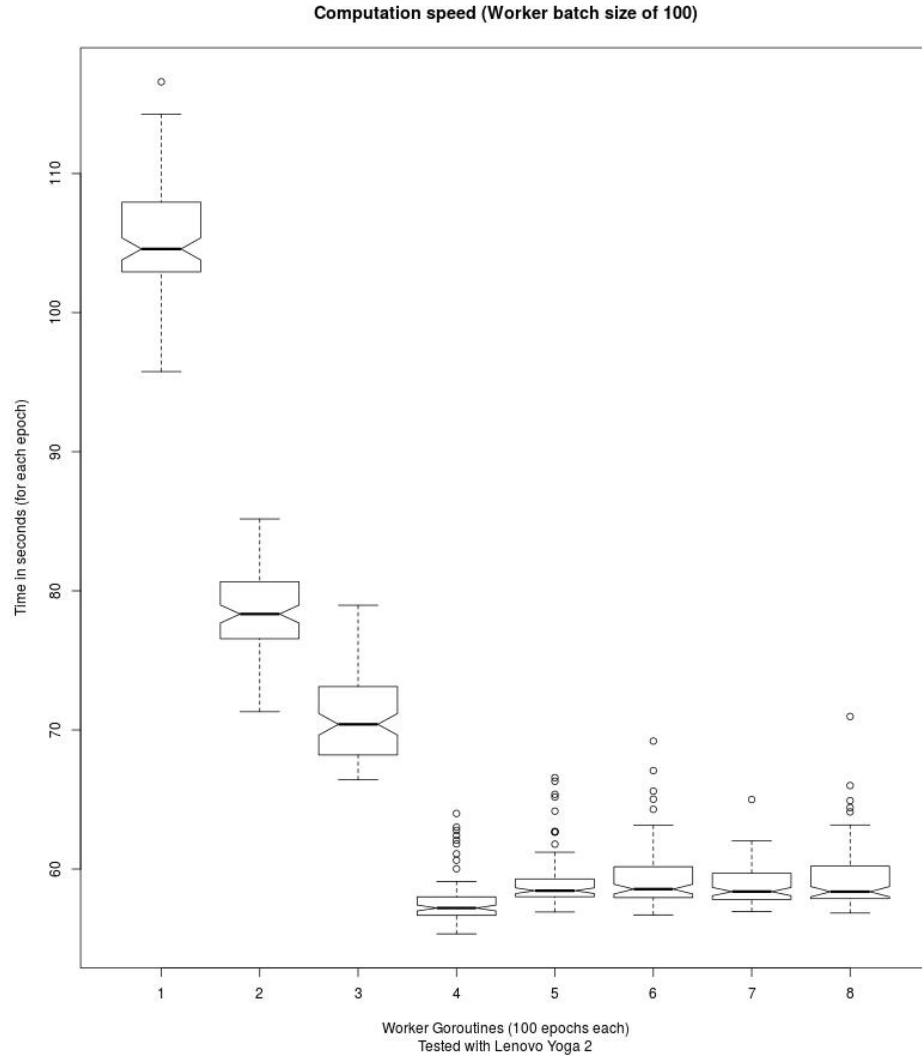

Figure 13: Benchmark before code changes

Figure 14: Benchmark after code changes

4.1.7   Never trust a single source

"Why is this important, when the objective is just to write code?" one may ask. The answer is pretty straightforward and was also a pitfall in the early stages of this work.

When implementing software to solve a certain problem, there are often different sources available. It is paramount to not trust a single source. If possible, look at the original studies, try to get the original data - but watch out for personal interests and possible skewed or even fabricated data. Therefore also look for meta analyses and systematic reviews, which look at a broader spectrum, the methods used, and other vital data points to find eventual outliers or problems with the data. Also don't trust sources that are given to you by a single source. Check it first. Everything else costs time and nerves: Only assuming, but not knowing, that the source is correct can lead to triple-checking code, math, and data, even though the error lies somewhere else. Two examples follow.

Many different sources in the web, including Wikipedia, managed to quote the correct source for linear unit activation functions, but used the old equations from the first two versions of the paper. There is a small effect on the derivative functions when x = 0. In example with ELU: The old and new equation are only equal when $\alpha = 1$. Otherwise, when x = 0, f'(x) should be α, not 1.

One trusted source - due to one University module handed out an excerpt without citing the source which was the basis for an assignment to calculate a forward, backward, and forward propagation by hand, and creating a simple neural network, which both got graded - used the updated weights as basis for the backpropagation. This lead not only to headaches[XVII], but also to interesting results.

[XVII]The network was very unstable - even the smallest parameter change lead to a massive drop which could occur sooner or later in testing. Sidequest: Find the error - is it code, math, or something else?

There was a larger accuracy dropFigure 15 when using multiple gorou-
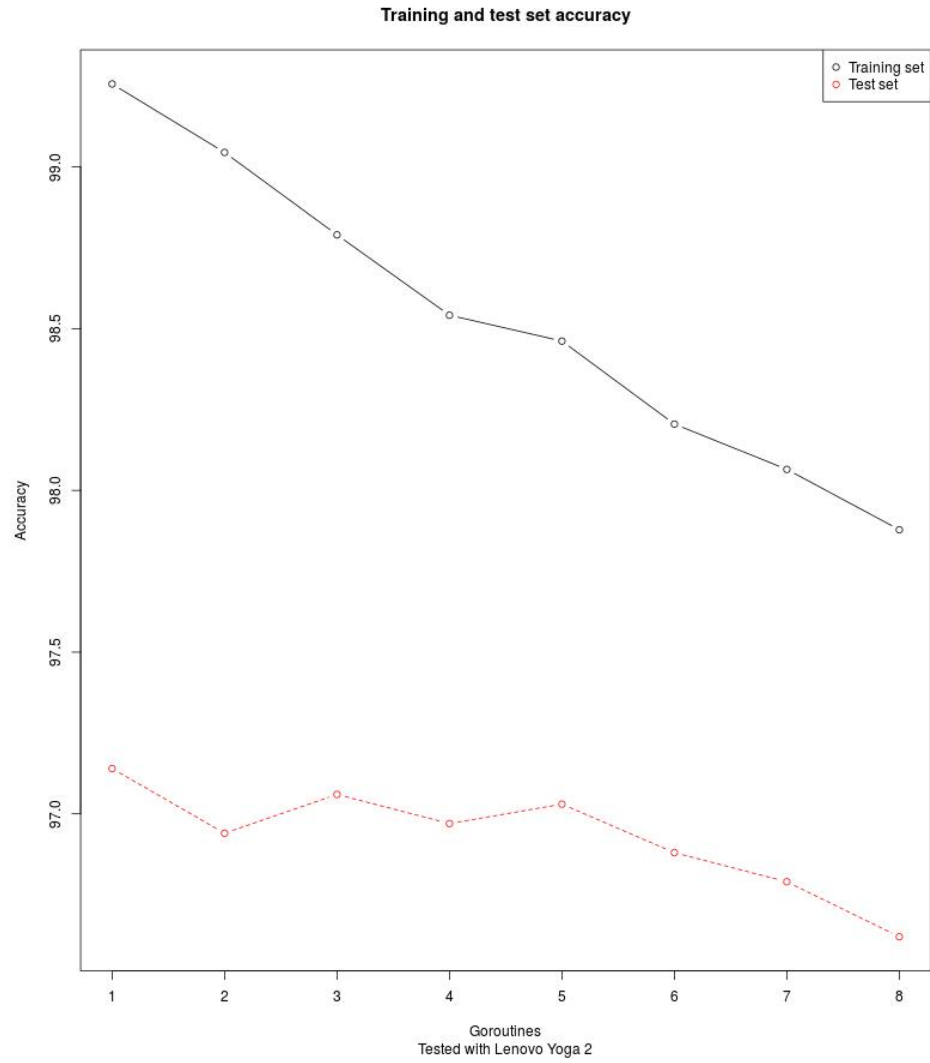tines.

Figure 15: Old Plot showing the accuracy decrease

Although it was possible to see when to tweak the parameters to gain a higher accuracy with a single coreFigure 16, there has been found no practical use of these values for the correct implementation - the hyperparameters vary widely, so they have to be tuned differently.
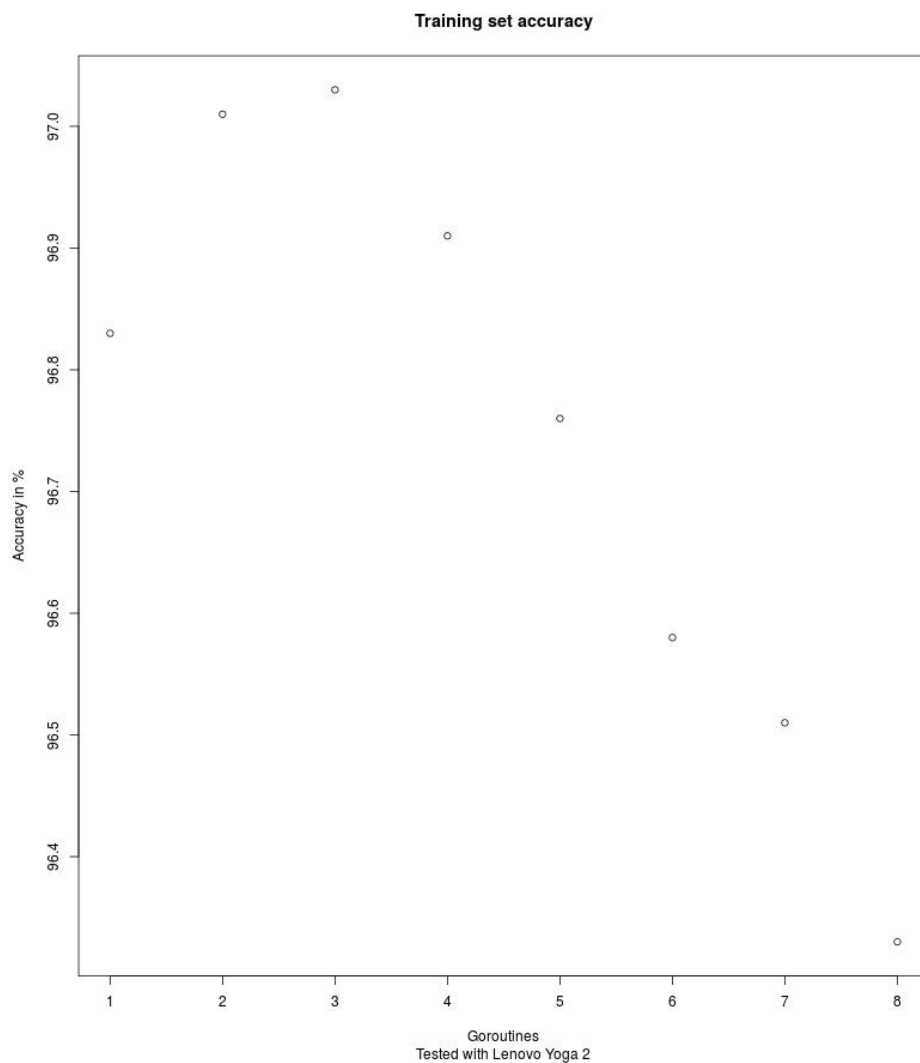

Figure 16: Accuracy with wrong parameters

Also mathematical sources are often not the best source for calculations in information systems. Math has been spared the problem of both errors due

to overflow and underflow (except when using calculators or information systems). Also there is no need to optimize for memory or computation speed.

The problem with trusting the wrong data has been solved with further research from different sources and consulting a mathematician to check if the partial derivatives and all formulas have been implemented correctly. The error has then been found very quick when checking against the standard reference [16].

4.2 Comment on Go

Go is a wonderful language to write code. Implementation and testing of the neural network seemed to be easier than with other programming languages. But go also has some drawbacks (as does any language).

The main annoyance were the "unused imports" bugs. Sometimes only certain outputs are needed for testing which will get dropped by the developer immediately afterwards. It's good that the Go compiler sees these oversights as errors, even though they are a huge annoyance. A probably better way would be if unused imports won't be tested in a debug environment, only in production. But this would have additional drawbacks when Go is used in environments where code quality is not highly valued.

Another annoyance is the "sudden 'bad file descriptor' of doom". Sometimes it's just a "data reader error: file already closed". It was not possible to pin down what exactly causes the error, only that it affects the file as a whole. Not even deleting and creating a new file with the same name helps to overcome that error. Further testing is needed.

An additional observation that can ruin ones day is, that the Go compiler for some reason accesses trashed files, at least under Linux. There is no problem when files are overwritten by new files. But if a file gets deleted, and a new one inserted instead, Go sometimes seems to try to compile the deleted files, which can lead to hard to trace errors. If there's, in example, an error where the Go compiler expects an integer value, the code provides an integer value, but recently a file with the same function expecting a double value had been trashed, simply empty the trash bin.

Another "hard to debug except when you know it" part is: "panic: runtime error: invalid memory address or nil pointer dereference". This error occurs

when the object has not been created with new(...). If it's further up in the code, i.E. some struct attribute, this error is not easy to find. When starting with Go that panic tells almost nothing about its nature.

Circular dependencies are not allowed. They can happen while refactoring code or when making some design mistake. It's good that Go does not permit them as they are a sign of bad software design.

The short variable declaration := is very handy. Go recognizes the type and assigns the value to the left hand variable. The best part: It won't break type safety[XVIII] which prevents weird behavior.

With the test coverage profiler it is easy to see the current code coverage. There is also the possibility to create test heat maps and to show the test coverage in the browser with highlighting good, poor, or not covered code parts[XIX].

There are memory and cpu profilers, and even a profiling tool[XX]. It is easy to list the top cpu or memory consumers or show a profiling web. Therefore memory issues can be found easily, as well as slow code parts.

Go uses function inlining which is a great method for speeding up code.

Goroutines are very lightweight[XXI]. As they're very efficient and only start to run if they get data from a channel, there's the probability of an application for parallelized neurons instead of only parallelized networks.

It's easy to find and fix race conditions with Go as it comes with its own race detector[XXII].

[XVIII]JavaScript type unsafety: When working with JavaScript then "1" + 1 - 1 would be... 10. "1" is a string, so it converts the 1 int to a string and concatenates it as + is also used for that purpose; but as there is no negative concatenation it then converts the "11" to an integer and substracts 1 which leads to 10. Even if the interpreter would subtract first, it would concatenate the 0 int to the "1" string, which would still be a "10" string.

[XIX]Go test coverage and html heatmap: `https://blog.golang.org/cover`

[XX]Go profiling: `https://blog.golang.org/profiling-go-programs`

[XXI]Currently 2, 4, or 8 kB per Goroutine, depending on the version, i.E. https://github.com/golang/go/issues/7514

[XXII]Usage of the race detector in Go: `https://golang.org/doc/articles/race_detector.html`

4.3 Conclusion

It has been learned how to use the programming language Go and about its parallel speedup possibilities. The main accomplishment of this work is to have managed to create a stable and fast neural network. The hardest part was to understand the mathematical concepts and ramifications behind neural networks and how to implement them software wise.

4.4 Future work

The main focus of this work was to see how the parallel speedup of a neural network behaves with the language Go. Due to time and resource restrictions only little derivations from the main focus were made. There are still ways left to make this neural network even more efficient, with higher accuracy, and so on. The current version could have some possible memory leaks. They will be fixed in a future version. As there will be further changes due to development and additional insights, the code will probably be refined and refactored in the future.

Some parts of the code are still untested - mainly file reading and writing. As they work as intended no additional effort has been made to get 100% test coverage in these areas. Here is room for improvement.

Optimization of the neural network would be the largest part of the future work. Currently it is only a simple network with Bias. It would be possible to implement momentum [19] and other artifacts to achieve higher accuracies. NADAM and other stochastic gradient descent optimization algorithms [18] could be implemented too.

Smaller changes will also include several options, in example if the user wants bias nodes, which error severity to log, and to choose different lambdas for the L1 and L2 Regularization in the elastic net. Adaptive learning rates [19] would be of interest too. Different loss functions, especially Cross Entropy Loss [20] will be implemented in the future.

There is an interest to look into Self-Normalizing Neural Networks [21].

Acknowledgements

5   References

[1] Yann LeCun, Corinna Cortes, C.J.B.: THE MNIST DATABASE of handwritten digits. `http://yann.lecun.com/exdb/mnist/index.html` Accessed: 2018-08-12.

[2] Google LLC: Go website. `https://golang.org/` Accessed: 2018-08-12.

[3] Google LLC: Go blog. `https://blog.golang.org/` Accessed: 2018-08-12.

[4] Google LLC: Go tour. `https://tour.golang.org/` Accessed: 2018-08-12.

[5] Multiple contributors, visible on `https://github.com/golang/go/wiki/Home/_history`: Github go wiki. `https://github.com/golang/go/wiki/` Accessed: 2018-08-12.

[6] Google LLC: Go version 1 release. `https://blog.golang.org/go-version-1-is-released` Accessed: 2018-08-12.

[7] Rogers, R.O., Skillicorn, D.B.: Strategies for parallelizing supervised and unsupervised learning in arti cial neural networks using the bsp cost model. (1997)

[8] Bottou, L.: Large-scale machine learning with stochastic gradient descent. In: Proceedings of COMPSTAT'2010. Springer (2010) 177–186

[9] Zinkevich, M., Weimer, M., Li, L., Smola, A.J.: Parallelized stochastic gradient descent. In Lafferty, J.D., Williams, C.K.I., Shawe-Taylor, J., Zemel, R.S., Culotta, A., eds.: Advances in Neural Information Processing Systems 23. Curran Associates, Inc. (2010) 2595–2603

[10] Zou, H., Hastie, T.: Regularization and variable selection via the elastic net. Journal of the Royal Statistical Society: Series B (Statistical Methodology) **67**(2) (2005) 301–320

[11] Glorot, X., Bordes, A., Bengio, Y.: Deep sparse rectifier neural networks. In Gordon, G., Dunson, D., Dudík, M., eds.: Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics. Volume 15 of Proceedings of Machine Learning Research., Fort Lauderdale, FL, USA, PMLR (11–13 Apr 2011) 315–323

[12] Maas, A.L.: Rectifier nonlinearities improve neural network acoustic models. (2013)

[13] Clevert, D., Unterthiner, T., Hochreiter, S.: Fast and accurate deep network learning by exponential linear units (elus). CoRR **abs/1511.07289** (2015)

[14] Ioffe, S., Szegedy, C.: Batch normalization: Accelerating deep network training by reducing internal covariate shift. CoRR **abs/1502.03167** (2015)

[15] Pascanu, R., Mikolov, T., Bengio, Y.: Understanding the exploding gradient problem. CoRR **abs/1211.5063** (2012)

[16] Goodfellow, I., Bengio, Y., Courville, A.: Deep Learning. MIT Press (2016) `http://www.deeplearningbook.org`.

[17] Elfadel, I.M., Wyatt, Jr., J.L.: The "softmax" nonlinearity: Derivation using statistical mechanics and useful properties as a multiterminal analog circuit element. In Cowan, J.D., Tesauro, G., Alspector, J., eds.: Advances in Neural Information Processing Systems 6. Morgan-Kaufmann (1994) 882–887

[18] Ruder, S.: An overview of gradient descent optimization algorithms. CoRR **abs/1609.04747** (2016)

[19] LeCun, Y.A., Bottou, L., Orr, G.B., Müller, K.R. In: Efficient Back-Prop. Springer Berlin Heidelberg, Berlin, Heidelberg (2012) 9–48

[20] Sadowski, P.: Notes on backpropagation. homepage: https://www. ics. uci. edu/˜ pjsadows/notes. pdf (online) (2016)

[21] Klambauer, G., Unterthiner, T., Mayr, A., Hochreiter, S.: Self-normalizing neural networks. CoRR **abs/1706.02515** (2017)