TALLINN UNIVERSITY OF TECHNOLOGY
Faculty of Information Technology
Department of Computer Engineering

IAY70LT
Hannes Kinks 132465 IASM

IMPLEMENTING NEURAL NETWORKS ON FIELD PROGRAMMABLE GATE ARRAY

Master Thesis

Supervisor: Peeter Ellervee PhD
Co-Supervisor: Siavoosh Payandeh Azad MSc

Tallinn 2015

TALLINNA TEHNIKAÜLIKOOL
Infotehnoloogia teaduskond
Arvutitehnika instituut

IAY70LT
Hannes Kinks 132465 IASM

NÄRVIVÕRGU REALISEERIMINE VÄLIPROGRAMMEERITAVAL LOOGIKAL

Magistritöö

Juhendaja: Peeter Ellervee PhD
Kaasjuhendaja: Siavoosh Payandeh Azad MSc

Tallinn 2015

Author's declaration of originality

I hereby certify that I am the sole author of this thesis and that no part of this thesis has been published or submitted for publication. All works and major viewpoints of the other authors, data from other sources of literature and elsewhere used for writing this paper have been referenced.

Author: Hannes Kinks

May 27, 2016

Abstract

The goal of the thesis is to provide an implementation for Feedforward Neural Network (FANN) on a Field Programmable Gate Array (FPGA), that would be able to do image recognition and classify subjects of a given set of greyscale faces. The FANN design is modelled first in software and a reasonable configuration is chosen. Afterwards the specifics and challenges of hardware implementation are analysed and the final FPGA synthesizable design is presented. The second goal was to test how does optimizing the network topology affect the performance of the hardware realization. In order to do that, network topology optimization and connection pruning methods were examined. The experimental results show that up to 64% of connections could be reduces in this case, without a major loss of network's accuracy. In effect, this decreases the number of clock cycles needed to find the result by 15% on FPGA.

The thesis is in English and contains 52 pages of text, 5 chapters, 23 figures, 8 tables.

Annotatsioon

Lõputöö eesmärk on välja pakkuda närvivõrgu realisatsioon väliprogrammeeritaval loogikal (FPGA), mis oleks võimeline teostama näotuvastust ja klassifitseerima isikuid etteantud must-valgete fotode põhjal. Närvivõrk on esmalt koostatud Matlab-i abil tarkvaras, selleks, et leida toimiv ehitus ja konfiguratsioon, mille põhjal oleks võimalik katsetusi läbi viia ning viimaks riistvaraline lahendus leida. Järgmise sammuna analüüsitakse riistvaralise realisatsiooni võimalikkust ja selle eripärasid. Viimaks esitatakse üks võimalik lahendus närvivõrgu jooksutamiseks FPGA-l, koos alamkomponentide ja süsteemi ülesehitusega. Lõpplahendus eeldab eelnevalt treenitud närvivõrku (*offline learning*), loeb sisendi sisemälust ning arvutamisprotsess toimub jadamisi. Võrreldes tarkvaralise lahendusega leiab riistvaraline tulemus värvivõrgu väljundi 13 korda vähema taktide arvuga, mis saaks olla eelduseks nii kiiremate kui ka ökonoomsemate närvinõrke kasutatavate seadmete loomiseks.

Lõputöö teine põhieesmärk oli katsetada erinevaid närvivõrgu topoloogiaid ja optimeerida ühenduste arvu. Tüüplahendused kasutavad sageli täielikult ühendatud võrgustikku, kuid funktsionaalse närvivõrgu jaoks ei ole enamasti kõik ühendused vajalikud. Iga ühenduse olemasolu tähendab aga ressursside kasutust, kas aja või riistvara hõivamise näol. Ühenduste optimeerimiseks katsetati kirjanduses välja pakutud kärpimis meetodit (*pruning*), kui ka enda poolset lähenemist geneetiliste algoritmi abil. Antud eksperimentidest selgus, et kuni 64% ulatuses on võimalik ühendusi kärpida, ilma, et olulist kadu närvivõrgu klassifitseerimistäpsuses ei esineks (kuni 10%). Kärbitud ühendustega närvivõrku jooksutati ka riistvara simulatsioonis. Kuna riistvara realisatsioon arvutas tulemusi jadamisi, siis riistvara hõive koha pealt erinevusi ei tekkinud, küll aga taktide arvu pealt hoiti kokku 15%.

Lõputöö on kirjutatud inglise keeles ning sisaldab teksti 52 leheküljel, 5 peatükki, 23 joonist, 8 tabelit.

Glossary of Terms and Abbreviations

FPGA	Field Programmable Gate Array
ASIC	Application Specific Integrated Circuit
HDL	Hardware Description Language
GPU	Graphics Processing Unit
CPU	Central Processing Unit
CLB	Configurable Logic Block
LUT	Lookup Table
LFSR	Linear feedback shift register
RAM	Random Access Memory
ANN	Artificial Neural Network
FANN	Feed-forward Artificial Neural Network
Genetic operator	Functions used in genetic computing to diversify the solutions (mutation) and combine them together (crossover). Mutation is unary and crossover is binary operation.
Genotype	Individuals full genetic information, set of rules that can be used to generate phenotype.
Phenotype	The individual itself that is the result of specific genotype.
PCA	Principal Component Analysis
2DPCA	Two-dimensional Principal Component Analysis
VHDL	VHSIC Hardware Description Language
FSM	Finite State Machine

Contents

List of Figures

List of Tables

1. Introduction

Even though computing power of processors has been increasing twofold every two years, they still do not seem to be a match for brains in many areas. Of course brain and computer intrinsically work through different principles and are not directly comparable. The motivation behind computers architecture has been to perform mathematical calculations while brains evolved to survive in its natural environment. Therefore more complex classification and estimation tasks, like speech and image recognition, planning and decision making, which brains can naturally solve, has proven to be a challenge for traditional mathematical approaches. At best, if computer is programmed accordingly to simulate brain's information processing, it can reach the level of an insect brain according to the approximation of Figure 1. Therefore, in order to engineer computers that are able to perform well in the real world, that is in its nature fuzzy and constantly changing, one of the approaches is to take inspiration from what already works the best - the human brain.

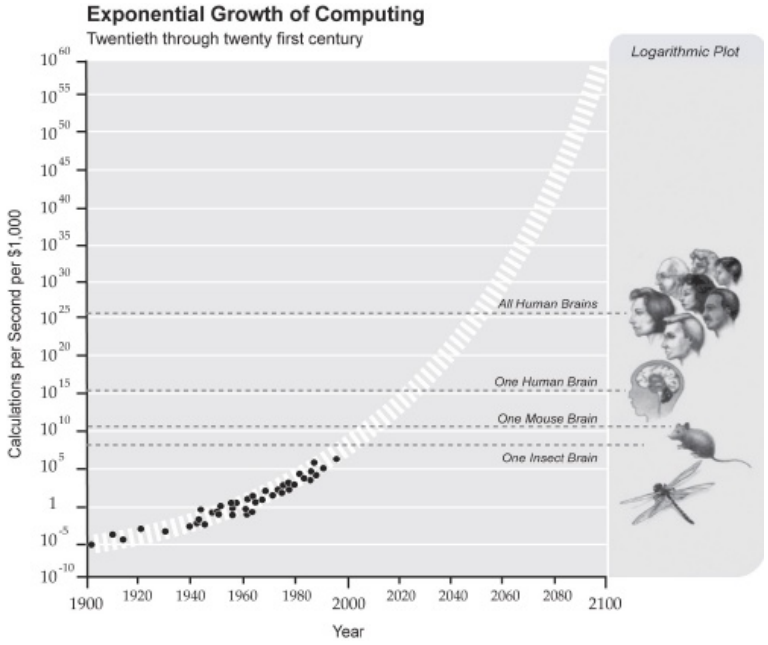

Figure 1. Exponential growth of computing [1]

1.1. Background and motivation

The topic of the thesis stemmed from interest towards the function of the brain and exploring the field of artificial intelligence. In addition from the supervisor's side there was

a general idea of experimenting with genetic algorithms to develop intelligent systems. In parallel an example project of implementing face recognition on a FPGA was done [6]. The challenges that arose from the project were the basis of the thesis at hand. As the number of configurable logic blocks and memory on a FPGA is limited, it is essential to choose efficiently tuned network for making the implementation feasible.

Even though state of the art machine learning techniques often involve highly interconnected Deep Neural Networks with large number of layers, the need for sparse and more simpler neural networks has not dissipated. Simple Feed-forward Neural Networks (FANN) can be of great use in embedded systems when classifying or predicting data with smaller dimensions. E.g. monitoring and security systems or smart home applications [7]. Furthermore, there are application areas, like embedded systems for space applications, which require minimizing the overhead of used subsystems as much as possible. Therefore, refining FANN for FPGA usage has a potential use in these areas, but there is currently a lack of formalized set of methods that would allow to do so.

1.2. Problem and goal definition

One of the problems with neural networks is the configuration due to their complexity. Selecting and tuning neural network hyperparameters, finding good architecture and data representation that works the best for given problem can be often very time consuming task that requires lot of manual trial and error. Even when a fairly good network has been found, it still usually contains a considerable amount of connection overhead. The strength of neural networks is that they can automatically adapt to fit the problem, based on the data that has been given, however the specifics of how the trained network finally solves it, is very complex to understand. More connectivity is therefore used that can be crucial for function, as the network itself balances out, which connections will be more important. In software implementations of neural networks, this overhead will not have a noticeable effect as parallel matrix multiplications are typically used. However, when implementing neural networks directly on hardware, it is not clear, how does this connection overhead affects the power consumption, speed and area.

The objective of the thesis at hand, firstly, is to provide guidance when designing simple Feedforward Artificial Neural Networks (FANNs) on lower end FPGAs. The final outcome will be a neural network hardware realization that can classify images of faces. Secondly, techniques of pruning the network are being explored with the intention of

minimizing speed and power requirements. Common existing methodologies and author's own approaches are described, experimented and compared with, giving an overview how neural networks could be optimized and configured for making FPGA mapping feasible.

The practical task is broadly divided into two parts: The first part concentrates on modelling the neural network in software to find a good configuration that the hardware realization could be based on. Reasonably minimal and accurate neural network is the prerequisite for an efficient FPGA implementation. Different network topology optimization approaches are being experimented with, where pruning and evolutionary computing is used. The results of these simulations can be used then in the hardware realization to compare whether there is any benefit from pruning connections.

The second part explains how the FPGA implementation was designed and what are the hardware specific challenges.

1.3. Methodology

A general overview will be given about the tools, methods and data used in this thesis. The main idea is to train a Feedforward Neural Network to recognize faces and use it as a benchmark to assess the performance of networks with different configuration. For the faces, Cambridge AT&T Laboratories database of faces is used, mostly known as *The ORL Database of Faces*. The images are in PGM format, greyscale, pixel values ranging from 0-255 and 92x112 pixels in size. The dataset consists of 40 subjects, taken in different times, with varying lightning and facial expressions [2]. All the images are taken as portraits, on the same background (Figure 2.

The data is fed through dimensional reduction algorithm, Principal Component Analysis (PCA). The training of the neural networks and experiments following it are all done in Matlab R2015a with the addition of the following toolboxes: Neural Network Toolbox, Optimization Toolbox, Global Optimization Toolbox and Fixed-Point Designer.

The hardware realization was done in VHDL using Vivado Design Suite 2015.4. For testing Xilinx Artix-7 XC7A100T[5] Field Programmable Gate Array (FPGA) on a Digilent Nexys 4 prototyping board was used.

During the development of the thesis the source code was managed by GIT version control

system. The whole project is available as a GIT repository [8].

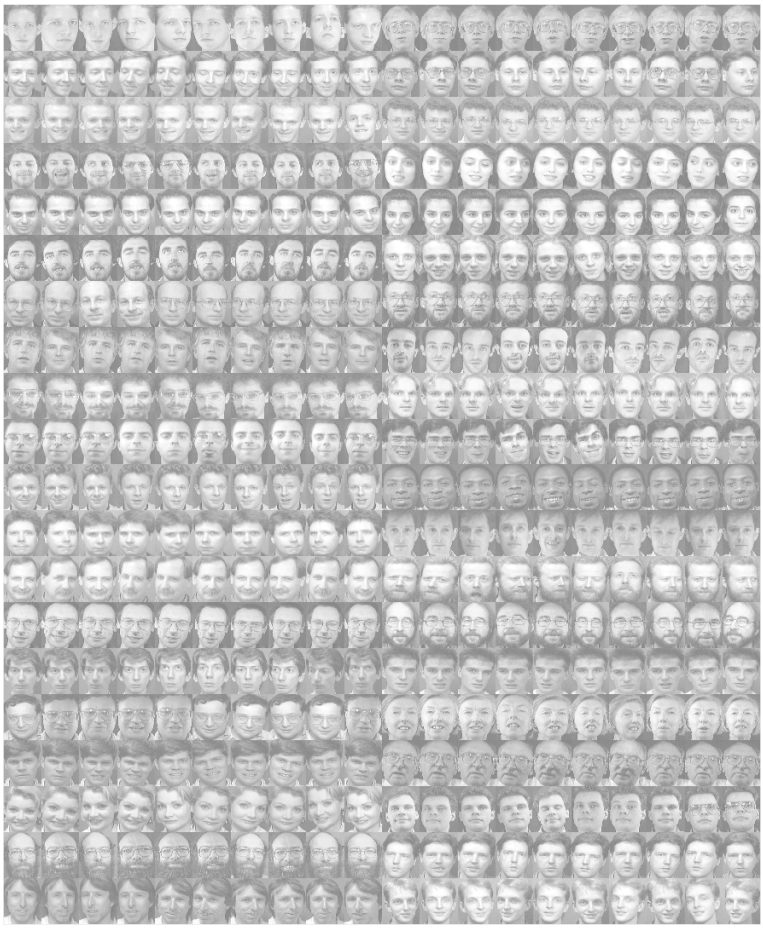

Figure 2. The ORL Database of Faces [2]

2. Theory

In the following chapters there will be an overview of the basics, brief background and the state of the art for the artificial neural networks, evolutionary programming and implementing them on FPGAs.

2.1. Artificial neural networks

Artificial Neural Networks (ANN) are a class of statistical models inspired by the brain research and the biological neural networks. The central idea of ANN is not to use feature engineering, where the rules and semantics of input data are previously specified by human. Instead, it can adapt and train itself based on given examples. It can be used to classify data, recognize patterns and predict. [9] Artificial neural networks became a new paradigm in 1980s and nowadays it has already proven useful in numerous applications like data mining, search engines, weather prediction, forecasting financial markets, monitoring systems, giving medical diagnosis, voice and image recognition etc. The latter includes also face recognition that is the goal of the thesis at hand.

2.1.1. Artificial neuron

The basic processing unit of a biological neural network is believed to be a neuron. Already in the beginning of 20th century it had been observed by anatomists of that time that the cortex of a brain has similar cellular structure all over it. It was known that each biological neuron cell consisted of cell body, dendrites and axons (Figure 3). Dendrites are carrying input signal to cell body and after reaching a certain threshold, neuron fires an output through axon. Neurons are connected to each other through synapses in between axons and the dendrites of other neurons. In 1978 Vernon Mountcastle proposed, based on his observations, that all of the brain regions are performing actually the same operation and the function that a specific region performs is related to the connections between neurons[10]. Therefore in principle any brain region can be trained to classify any type of information, for example visual recognition is no different from hearing in terms of the underlying mechanism. To illustrate that, there has been experiments done where newborn ferrets' brain has been rewired so that the eyes send their signals to ar-

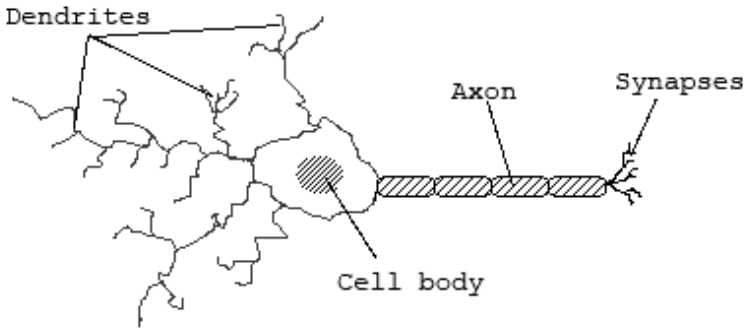

Figure 3. Structure of biological neuron cell

eas where hearing normally occurs. As a result these auditory areas develop functioning visual pathways instead. [11] When applying this knowledge to the artificial intelligence theory, we can make a presumption that pattern recognition can be effectively done on any input information, by having a great enough number of interconnected artificial neurons.

Perceptron These observations of that time were used for creating perceptron in 1978 by Frank Rosenblatt [12]. Perceptron is a simplified, artificial neuron, that takes in a vector of n inputs, which are being multiplied by their associated weights $\sum_{i=0}^{n} x_i w_i$ and gets the output y by feeding it to the activation function ϕ. This can also represented as a dot product of two vectors.

$$y = \phi\left(\sum_{i=0}^{n} x_i w_i\right) = \phi(\mathbf{w}^T \mathbf{x})$$

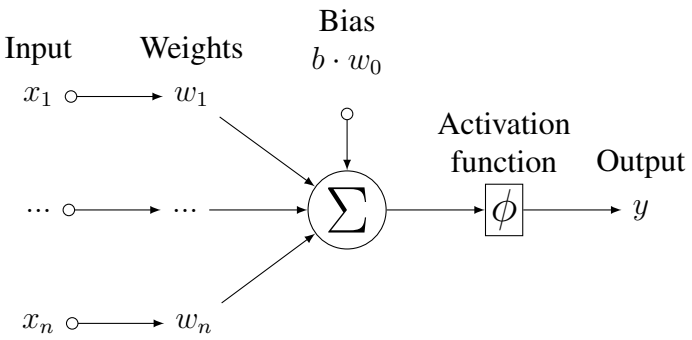

Figure 4. Artificial neuron with bias

Graphically it can be represented as shown in Figure 4.

Bias Artificial neurons often also have an additional bias input b, with a value of 1. The role of the bias is to provide a constant value in order to shift the activation function, consequently allowing the representation of all linear functions. For example, to illustrate that let us say that we have a artificial neuron with one input x and a bias. As $n = 1$ we can expand as follows

$$\phi\left(\sum_{i=0}^{n} x_i w_i\right) = \phi(x_0 w_0 + x_1 w_1)$$

if we have a linear activation function, then

$$\phi(x) = x \implies y = \phi(x_0 w_0 + x_1 w_1) = x_0 w_0 + x_1 w_1$$

and as the bias is always one we end up with a classical two variable linear equation

$$x_0 = b = 1.0 \implies y = x_0 w_0 + x_1 w_1 = x_1 w_1 + w_0 = ax + b$$

Activation function The weighted sum of neuron's inputs will be given as an argument to the activation function. By definition, activation function transforms neuron's input signals into output signal. Figuratively speaking, the activation function makes the final decision, if and how how much should the neuron react to the received input information. Classically there's three types of activation functions: linear, threshold (step) and sigmoid (soft-step) (Figure 5). Linear activation function can be used for example in linear regression and linear classification. Binary or step function either outputs 0 or 1 and can be used for classification of two sets. Sigmoid function is characterized by its S-shaped curve. Around the midpoint it has a exponential growth, however its boundaries are fixed where it decays to certain values. It is often used as it introduces non-linearity to the network and is easily derivable for weight learning. Based on the output range sigmoid functions divide into: logarithmic sigmoid, which is range from [0,1] and a scaled version of it, hyperbolic tangent sigmoid, that is in the range of [-1,1].

The normalized exponential is known also as a softmax function, as it represents a smoothed version of the *max* function[13]. It is not used on the output of the individual neurons like the previous functions. Instead, is often used in the output layers for multiclass classification. The function normalizes the outputs, essentially transforming levels of activation into a probabilities in the value range of [0,1] that sum up to 1.

- Identity (Linear)

$$\phi(x) = x$$

- Binary step

$$\phi(x) = \begin{cases} 0 \text{ for } x < 0 \\ 1 \text{ for } x \geq 0 \end{cases}$$

- Logarithmic sigmoid (Soft step)

$$\phi(x) = \frac{1}{1 + e^{-x}}$$

- Hyperbolic tangent sigmoid (TanH, Tansig)

$$\phi(x) = \frac{2}{1 + e^{-2x}} - 1$$

- Normalized exponential (Softmax)

$$\phi(x) = \frac{e^x}{\sum e^x}$$

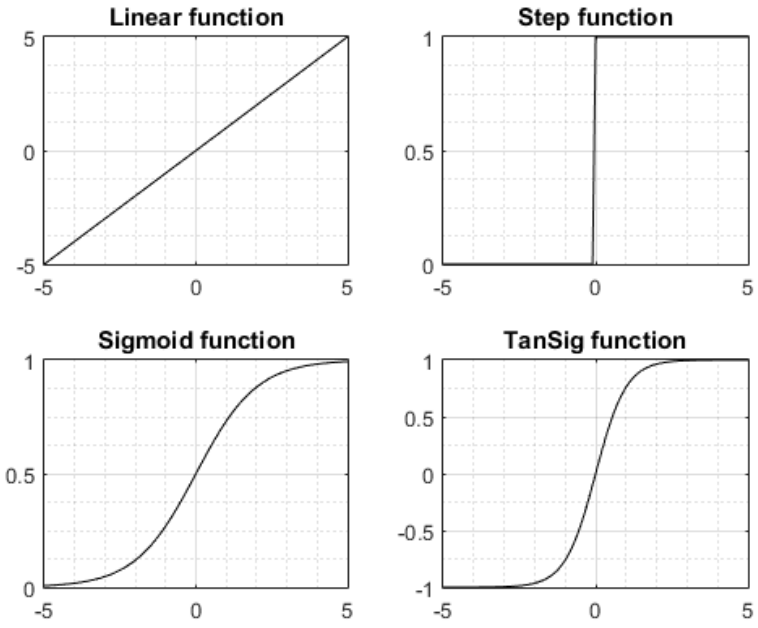

Figure 5. Typical activation functions used in ANNs

Perceptron itself can only do linear classification, which at the time of its invention was the main criticism over it. For example, it can successfully learn logical 'AND' and

'OR', yet classifying 'XOR' is impossible, as the classes of it are not linearly separable. However, if perceptrons are connected together into multiple layers, they can be far more powerful.

Learning In order to make the neural network functional, it needs to have a set of weights that correspond to the specific problem and set of data. To finds these weights, neural networks need to be trained. The training data will be fed to the network, the output will be evaluated against the expected output and the weights will be modified accordingly, so that the output will get closer to the expected output value. In case of a single neuron performing linear regression, it can be done as following [14]: Quadratic loss function is a common choice for measuring how well our network performs. This can be also thought of as a cost function that we have to minimize.

$$L(\mathbf{w}) = \sum_i (h(\mathbf{x}_i, \mathbf{w})) - y_i)^2$$

Where $\mathbf{w}$ is the vector of weights in the network, $h(\mathbf{x}_i, \mathbf{w})$ is the calculated output of a layer and y_i is the expected output. i shows the index of the current data sample. Next, to minimize the error different optimization algorithms can be used, however the most typical is the standard gradient descent:

$$\mathbf{w} = \mathbf{w} - \gamma \nabla_{\mathbf{w}} L(\mathbf{w})$$

Where the γ is the step size and $\nabla_{\mathbf{w}} L(\mathbf{w})$ is the gradient of loss function. From the step size it depends, how fast and how accurately the algorithm is able to determine the minima. If the step size is too big, the algorithm might not converge as it 'overshoots' the minimum value. If the step size is too small, it takes long time to reach the minimum. The correct step size is application and data dependent and choosing it usually takes a few tries. The gradient of loss function in case of a two input neuron can be written out as below, as the network function is $h(\mathbf{x}_i, \mathbf{w}) = w_1 x_i^{(1)} + w_2 x_i^{(2)}$.

$$\nabla_{\mathbf{w}} L(\mathbf{w}) = \left(\frac{\partial L(\mathbf{w})}{\partial w_1}, \frac{\partial L(\mathbf{w})}{\partial w_2} \right) = \left(\sum_i 2 x_i^{(1)} h(\mathbf{x}_i, \mathbf{w}), \sum_i 2 x_i^{(2)} h(\mathbf{x}_i, \mathbf{w}) \right)$$

2.1.2. Feedforward neural network

To do non-linear regression and classification, multiple neurons and layers are needed. Feedforward neural network (FNN) (Figure 6) is the earliest type of neural networks and the most basic way the neurons can be connected together and organized into layers. The output of one neuron is connected to the input of another neuron that is in the next layer. The first layer of artificial neuron nodes is called the input layer and the last output layer. The intermediate layers are called hidden layers. In feedforward neural network as the name suggests, the information moves only forward, from one layer to another. There are no feedback loops in the network opposed to recurrent neural networks (RNN).

The forward propagation of the signals is fairly simple as outputs of neurons are simply passed along to the next layer's inputs. However, for training we need to compare the network's output with the expected value and update the weights in the whole network. This is done by transmitting intermediate errors backwards, thus leading to the name *backpropagation* algorithm [15]. Instead of propagating the inputs forward, we propagate the errors backward and update the weights in the process.

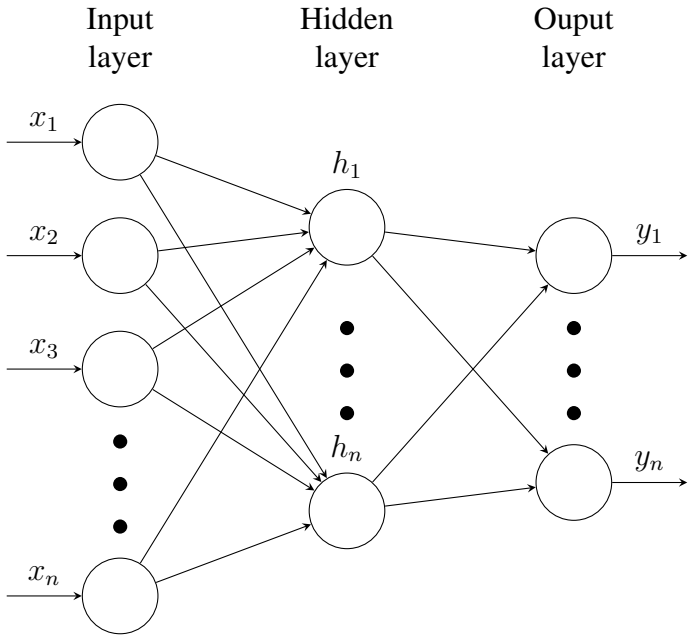

Figure 6. Feedforward neural network

2.1.3. Constructive Learning

The classical use of neural networks is done by deciding upon the architecture, setting up the connections between neurons and layers statically before starting the training phase. This however, does not guarantee the minimal network size needed. Constructive (or generative) learning algorithms have another approach, where the network is started off very small (usually with single neuron) and grown by adding neurons until satisfactory solution is found [16]. The key benefits of this approach are the following [17]

- In addition to weight space, exploring the space of network topologies and thus overcoming the limitation of fixed topology.
- Potential to construct a minimal solution, which matches the intrinsic complexity of the underlying learning task.
- An approximation of the expected case complexity of the learning task.
- Trade-off possibilities (between the network size and accuracy for example.)
- Incorporating previous domain knowledge by learning construction on a simpler task and applying the topology on a new, related task. [18]

2.2. Evolutionary computation

Evolutionary computation is a subfield of artificial intelligence characterized by using techniques inspired from Darwinian principles and natural evolutionary processes. Techniques categorized into the field include evolutionary algorithms like genetic algorithms, differential evolution, evolutionary programming, neuroevolution and also algorithms that are based on some naturally observed behaviours already emerged from biological evolution, like swarm intelligence, ant colony optimization, artificial life and bees algorithm.

2.2.1. Genetic algorithms

The origins of Genetic Algorithms are contributed to John Holland who published it during 1970's. Evolutionary algorithms are hugely inspired by evolution of lifeforms in the

nature, using the idea of survival of the fittest and knowledge obtained from genetics. The algorithms are used as a search heuristic mostly in optimization problems and machine learning. They are more robust than deterministic search algorithms as they are able to filter out some level of noise in the data and adapt to changes in input. One of the problems can be getting stuck in local minima. Presently EA successfully being used in areas such as computer-automated design[19], fault diagnosis in hardware [20], software engineering etc.

John Holland's genetic algorithm is known in literature as Simple Genetic Algorithm (SGA) and it has the following components [21]:

- population of individuals
- individuals encoded as binary strings
- fitness function
- genetic operators: crossover and mutation
- selection mechanism

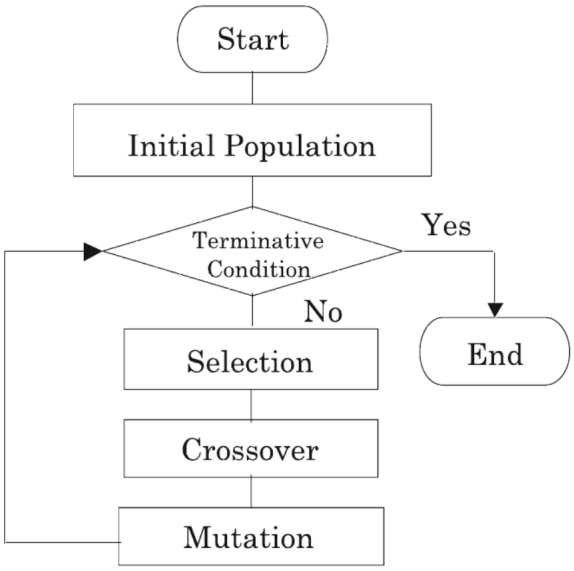

Figure 7. Simple Genetic Algorithm [3]

The algorithm of SGA can be seen on figure 7. It begins by generating initial population, which consists of a set of individuals, each individual representing a possible solution to a problem . Individuals are encoded as a finite length vector of bits, which corresponds to chromosome that consists of biological genes (Figure 8).

After generating the initial population, the best solutions are selected using a fitness func-

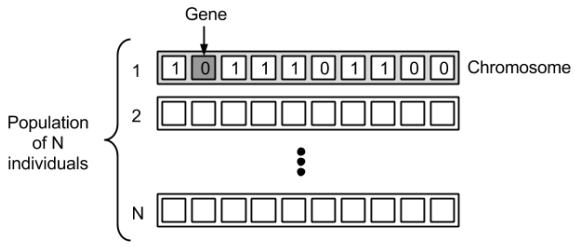

Figure 8. Population of Simple Genetic Algorithm

tion, which determines individuals who performed better than the others. The selected individuals will be then allowed to pass on their genes to next generation by organizing them into pairs and combining their chromosomes using crossover operation (Figure 9). With this operation there is a great chance that the attributes which made the parents the best individuals are being carried over to the offspring and combining genes from both of them can produce even fitter individuals.

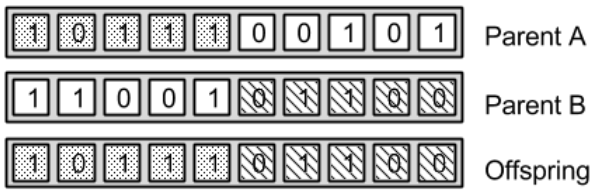

Figure 9. Crossover operator

In addition to crossover there is a low probability of random changes happening in the genes, called mutation. Mutation inhibits premature convergence and it helps maintain diversity in the population. [22]

2.3. Field Programmable Gate Array basics

Field Programmable Gate Array (FPGA) is an integrated circuit designed in a way that it can be reconfigured after manufacturing. Due to the possibility of directly describing hardware with HDLs, specifically for the task at hand without much overhead, their performance is much higher than using CPU or GPU for solving the same tasks. Even though Application Specific Integrated Circuits (ASICs) have even higher computational capability and efficiency, they are expensive to manufacture. Therefore FPGA's flexibility, coming from the ability to be reprogrammed multiple times, makes them often the choice of platform for prototyping hardware or accelerating demanding computing tasks. FPGA's architecture commonly consists of an array of configurable logic blocks (CLBs)

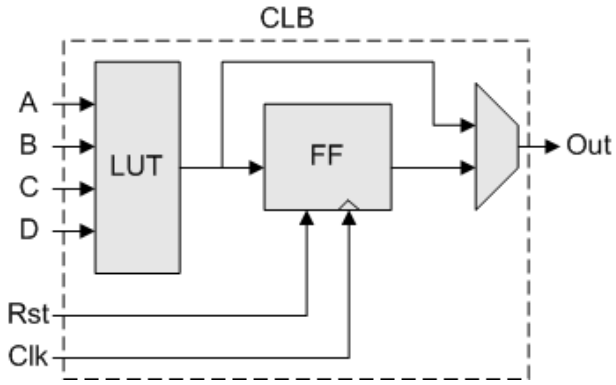

Figure 10. Typical logic block [4]

(Figure 10), which are surrounded by configurable interconnection structure. CLB typically consists of a Lookup Table (LUT) with 4 inputs which essentially is a truth table that can be defined to behave as any 4-input combinational function. LUTs themselves are typically built out of SRAM bits to hold the configuration memory (LUT-mask) and multiplexers that are used to select the according bit to be driven to the output.

For specific FPGAs' the CLB architecture and even the terminology varies. E.g. Xilinx divides the CLB further into slices and logic cells. [23]

The CLBs of most Xilinx FPGA's can also be configured to behave as so called *Distributed RAM* that spreads out over number of LUTs rather than being located in a single dedicated block. This gives them flexibility, however they are not area efficient and rather small. In addition to logic blocks there are also a number of dedicated areas of *Block RAM* which cannot be configured for other functionality, but are larger in size. Which RAM to use depends on the memory requirements - for small sized memories the distributed RAM is better as the usage of block RAM would be waste of space. On the other hand using distributed RAM for bigger sized memories would cause extra wiring delays and the available amount might not be sufficient. Also the reading of block RAM is synchronous while distributed RAM is asynchronous.

2.4. FPGA implementation of neural networks

Parallelism in neural networks To fully exploit the power of neural networks, they should be parallelized akin to their biological counterpart, but it can be made parallel on hardware in different ways. In general, the only categorical statement that can be made is

that, except for networks of a trivial size, fully parallel implementation in hardware is not feasible - virtual parallelism is necessary, and this, in turn, implies some sequential processing [24]. Therefore, it needs some analysis at which stage we make the computation happen parallel. According to [24] the types of parallelism are the following from higher level to lower level:

- Training parallelism - Parallel training sessions running simultaneously.
- Layer parallelism - Multilayer networks layers are processed in parallel
- Node parallelism - Each individual node is processed in parallel.
- Weight parallelism - During the computation of weights, multiplications can be done in parallel.
- Bit-level parallelism - Increasing word size of individual processors or making communication between different functional units bit-parallel.

Weight multiplication in hardware One of the fundamental problems with implementing parallel processing ANNs on FPGAs is due to the large number of connections between neurons. To calculate neuron's input values, multiplication is required and making it fully-parallel on the weight level it would mean one binary multiplier for each input. If there is a fully connected layer with n inputs and l neurons, it requires $n \times l$ multipliers. As we increase the input dimensionality for more complex problems and keep the number of neurons in proportion to the inputs, the growth of multipliers needed becomes quadratic.

There are broadly two ways to make the implementations of ANN feasible: either decrease the number of multipliers and lose in parallelism and performance, or decrease the complexity of multipliers and lose in accuracy [25]. Typical solution for example is to use time-division multiplexing (TDM) to share one multiplier per neuron across it's inputs [26].

Another approach, proposed in [25], is the use of stochastic computing, which uses probabilistic properties of bit streams to find approximations. To multiply for example operands 0.12 and 0.3 together, it can be done as following: Let there be two streams of random

bits, A and B

$$A = \{a_0, \ldots, a_n\}; a_i \in \{0, 1\};\ n, i \in \mathbb{N}$$
$$B = \{b_0, \ldots, b_n\}; b_i \in \{0, 1\};\ n, i \in \mathbb{N}$$

The random sequence of bits will be generated with a given probability according to the operands. So the probability of 1 in the first stream will be 0.12 and for the second stream 0.3

$$p = P_{a_i}(1) = 0.12$$

$$q = P_{b_i}(1) = 0.3$$

The product of these two probabilities is equal to the probability of 1 in an output stream of logical AND taken from A and B.

$$p \times q = P_{a_i \wedge b_i}(1) = 0.36$$

In order to get a good estimation of the frequencies of ones in the stream and therefore accurate enough multiplication result, the n has to be sufficiently large. Traditional binary multiplication needs a state machine and an adder, while the complexity of the whole operation is $O(n^2)$. With the stochastic method we need to generate random bit stream, for which LFSR can be used, and only an AND gate to find the product.

Routing Arguably the main bottleneck for building a fully parallel ANN on hardware would be the routing limits of FPGA. The problem stems from the FPGA architecture itself and due to the high interconnectivity of ANNs themselves, as their size increases, the routing limits are hit fast. As process technology improves, FPGA vendors are able to build larger arrays of these identical tiles. As they do, routability degrades because proportionally more interconnect is required on large device [27]. The additional problem is that the routing utilization is difficult to measure and estimate when the FPGA capacity limit is reached due to the interconnect usage [28]. Practically it means that at some point the speed advantage the parallelism should provide, will be lost to due routing delays.

2.5. Data representation and precision

An important aspect to consider in a NN hardware implementation is the data representation - the number format and bit length that is used for the inputs, weights and activation function. Mainly the format of weights is the trade-off point between accuracy of the network and the hardware implementation costs, because that in turn decides the complexity and area needs for multipliers and activation function. In commercial computers floating point arithmetic is typically used as they can provide much higher accuracy in dynamic ranges - with the use of exponents very large or small numbers can be represented. However by raising the magnitude, loss of precision will occur. Fixed point representation on the other hand has a more limited precision and data range. The advantages are that the precision and absolute error always stays the same, which is needed in some applications e.g. finance. Secondly, fixed point arithmetic in hardware is also more closer to integer operations - simpler, more area-efficient and faster. When the application at hand requires arithmetic only in small fixed range, then fixed point implementation on FPGA can be more feasible.

For neural network FPGA implementations the fixed point representation is often chosen because of the lower implementation costs. Although the use of floating point have been researched [29], it has been found that fixed point numbers with precision of 16 bits for weights and 8 bit for activation function are sufficient [30], [31]. Also it should be noted that learning requires more precision, as the back-propagation errors can more rapidly accumulate.

To hold the data in memory, block RAM should be used as Distributed RAM would be too small for non-trivial sized networks, including the approximate network needed for image recognition.

2.6. Activation function implementation

Computation of activation functions directly by its formula is often not feasible in digital systems due to the high area requirements and delay. Instead, approximation methods can be used to trade off precision. Piecewise Linear (PWL) Approximation and Lookup Table (LUT) Approximation will be discussed more in detail, however other methods also exist like e.g. Truncated Series Expansion.

1. *Piecewise Linear Approximation* uses a series of linear segments to approximate the activation function [32]. Different PWL schemes exist: A-law based approximation [33], Approximation of Alippi and Storti–Gajani [34], second-order approximation of Zhang, Vassiliadis and Delgado–Frias [35] and Centered Linear Approximation (CRI) [36].

   As an example, the CRI is a recursive computational scheme for the generation of PWL. The simplest initial logarithmic sigmoid approximation with CRI is defined by three line segments (Equation 1) and as the algorithm goes over its iterations the segmentation increases as seen on Figure 11. Depending on how much precision is needed, the sigmoid approximation can be divided into more segments, decreasing the error.

$$H(z) = \begin{cases} 1 & \text{for } z \geq L \\ \frac{1}{2} \cdot (1 + \frac{z}{2}) & \text{for } -L > z > L \\ 0 & \text{for } z \leq -L \end{cases} \tag{1}$$

   Where L is the given range in which the segmentation is carried out.

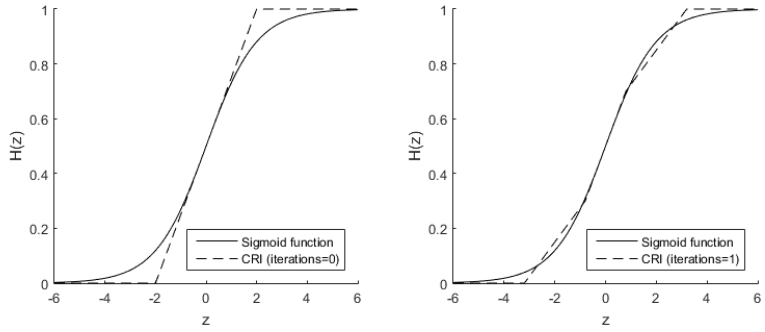

Figure 11. Sigmoid function Piecewise Linear Approximation with CRI (L=2). On the left the initial approximation with three line segments and on the right approximation after one iteration, resulting 5 line segments

2. *Lookup Table Approximation* maps input values to uniformly distributed set of output values. This method results in a design with better performance in terms of speed, as no arithmetic operations are needed. However, the area or memory requirements would be higher due to the need of storing the mapped values. Relation between the area requirements and precision is exponential [37], which makes this method impractical if high precision is needed. The approximation for a LUT with a width of 4 bits can be seen on Figure 12. In terms of average error, this method also outperforms PWL schemes according to [38].

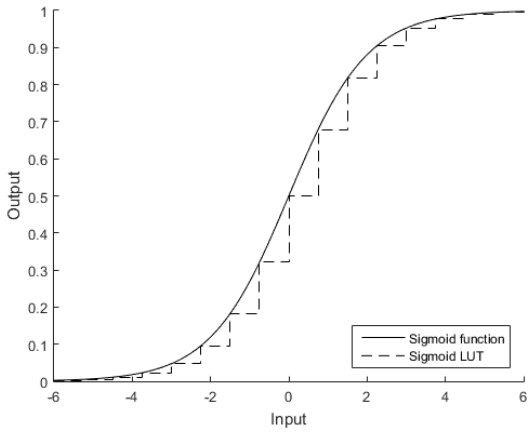

Figure 12. Sigmoid function Lookup Table Approximation

3. Face recognition task

3.1. Neural network models

In various stages of modelling neural network accuracy in software, two different models were mainly used. One was mainly a comparison reference and the other one was used for more precise simulation.

For evaluating the results of the experiments made, a reference network neural network was taken, which worked already fairly well. It was generated with Matlab's built in function 'patternet' and it is very close to the neural network used in [6]. The basic structure of the network can be seen on the first example of Figure 3. The inputs were preprocessed beforehand with 2DPCA to decrease the input dimensions. There are fully connected hidden and output layer, both with 40 neurons, using tansig activation functions and a bias. In addition normalization methods were used on the input data beforehand. The average test accuracy of 10 experiments when using the ORL dataset was 83.5%. However, in the best cases up to 98% accuracy can be achieved. The dataset was divided into training and test set with 8:2 ratio. From further on, this neural network model will be referenced as Matlab NN.

Matlab Neural Network Toolbox allows creating custom networks while specifying relations between layers, input sources, activation functions and allows modifying the weights, however specifying individual connections is non-trivial and would still require getting into the source code itself. For example, when setting the weights to zero and essentially

	Matlab NN	**Coursera NN**
Activation fn	Tansig	Logsig
Training	Scaled conjugate gradient	Gradient descent
Preprocessing	2DPCA	2DPCA
Layers	2	2
Hidden neurons	40	40
Avg. accuracy	83.5%	85%
Best accuracy	98%	92.5%

Table 1. Differences between Matlab and Coursera NN implementation

cutting them off, the default training methods will still retrain them. Due to these limitations a custom neural network implementation was written as it was deemed to be easier than rewriting the source code of existing tool. The ANN implementation is based on the lessons of Coursera's Machine Learning course [39]. It is a simple two layer feedforward network, with gradient descent back propagation learning and regularization. The average test accuracy over 10 experiments was 85%. In the best cases, the accuracy was up to 92.5%. The dataset was divided into training and test set with 8:2 ratio as before. For the purpose of testing optimization and pruning methods the lower accuracy in the best cases was considered less relevant at this point. This neural network model will be referenced as Coursera NN.

The main differences (and similarities) between these two models can be seen in Table 1. The input given for both networks is normalized beforehand in the range of $[0, 1]$ and preprocessed with 2DPCA. After that the values are being normalized once again, this time with *mapminmax* function that maps the values in the range of $[-1, 1]$. The choice for this range does not have a good argumentation over for example the range $[0, 1]$ other than simply performing slightly better. Possibly because the activation function's point of symmetry is where $x = 0$ and the inputs get then more evenly distributed in the range where function derivative is the highest.

3.2. Experiments with optimizing the network

Common approach with neural networks is to make fully connected layers and let the training phase utilize the connections as needed. This means that a number of connections can be redundant. The question that arose, was whether to include the individual connections and weights optimization into the GA. When combining ANN and GA together, then essentially two different optimization methods are working in parallel. If the

training of ANN already modifies weights, the addition of GA does not have any substantial value. When looking into the weights of a trained neural networks, it is very unlikely for any weight becoming zero due to the working of back propagation. However, they can get very close to zero and insignificant. At this point they could be removed if necessary. The first experiments goal was to see how pruning affects the performance of ANN in general and whether there is a reason to proceed with more complex, evolutionary optimization methods.

Another issue to consider is the way that the calculation of ANN weights will be implemented. State of the art approach of running ANN on general-purpose computers is to use matrix operations to speed up the process of calculating outputs in fully connected layers. For further acceleration this can be parallel computed on GPU for example [40]. The practice of engineering individual connections between layers is therefore unnecessary as the matrices are representing fully connected layers and disconnecting a synapse would just mean zero valued weight in a corresponding location. This would not however bring any computational gain.

Based on that reasoning it is rational to optimize the individual connections only if its number has an effect on the performance. That is in the case if computing the weight updates separately. If computing them one by one serially, then it affects the time and if done fully parallel, the area.

Initially the vectorized approach was focus on, where matrix operations are used, therefore in this experiment the individual connections were left out and the topology was explored only on layer level.

3.2.1. Rounding the weights

The experiment was done by taking mean of 10 training results and rounding down the weights to nearest ten thousandth, thousandth, hundredth and so on, as described in Algorithm 1. This essentially rounds the insignificant weights to zero and by reducing the

accuracy, the amount of memory necessary to hold the weight values reduces as well.

```
for i=1:n of experiments do
    net = train(net);
    tempnet = net;
    for j=0:4 do
        tempnet.weights = floor(net.weights * pow(10,j))/pow(10,j);
        acc = validate(tempnet);
    end
end
```

Algorithm 1: Rounding down the weights

It can be seen from the Figure 13 that rounding weights down to hundredths has a very marginal effect on the accuracy of the network (increasing the error by 0.25%). As the weights with zero values can be considered unconnected, this will mean the removal of around 2% of the connections. Therefore, the gain from pruned connections itself is low with this approach, however it shows that there can be room for optimization when pruning further with better methods.

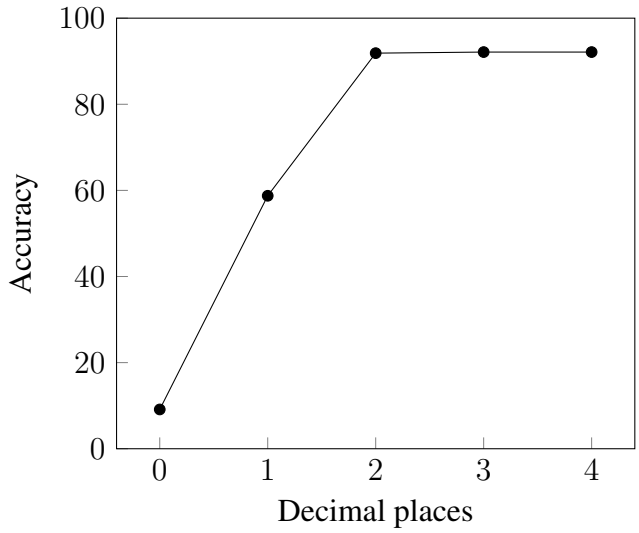

Figure 13. The effect of rounding connections with floor on the accuracy of the network

3.2.2. Pruning insignificant connections

This experiment was done by using the methodology proposed in [41]. The Coursera NN was first trained, then pruned so that more important connections remain and finally

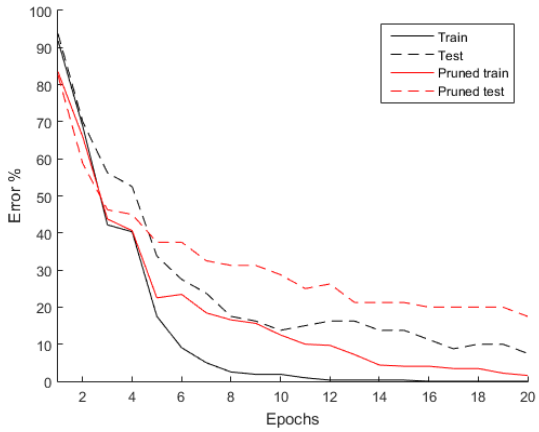

Figure 14. An example result of pruning connections

trained once again to balance out the weights for the lost connections. The pruning was done by changing weights that were below a certain threshold to zero. The accuracy of pruned connection compared to the original is plotted on Figure 14.

3.2.3. Exploring layer connections with genetic algorithm

Another approach is using genetic algorithm to optimize neural networks for the task at hand. The intention was to find a network for implementing on hardware, that has a better performance and more optimal topology. Focus here was mainly the general topology - connections between layers, activation functions and the number of neurons, while individual connections and more specific hyperparameters were left out. Table 2 represents the variables being optimized or 'gene' in the context of genetics. To explore topologies with different number of hidden layers, the GA was given an argument m for maximum number of layers to test. This also determines the length of the gene, as for example layers' connection matrix has a quadratic growth when the number of layers is being increased. After that the gene's first field designates how many layers are actually being used. This however means that also a great number of fields in the gene would be left unused if the number of layers is less than the maximum.

Before evaluating each individual with a cost function, the values from generated gene will be read and a neural network will be built based on it. After that it trains the network, measures the time of training and finally the results are evaluated by the cost function. It can happen that from the generated gene no valid network can be constructed, in this case a high cost value will be assigned to eliminate them from the gene pool. The cost is

Field	Number of layers	Number of neurons	Bias connections	Transfer functions	Layer connections	Input connections
Field size	1	m	m	m	m ×m	m
Value type	Integer	Integer	Binary	Enumerated	Binary	Binary
Description	Number of hidden layers used (out of the possible number m) in the current individual.	Number of neurons in each hidden layer	Specifies in which layers the bias input is being used.	1 - tansig 2 - logsig 3 - purelin 4 - softmax	Specifies the connections between hidden layers	Specifies to which hidden layers are inputs connected to

Table 2. Description of variables optimized

calculated as following:

$$C = \begin{cases} 1000 & \text{for invalid network} \\ t + (1 - \frac{e}{n}) \cdot 100 & \text{for valid network} \end{cases}$$

where
t - time of training (s)
e - number of errors
n - number of classes

First of all, to make the networks comparable with the reference network, it was tested whether the same example with similar results could be generated using by giving the gene as an input.

The GA optimization process generated different architectures with good performance, though compared with hand engineered reference, there was no significant gain. In Table 3 there can be seen the generated networks and their corresponding classification accuracies both with raw data and preprocessing with 2DPCA.

1. reference network
2. GA generated
3. 2nd individual with removed feedbacks
4. GA generated

In general the results tended to converge still to very simplistic structures, not too different from manually configured standard designs. Part of the problem here can be also the

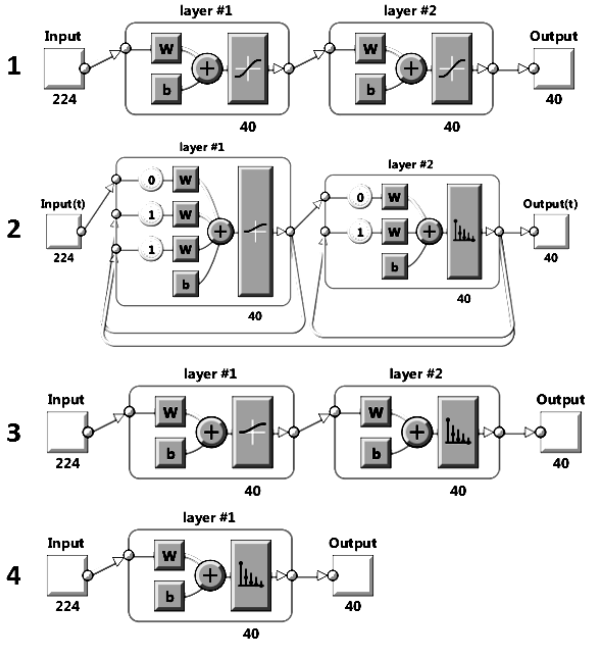

Figure 15. Chosen individuals for comparison. The numbers below the layers show how many neurons are in that layer.

	1	2	3	4
2dpca	93.38 %	94.58 %	94.40 %	94.10 %
raw	88.48 %	93.40 %	93.70 %	44.70 %

Table 3. Results of optimizing the structure of the ANN with GA. Columns represent the ANN designs in comparison given in Figure 15. The values given show the percentage of correct test data classifications.

choice of the data set. It is possible that with more diverse data set, with higher variance, the optimal structure would be more complex.

In addition the best performing individuals often had still redundant or questionable properties present. E.g. feedback loops or layers without outputs. Because the cost function did not prohibit such phenomena, once they appeared, they persisted into later generations.

For the purpose of implementing simple and minimal neural network on a FPGA, the last GA generated individual (Figure 15) could be of use when coupled with feature extraction. It has only one output layer with softmax activation function and has no hidden layers. Compared to the reference network, it therefore needs 40 neurons less and the weights associated with it, while showing similar performance.

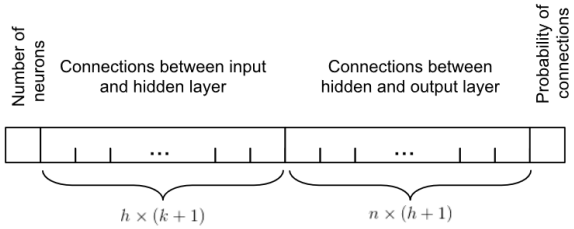

Figure 16. Gene layout used for finding optimal balance between performance and number of connections

3.2.4. Optimization of individual connections with GA

Genetic algorithm was also applied on individual connection level to find a good trade off between network size and prediction accuracy. Due to the need of having control over the neural network's connections, the Coursera NN (Section 3.1) was used.

The gene used for this experiment consisted of the fields shown on Figure 16. First field specifies the number of neurons as an integer, then values a_i in the range [0,1] will follow for each connection and each layer. The very last field gives the probability of connections forming ($P(connection)$), that is used to compare the individual connections. The actual connectivity is then decided, based on these probabilities as seen on Algorithm 2.

if a_i => $P(connection)$ **then**
  $connection_i$ = true
else
  $connection_i$ = false
end

Algorithm 2: Finding connectivity of the network

The necessity for the 'base' probability value, that decides whether the connection will be made, is to have a higher variance in the population. If the connection would be taken directly from the gene, the mutation in the GA does not happen often enough to create the necessary variance. As a result the better performing individuals were quickly sorted out and became highly similar, depending on the random genes they started with. Therefore, when running the experiment multiple times, the results were completely different each time. Another possible approach to solve this problem might have been to make the mutation rate higher, but then it might have been more like a random search.

The gene defines one specific individual, with specific connections that are reproducible.

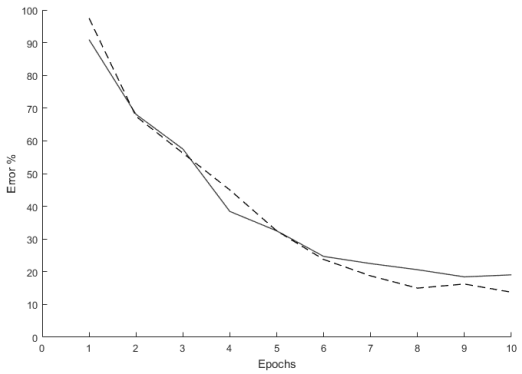

Figure 17. Performance of a partially connected network found with GA

However, the resulting cost can be different each time, due to taking network error and training time into account. As the initial weights are randomly initialized each time, the trained networks always differ slightly from each other. Also the training time depends on CPU utilization at that specific moment.

Calculation of cost function was done as following:

$$C = \alpha \cdot \text{connectionsPercentage} + \beta \cdot \text{errorPercentage} + \gamma \cdot \text{trainingTime}$$

Where α, β and γ are coefficients for modifying the weight of each component. Connection percentage is a value in the range [0,100] that shows how many connections between neurons were kept intact compared to the fully connected network. The fully connected network would have the biggest cost with value of 100. Error percentage shows how many misclassifications happened during the testing phase, with values in the range of [0,100]. In case none of the input data could be classified correctly, the cost would be 100. Training time was the time in seconds that took to train the neural network. Coefficients can be used to increase the importance of one or another factor in the optimal solution. It was observed from the experiments that the β coefficient should be considerably higher for finding networks with acceptable accuracy (>70%).

An example of the result found with this method was a network with 17% of the connections intact. The performance of such network can be seen on Figure 17, with test accuracy being at 86%. To have a better grasp on the optimized network, the connections can be visualized as seen on 18. It shows a graph of individual connections between input (224 nodes) and hidden layer (40 nodes) with total of 1502 synapses. If it was a fully connected, the number of synapses would be $224 \cdot 40 = 8960$.

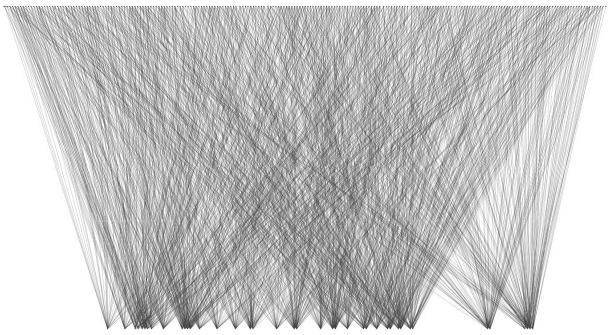

Figure 18. Example of an optimized connection between input and hidden layer visualized as a graph

3.3. Approximation of activation function

For finding a good approximation of activation function, LUT and CRI methods (explained in more detail in Section 2.6) were experimented with. For the LUT approximation it was necessary to find out what was the table size needed to hold the values without losing too much in accuracy of the network. In addition a decision needs to be made for the range in which the LUT stores the values. For example, when choosing the range [-5,5] and LUT table width of 4 bit, the function $H(z)$ will be split into 2^4 segments in the given range, while $H(z) = 0$ when $z < -5$ and $H(z) = 1$ when $z > 5$. The Figure 19 shows the approximations with different range and bit widths. The choice of trade-off should minimize bit widht and range, while maximizing the accuracy of the network. The example highlighted gave 95.25% accuracy over the whole dataset, while regular sigmoid function gave 97.75% (calculated without restrictions on range or bit width). The 2.50% loss of accuracy can be considered marginal and therefore a reasonable choice. As the ranges are encoded in binary, the range can actually be increased up to 8, because the bits needed for encoding the numbers would be the same as for 6.

Secondly, the CRI approximation produced accurate results in a neural network already with the initial 3-segment function described in Equation 1, without the need for any further iterations. The two methods are compared with the regular sigmoid function on a test dataset on figure 20. In this sample run all three cases the accuracy of 92.5% was reached, with CRI method even having a slightly better result than the original. From the training process it can be seen that the LUT method lags behind the CRI, but for the trained network both have relatively similar accuracy.

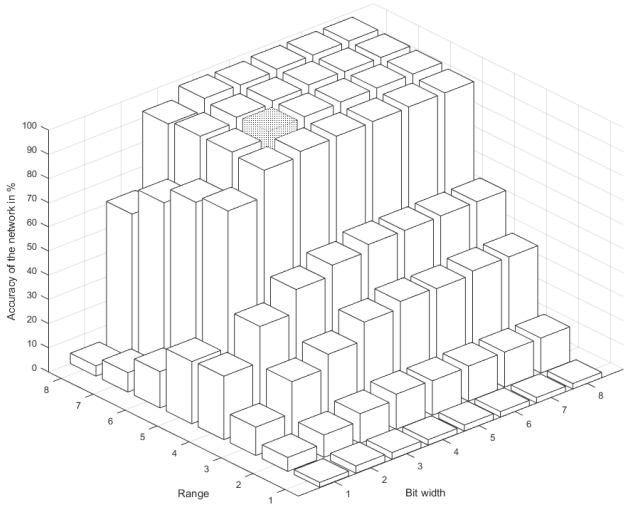

Figure 19. Accuracy of neural network using LUT sigmoid approximations with different LUT sizes and ranges. A good choice would maximize the accuracy, while minimizing range and bit width.

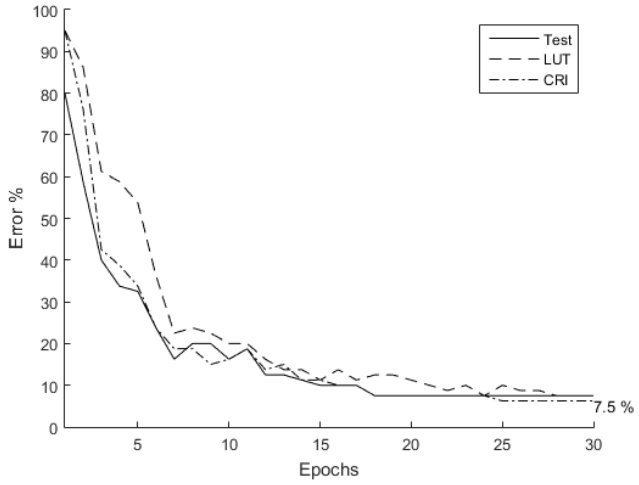

Figure 20. Comparing neural network test accuracy with LUT and CRI approximation.

4. Hardware realization

The final part of the work at hand was to implement a neural network on a FPGA. The first reason of this was to take advantage of the speed increase that the use of FPGA can provide. Secondly, as the objective was to experiment with optimizing the network, hardware realization was needed for observing the results in power consumption and speed.

The hardware was written in VHDL, which is a language for describing digital electronic systems. The choice of VHDL over for example Verilog or SystemC came from having previous experience with it. VHDL is designed to write down the structure of a system and its functionality using typical programming language forms. What makes VHDL (and other hardware description languages) different from other programming languages is the notion of time and concurrent statements. This makes it possible to also simulate the design in software to verify its functionality. Finally, the description can be used to synthesize hardware or configure a FPGA.

4.1. Design process

The design process can be formally best described with a Waterfall or V-model in terms of development model used in this project. Even though it was not chosen consciously in the beginning, it was a somewhat natural choice as the development was carried out by an individual and it was small sized project with fixed requirements. Also Waterfall and V-model are still the development models used most often in the hardware design [42] opposed to agile methodologies in software development. V-model puts an emphasize on testing and verification, which was also a substantial part of the development in this case.

V-model usually starts with a concept of operation and a practical need for the system by someone. Whether it is achieved is checked by client or user acceptance. In this case the goal was to answer a research question instead, which means the development model is somewhat mixed with scientific method instead. As an concept there was research questions posed in the beginning which by the end should be either verified or dismissed by the end by analysing the results of experiments.

As a result the design process with those mixed aspects can be visualized as seen on Figure 21.

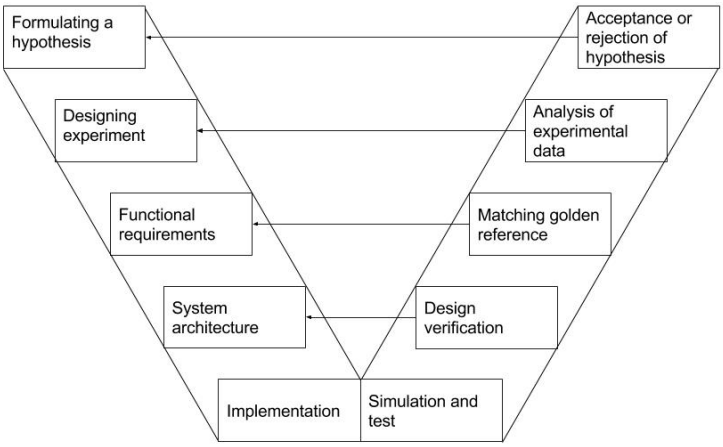

Figure 21. The development process of the hardware realization

4.2. Design choices

Offline against online learning The hardware design is limited at this point only to the neural network structure itself, without having capabilites of learning. The training process has to be carried out on software and the weights of a trained network must be transferred to the BRAM of the FPGA. Therefore, it is called offline training, as opposed to online training (for example [43]), where learning happens on FPGA itself. The choice was made with the intention to start with crucial components of the system and expand in the future as needed. For running the experiments with modified weights, the offline learning is sufficient and better suited for its simplicity.

Parallelism When designing the hardware for ANN, there are many ways to approach the parallelism and there are also many limitations that come with it as it was explained in the section 2.4. The main parallelization approaches applicable for the design at hand were the node, weight or bit-level parallelism. The advantages of pruning weights should come apparent mostly with either fully weight parallel design or when weight calculation is fully serial. In the first case the trade-off is mostly area-wise and in the second case it is the time we are trading off. In addition it can affect the power consumption. The fully weight-parallel design would quickly hit the area and routing limits on the FPGA, therefore this approach was discarded and serial computation of weights was chosen. For node level parallelism calculations could be done on $2..N$ neurons in parallel, while the weight calculations are still happening in a serial manner for each neuron. Each parallel process would need its own multiplier in this case. Even though this approach could be feasible and possibly a path for further research, for first implementation it was decided

to make the calculations on one neuron at a time. Therefore, the only parallelism present in this design is bit-level.

Precision For representing real numbers in hardware, fixed-point was used due to easier implementation on the FPGA. The number format used is **Q**3.4 with a signed bit (total 8 bits). More in-depth reasoning about the choice of fixed-point representation can be seen in Section 2.5 and 3.3. The word length 8 bits has been said to be enough for activation function. During the multiplication of weight and input, the word length will be doubled for getting accurate enough results. The integer part was decided based on how big range does the activation function need to give good enough results. From the Figure 19, it can be seen that the range $[-6; 6]$ is large enough and this can be represented with 3 bits + signed bit. This way the actual range will be even higher: $[-8; 8]$. Finally, the fraction part is left with 4 bits. Although a certain number format was chosen and has been taken as an example here, the implementation was made in a generic manner so that the word and fraction sizes could be changed if needed.

Activation function Logarithmic sigmoid function approximated by using LUT was used for the activation function. It was decided on as the LUT approximation was found to be sufficiently accurate (tested in Section 3.3) and as FPGAs' CLBs use LUTs anyway to implement the logic, it should be a straight forward approach. However, CRI approximation would be a feasible option as well.

4.3. Architecture

Overview The general overview over the design can be see on Figure 22. All the data is stored in three BRAMs: image, weight and results. The main computational unit is the *neuron*, which calculates the results for all the neurons one by one. The neuron outputs are stored in *results BRAM*. The multiplexer before the *neuron* is used to select the input depending which layer is being calculated. The inputs are initially read from *image BRAM* until the first layer has been calculated, after that the new inputs will be the results of last layer and read from *results BRAM*. The neural network design uses bias as well, therefore for the first input of each node, 1 needs to be read. During the calculation of last layer, the outputs of neuron will be read by *max* component, which finds the output with the highest value (essentially playing the role of a competitive transfer function). The index of the

maximum value will correspond to the predicted class. The *display* component is then used to convert the index for displaying on the 7-segment display. The *controller* runs the Finite State Machine (FSM) controlling the whole calculation process with the trigger of *start* input signal.

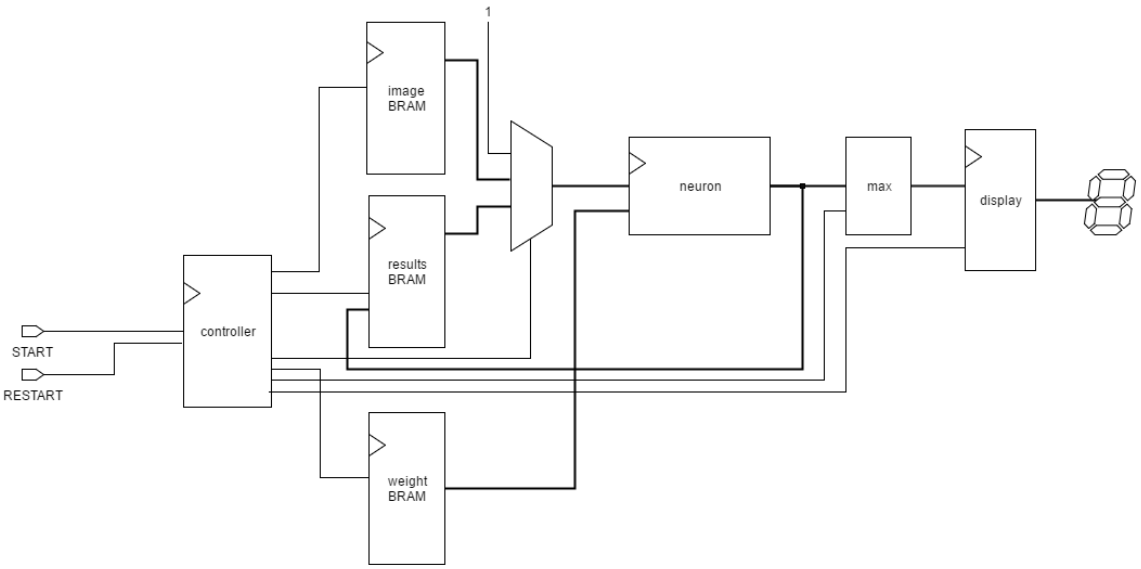

Figure 22. General view of the datapath of the design

Controller The controller is a Moore FSM with the states shown on Figure 23. The beginning state is the *idle* state and until no input signal is given, the FSM stays in this state. Upon giving the start signal it moves to the *init* state, where BRAMs are being enabled for reading. In the next *read* state input and weight values will be read from BRAM. Next *calculate* state follows, where the neuron is enabled for weight and input multiplication. Until all the inputs have been read, the states will alter between *read* and *calculate*. During each iteration a counter will be incremented that keeps track of the inputs. If the counter has reached to the number of inputs there are, the FSM continues with *write_back* state. During the write back, the output value received from the neuron will be written back into results BRAM. After that, in the *next_node* state the previous sum and input counter will be reset back to 0. Also another counter for keeping track of neurons is being incremented at this point. Until the counter value is below the number of neurons in the layer, it continues with the *read* state to start with the process of calculating the result for the next neuron. When all of the outputs of neurons have been calculated and stored, the FSM moves on to the *next_layer* state. Similar pattern repeats here: the neuron counter gets reset back to 0 and layer counter is being incremented. After all the layers have been gone through, the FSM returns to its initial *idle* state. After returning to

the *idle* state, the found result will remain on the display. When reset signal is given, then the FSM switches to *reinit* state where the result value is cleared from the display.

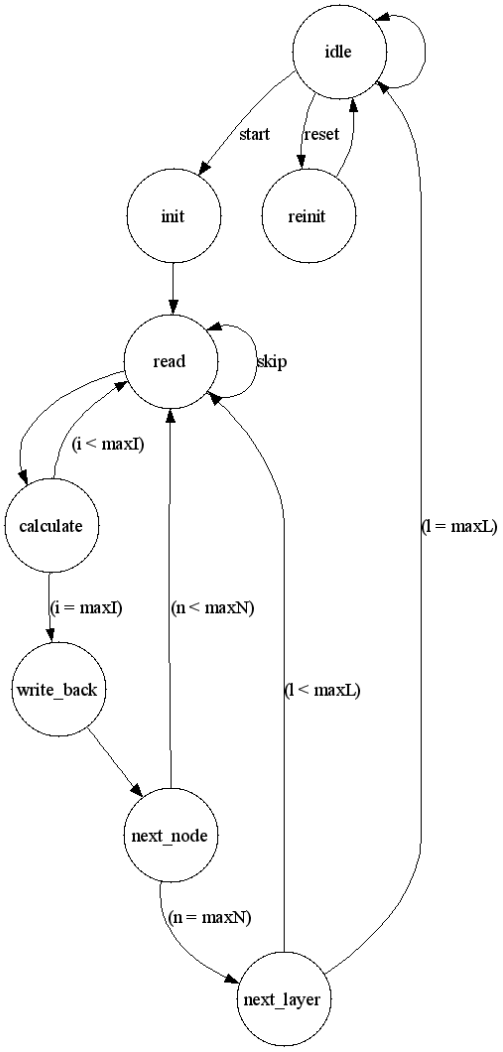

Figure 23. Controller's state diagram

Neuron The functionality of the neuron can be summarized with the VHDL code snippet on Algorithm 3. If the neuron is enabled, the input will be multiplied by weight and cumulated to the sum according to $\sum_{i=0}^{n} x_i w_i$. Finally, on the enabling of activation function the total sum will be input to the sigmoid function LUT approximation and corresponding output will be found.

During the computation of the product, the precision gets doubled to 16 bits to hold the multiplication result. This precision is also kept while finding the sum. When the sum is given as an input to the activation function, it gets truncated back to 8 bits. The truncation

is made in a symmetrical manner to keep the **Q**3.4 format. In the event of overflow (when the sum is either larger than 8 or less than -8) the output will be saturated to the minimum/maximum value.

```
sum_in = sum;
activation_function: logsig port map(clk, act_ena, sum_in, y);
process (clk) is
begin
  if (rising_edge(clk)) then
    if (res = '1') then
      sum <= (others => '0');
    elsif (ena = '1') then
      sum <= sum_in + (signed(x) * signed(w));
    end if;
  end if;
end process;
```

Algorithm 3: Neuron VHDL code

4.4. Simulation and verification

The ANN design uses offline learning and the input is preloaded into memory. Therefore, in order to simulate the design, the input and weight BRAM had to be initialized with values. The values are specified in a 'coefficient' file (a text file with a .coe extension) which Xilinx Vivado Design Suite uses to load the memory locations. To generate these 'coefficient' files Matlab scripts were used.

To get the weight coefficients, Coursera network model (explained in Section 3.1) was trained and the adjusted weight values were converted to hexadecimal numbers and exported to the 'coefficient' file. The same process applies when networks with pruned connections are tested with the hardware design. For getting the image data into a 'coefficient' file, similar script was made in Matlab. A number identifying an image is given to the script and it will fetch the image, feed it through the 2DPCA preprocessing and write the data into the coefficient file, naming it according to the image identificator so that it can be easily checked whether the prediction matches the image class.

The golden model To verify and debug the hardware design, a reference model was made in Matlab to compare with. The reference model has to follow the hardware imple-

mentation's algorithm exactly, use the same data format and output intermediate values for comparing. Algorithm 4 describes the general algorithm that the ANN follows, with the addition of writing intermediate values into file. The files can be then used for debugging by reading them in during simulation and comparing with values that hardware design produces. Another option is to write the same intermediate values of the hardware design during the simulation into files and use software like *diff* tool to compare the results side by side. Both approaches were tried and due to the limitations of VHDL simulation in Vivado, the second one worked the best.

```
for l = 1:layers do
    for n = 1:neurons do
        sum = 0;
        for i = 1:inputs do
            sum = sum + input * weight;
            output = sigmoidApproximation(sum);
            write_file(input);
            write_file(weight);
            write_file(sum);
        end
        write_file(output);
    end
    if l == layers then
        if output > max then
            max = output;
            max_index = n;
        end
    end
end
classification_result=max_index;
```

Algorithm 4: The algorithm for reference model

4.4.1. Xilinx Artix-7 FPGA overview

For testing the synthesized design, Xilinx Artix-7 XC7A100T[5] was chosen as the target. The choice was due to the access to this specific trainer board and its support for Xilinx Vivado Design Suite. Artix-7 uses 28nm technology and the specifications of the

FPGA Model	XC7A100T
Logic cells	101440
Slice LUTSs	63400
DSP48E1 Slices	240
Distributed RAM (Kb)	1188
Block RAM (Kb)	4860
I/O Pins	300

Table 4. Xilinx Artix-7 XC7A100T [5]

XC7A100T model can be see on Table 4. Generally speaking the Artix-7 series is designed to be a low-cost and low-power option in the FPGA market. Due to its small form factor and low-power performance requirements it can very well be used to develop battery powered image recognition devices for example.

4.5. Results

In this section results are presented to answer the following questions:

1. How does the hardware realization perform compared to software realization?
2. How does the connection pruning affect the performance?

The aspects of performance under focus are:

- operation speed
- area utilization
- power consumption

Operation speed To compare the operation speed of the hardware implementation, experiments were run first in software, on a Intel Core i7-4820K CPU running on 3.70 Ghz. The average speed of calculating the neural network outputs was measured in Matlab and based on that the number of clock cycles was calculated. The experiments were run both

	Connections	Time	Clock speed	Clock cycles
Software original	100.00%	63.00 μs	3700 Mhz	$2.331 \cdot 10^5$
Software pruned	45.78%	33.00 μs	3700 Mhz	$1.221 \cdot 10^5$
Hardware original	100.00%	55.80 μs	324 Mhz	$1.808 \cdot 10^4$
Hardware pruned	45.78%	47.47 μs	324 Mhz	$1.538 \cdot 10^4$

Table 5. Comparison of speed of the software and hardware design, both with pruned and unpruned connections.

with fully connected network and a pruned one, with 45.78% of the connections remaining. It took approximately 230 thousand clock cycles for the CPU to calculate the result and the pruned design was almost twice as fast.

When running the same network on FPGA, the actual time that took to calculate the result was not that much shorter than in software due to the fact that the clock speed of the FPGA was 13 times slower. Nevertheless, this on the other hand means that the clock cycles needed for reaching the result was also 10 times less on the FPGA and more power efficiency can be achieved without compromising the speed.

When using the pruned weights and modifying the design to skip the zero values, 85% faster operation can be achieved on the FPGA. The current 'connection skipping' technique in hardware was implemented fairly straight-forward, thus if more effort was invested on pipelining the calculation process, additional gains could be possible. The full table of comparison can be seen on Table 5.

Area and Power consumption In the comparison of area and power consumption the differences were not observable between the fully connected and pruned design. The reason is that the design processes the connections serially and by increasing or decreasing the number of connections, the area does not change, only the operation time does. In terms of the power consumption, in theory there should be differences as the switching activity would be lower with the pruned connections. However, the difference was not observable in the power analysis of Xilinx Vivado Design Suite, even though simulation activity was taken into account when executing the analysis. This means that the decrease in power is so minuscule in this case that it gets rounded off and can be discarded. One way to confirm could be scaling up or duplicating the design on the FPGA in order to

		Power (W)
Dynamic	Clocks	0.002
	Signals	0.002
	Logic	0.002
	BRAM	0.003
	I/O	0.002
	Total	0.011
Static		0.097
Total		**0.108**

Table 6. Power consumption of the neural network design

Resource	Utilization	Available	%
LUT	381	63400	0.60 %
FF	199	126800	0.16 %
BRAM	4	135	2.96 %
IO	18	210	8.57 %
BUFG	1	32	3.13 %

Table 7. FPGA utilization

make the difference observable. However, this can be considered as future works at this point.

The power analysis results can be see on Table 6. The Table 7 shows the area utilization of Lookup Tables, Flip Flops, Block RAMs, Input Output buffers and Global buffers and on Table 8 a utilization breakdown of different components is shown. In addition the Appendix 1 on page 52 includes a screenshot of the device slice utilization view.

	Slice LUTs (Logic)	Slice Registers	DPSs	BRAM	LUT as Memory
Controller	279	136	0	0	0
Neuron	51	16	0	0	0
Display	13	34	0	0	0
Max	12	9	0	0	0
Weight memory	13	4	0	3	0
Image memory	0	0	0	0.5	0
Results memory	0	0	0	0.5	0
Total	**381**	**199**	**0**	**4**	**0**

Table 8. FPGA utilization breakdown by components

5. Conclusion

In this work a Feedforward Artifical Neural Network was first built in software, that would be able to do image detection and classify subjects by inputting a predefined set of faces. Couple of methods were explored, including pruning and genetic algorithms, in order to optimize the neural network's topology. The purpose was to see, how does the decreased number of connections affect the performance of the network in hardware. In the next step, different aspects of the neural network implementation were analysed to make the hardware realization feasible: parallelization of the design, multiplication of weights, data representation and precision, routing and also approximation of activation function, that was also modelled in software in order to find a good solution. Finally, a design was chosen to be implemented on FPGA using VHDL hardware description language. The neural network is implemented in this case in a serial manner, without online learning and a image that has been preloaded to the memory.

As a result, the FPGA realization used 7% of the clock cycles that it would take the CPU to find the output. This can mean decreased power consumption or faster operation time. It was also tested how would reduced connections affect the performance, by skipping the redundant weights during calculation. Due to the serial design of the neural network, the circuitry itself did not change. Therefore, no changes in area or power consumption were observed. However, it was found that the number clock cycles can be reduced by 15% by trading off the network's accuracy.

The work serves also as a generic guide to implementing a synthesizable Feedforward Artificial Neural Network by going through all the essential steps required. The neural network configuration is described, including the components of hardware.

References

[1] K. Ray, *The Singularity Is Near: When Humans Transcend Biology*. 2005.

[2] "The orl database of faces." http://www.cl.cam.ac.uk/research/dtg/attarchive/facedatabase.html.

[3] P. Gang, I. Iimura, H. Tsurusawa, and S. Nakayama, "Genetic local search based on genetic recombination: A case for traveling salesman problem," in *Parallel and Distributed Computing: Applications and Technologies* (K.-M. Liew, H. Shen, S. See, W. Cai, P. Fan, and S. Horiguchi, eds.), vol. 3320 of *Lecture Notes in Computer Science*, pp. 202–212, Springer Berlin Heidelberg, 2005.

[4] "File:clb block diagram.png." https://en.wikibooks.org/wiki/File:CLB_Block_Diagram.pngs.

[5] "Artix-7 series fpgas overview." http://www.xilinx.com/support/documentation/data_sheets/ds180_7Series_Overview.pdf.

[6] S. K. Dwived and S. P. Azad, "Hardware implementation of face recognition using low precision representation.".

[7] M. B. I. Reaz, F. Assim, Awss; Choong, M. S. Husn, and F. Mohd-Yasin, "Prototyping of smart home: A multiagent approach," 2006.

[8] "Git repository of the thesis' source files." https://bitbucket.org/hkinks/thesis.

[9] M. Koit and T. Roosmaa, *Tehisintellekt*. Tartu Ülikool, Arvutiteaduse Instituut, 2011.

[10] V. B. Mountcastle, "Modality and topographic properties of a single neurons of cat's somatic sensory cortex"," *Journal of neurophysiology*, vol. 20, pp. 408–434, 1957.

[11] A. Angelucci, "Experimental retinal projections to the ferret auditory thalamus : morphology, development and effects on auditory cortical organization," 1997.

[12] F. Rosenblatt, "The perceptron – a perceiving and recognizing automaton," 1957.

[13] C. M. Bishop, *Pattern Recognition and Machine Learning*. Information science and statistics, Springer, 1st ed. 2006. corr. 2nd printing ed., 2006.

[14] "Artificial neural networks: Mathematics of backpropagation." http://briandolhansky.com/blog/artificial-neural-networks-linear-regression-part-1.

[15] "Artificial neural networks: Mathematics of backpropagation." http://briandolhansky.com/blog/2013/9/27/artificial-neural-networks-backpropagation-part-4.

[16] S. I. Gallant, *Neural network learning and expert systems*. MIT Press, 1993.

[17] R. Parekh, J. Yang, and V. Honavar, "Constructive neural-network learning algorithms for pattern classification," *IEEE Transactions on Neural Networks*, vol. 11, pp. 436–451, Mar 2000.

[18] S. Thrun, "Lifelong learning : a case study," Tech. Rep. CMU-CS-95-208, Carnegie-Mellon University.Computer science. Pittsburgh (PA US), 1995.

[19] Y. Li, K. Ang, G. Chong, W. Feng, K. Tan, and H. Kashiwagi, "Cautocsd-evolutionary search and optimisation enabled computer automated control system design," *International Journal of Automation and Computing*, vol. 1, no. 1, pp. 76–88, 2004.

[20] J. Anita and P. Vanathi, "Multiple fault diagnosis and test power reduction using genetic algorithms," in *Eco-friendly Computing and Communication Systems* (J. Mathew, P. Patra, D. Pradhan, and A. Kuttyamma, eds.), vol. 305 of *Communications in Computer and Information Science*, pp. 84–92, Springer Berlin Heidelberg, 2012.

[21] M. Srinivas and L. Patnaik, "Genetic algorithms: a survey," *Computer*, vol. 27, pp. 17–26, June 1994.

[22] "Genetic algorithms." http://www.doc.ic.ac.uk/~nd/surprise_96/journal/vol1/hmw/article1.html.

[23] A. Cosoroaba, "Achieve higher performance with virtex-5 fpgas," *Xcell journal*, 2006.

[24] J. C. R. Amos R. Omondi, *FPGA Implementations of Neural Networks*. Springer, 1 ed., 2006.

[25] S. L. Bade and B. L. Hutchings, "Fpga-based stochastic neural networks-implementation," in *FPGAs for Custom Computing Machines, 1994. Proceedings. IEEE Workshop on*, pp. 189–198, Apr 1994.

[26] D. Hammerstrom, "A vlsi architecture for high-performance, low-cost, on-chip learning," in *Neural Networks, 1990., 1990 IJCNN International Joint Conference on*, pp. 537–544 vol.2, June 1990.

[27] A. E. Gamal, "Two-dimensional stochastic model for interconnections in master slice integrated circuits," *IEEE Transactions on Circuits and Systems*, vol. 28, pp. 127–138, Feb 1981.

[28] S. Trimberger, "Effects of fpga architecture on fpga routing," in *Design Automation, 1995. DAC '95. 32nd Conference on*, pp. 574–578, 1995.

[29] K. R. Nichols, M. A. Moussa, and S. M. Areibi, "Feasibility of floating-point arithmetic in fpga based artificial neural networks," in *In CAINE*, pp. 8–13, 2002.

[30] J. L. Holi and J. N. Hwang, "Finite precision error analysis of neural network hardware implementations," *IEEE Transactions on Computers*, vol. 42, pp. 281–290, Mar 1993.

[31] J. L. Holt and T. E. Baker, "Back propagation simulations using limited precision calculations," in *Neural Networks, 1991., IJCNN-91-Seattle International Joint Conference on*, vol. ii, pp. 121–126 vol.2, Jul 1991.

[32] K. Basterretxea, J. M. Tarela, and I. del Campo, "Approximation of sigmoid function and the derivative for hardware implementation of artificial neurons," *IEE Proceedings - Circuits, Devices and Systems*, vol. 151, pp. 18–24, Feb 2004.

[33] D. J. Myers and R. A. Hutchinson, "Efficient implementation of piecewise linear activation function for digital vlsi neural networks," *Electronics Letters*, vol. 25, pp. 1662–1663, Nov 1989.

[34] C. Alippi and G. Storti-Gajani, "Simple approximation of sigmoidal functions: realistic design of digital neural networks capable of learning," in *Circuits and Systems, 1991., IEEE International Sympoisum on*, pp. 1505–1508 vol.3, Jun 1991.

[35] M. Zhang, S. Vassiliadis, and J. G. Delgado-Frias, "Sigmoid generators for neural computing using piecewise approximations," *IEEE Transactions on Computers*, vol. 45, pp. 1045–1049, Sep 1996.

[36] K. Basterretxea, J. M. Tarela, and I. del Campo, "Approximation of sigmoid function and the derivative for hardware implementation of artificial neurons," *IEE Proceedings - Circuits, Devices and Systems*, vol. 151, pp. 18–24, Feb 2004.

[37] A. H. Namin, K. Leboeuf, H. Wu, and M. Ahmadi, "Artificial neural networks activation function hdl coder," in *Electro/Information Technology, 2009. eit '09. IEEE International Conference on*, pp. 389–392, June 2009.

[38] M. T. Tommiska, "Efficient digital implementation of the sigmoid function for reprogrammable logic," *IEE Proceedings - Computers and Digital Techniques*, vol. 150, pp. 403–411, Nov 2003.

[39] "Machine learning | coursera." `https://www.coursera.org/learn/machine-learning`.

[40] L. Wang, W. Wu, J. Xiao, and Y. Yi, "Large scale artificial neural network training using multi-gpus," *CoRR*, vol. abs/1511.04348, 2015.

[41] S. Han, J. Pool, J. Tran, and W. J. Dally, "Learning both weights and connections for efficient neural networks," *CoRR*, vol. abs/1506.02626, 2015.

[42] "Hardware development with waterfall/v-model." `http://intland.com/blog/hardware-development-with-waterfallv-model/`.

[43] R. Gadea, J. Cerdá, F. Ballester, and A. Mocholí, "Artificial neural network implementation on a single fpga of a pipelined on-line backpropagation," in *Proceedings of the 13th International Symposium on System Synthesis*, ISSS '00, (Washington, DC, USA), pp. 225–230, IEEE Computer Society, 2000.

Appendix 1