THM | CAMPUS GIESSEN | MNI
TECHNISCHE HOCHSCHULE MITTELHESSEN
Mathematik, Naturwissenschaften und Informatik

A Thesis submitted for the degree of Master of Science

# SoC-based Image Classification using Binarized Neural Networks

Bastian Zeller

August 2018

Technische Hochschule Mittelhessen
University of Applied Sciences

Supervisor: Prof. Dr.-Ing. Hartmut Weber
Second Reader: Prof. Dr.-Ing. Martin Gräfe

# Contents

# List of Figures

# List of Tables

# Listings

### Abbreviations in Alphabetical Order

AI: Artificial Intelligence
ANN: Artificial Neural Network
ASIC: Application-specific Integrated Circuit
BNN: Binarized Neural Network
BOVW: Bag-of-Visual-Words
CNN: Convolutional Neural Network
CPU: Central Processing Unit
DBN: Deep Belief Network
DSP: Digital Signal Processing
FPGA: Field Programmable Gate Array
GPGPU: General-Purpose computing on Graphics Processing Units
GPU: Graphics Processing Unit
HPS: Hard Processor System
LUT: Lookup Table
LOC: Lines of Code
MNIST: Mixed National Institute of Standards and Technology
OpenCL: Open Computing Language
SGD: Stochastic Gradient Descent
RTE: Run-Time Environment
SIFT: Scale-invariant feature transform
SoC: System on Chip
TANH: Hyperbolic Tangent
TPU: Tensor Processing Unit
ReLU: Rectified Linear Unit

## Eidesstattliche Erklärung

Hiermit versichere ich, die vorliegende Arbeit selbstständig und unter ausschließlicher Verwendung der angegebenen Literatur und Hilfsmittel erstellt zu haben.

Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungsbehörde vorgelegt und auch nicht veröffentlicht.

Gießen, 26. August 2018

## Statutory Declaration

I declare that I have authored this thesis independently, that I have not used other than the declared sources / resources, and that I have explicitly marked all material which has been quoted either literally or by content from the used sources

The Master Thesis was not used in the same or in a similar version to achieve an academic grading or is being published elsewhere.

Gießen, 26. August 2018

# Acknowledgements

I would like to express my gratitude to my supervisor, Prof. Dr.-Ing Hartmut Weber at THM for the useful comments, remarks and engagement through the learning process of this master thesis. I would also like to acknowledge Prof. Dr.-Ing. Martin Gräfe at THM as the second reader of this thesis, and I am gratefully indebted to him for his very valuable comments on this thesis.

Finally, I must express my gratitude to my loved ones, who have supported me throughout entire process of writing this thesis, both by keeping me harmonious and helping me putting pieces together.

Bastian Zeller

# Abstract

Image classification with artificial neural networks is a field gaining much attention in the last years. Participants of classification competitions like ImageNet are exclusively relying on this technology, like the latest winners AlexNet or ResNet prove. While their classification capabilities even win against human capabilities, the implementations were growing to very deep networks, focusing mainly on classification accuracy and not on processing-speed or the limited availability of computational resources.

With artificial neural networks being integrated into modern life, like the usage in smart-assistants, smart-phones or self-driving cars the need to reduce the processing power to compute these networks is growing.

One technique to reduce the networks computational complexity is binarization of weights and activations as shown by Courbariaux et al. With weights and activations restricted to the values $-1$ and 1, the calculation when applying weights to neurons activations can be simplified by replacing floating-point multiplications with the binary XNOR operation.

This work refines the theory of binarized neural networks by providing a distributed implementation on an ARM CPU and a FPGA with the usage of the OpenCL framework. While the implementation on a FPGA can directly benefit from the binary logic of the new network type, the ARM core provides easy accessibility to connected peripherals providing the image sources.

New concepts for training binarized networks are developed within this thesis, leading to an enormous reduction of training time and therefore, a reduction of real-life production costs.

This implementation proves to execute faster than conventional methods on the same hardware, while using less resources and without trading classification accuracy.

**Keywords:** *Neural Network, Deep Learning, Embedded Systems, OpenCL, FPGA, SoC, Image Classification*

# 1 Introduction

This thesis deals with the research and implementation of binarized artificial neural networks used for image classification tasks.

The first chapter introduces the context in which neural networks are applied and describes recent challenges with their implementation and optimization.

A thesis on overcoming these challenges is proposed and finally, a roadmap of this thesis will be presented.

## 1.1 Context

In the last years the usage of neural networks is rising in a lot of fields. Especially image processing tasks are gaining lots of attention: Cars and other vehicles are becoming self-controlled with the help of computer vision and artificial intelligence. They are able to detect traffic signs or pedestrians [2] and take their own decisions on the roads. Manufacturing plants are becoming highly automated and smart phones can be unlocked by face recognition. Even smart personal assistants are based on neural networks that are recognizing and processing voice commands [3, 4].

Most of those tasks are shifting from conventional image recognition, meaning processing and analyzing the image through computational filters, to processing the images with artificial neural networks that can be trained to detect various image classes. The state-of-the-art approach on image classification using neural networks is making use of deep neural networks which are computationally expensive due to the usage of floating point calculations and require a large memory to store high precision weights.

This work refines the theory of image classification with binarized neural networks, a concept introduced recently. It allows to transfer computationally expensive neural networks using floating point calculations to networks almost exclusively using binary operations during forward propagation. This is achieved by binarizing all neurons weights as well as their activation functions. Courbariaux introduced the mathematical principles behind the research and explores a basic implementation [5].

The work shall show the possibility of implementing binarized neural networks on less expensive embedded devices with limited processing power and how they will increase the processing speed of image classification in comparison to conventional neural networks. It shall further investigate on how the usage of OpenCL with parallel hardware like FPGA fabric establishes additional potential, like reducing development complexity and time.

While neural networks are usually computed on general purpose CPUs, the trend is shifting to expensive specialized processors that offer functionalities to calculate those networks. In low cost platforms like smart phones these can be specialized ASICs with very low power consumption, while in more general and mostly scientific scenarios GPUs are used as accelerators for these tasks.

Also, in cloud computing there is a trend towards the usage of neural networks. The possibility to process huge amounts of data and have nearly unlimited resources available is shifting even conventional big data processing to self-learning tasks. Big data farms are hosting huge numbers of powerful GPUs to process those amounts of data in short time. The downside of those farms is their power consumption, due to the usage of general purpose CPU and GPU accelerators.

The usage of FPGAs, programmable logic devices that can adapt any computational architecture is pointing towards lower power consumption and the possibility to further specialize these farms to neural computing tasks. These FPGA devices are usually programmed in a hardware description language that offers specialized operations to achieve very fast computation times. Compared to conventional methods this increases the time to market, because the development time is much slower than writing these algorithms in high level programming languages like Python or Java. An uprising technology to overcome the high computational costs of neural networks in binarizing them on different levels, like shifting from high precision weights to binary values and limiting neurons activation functions to binary output.

With these technologies the amount of memory used for storing weights as well as the complex computation expenses can be drastically reduced. Due to the fact that only binary intermediate values are used in the computations, these algorithms can be synthesized very easily in FPGA logic.

With the latest development in manufacturing technologies, FPGA logic and general purpose processors can be built into the same chip enclosing and even make use of the same peripherals, like DDR memory or interfaces like USB or ethernet. This enables developers to easily outsource complex computing tasks into FPGA fabric while maintaining business logic still in general purpose processors like ARM cores.

Thus, high speed computational neural network processors can be built with very low costs to be used in cheap everyday products.

This thesis shall show that by binarizing neural networks weights and activations, processing time and memory bandwidth can be saved compared to traditional artificial neural networks. FPGA technology used for hardware acceleration is a promising choice. Thus, implementation of the binarized network in an FPGA system shall show an acceleration in performance compared to traditional approaches on CPUs and GPUs. Distributing the algorithm across an ARM core and FPGA by using the OpenCL framework shall show that by accelerating timing critical parts of the networks algorithm development time can drastically be decreased.

## 1.2 Motivation

While research on neural networks used for image classification is growing, the networks are growing too. Recent winners on image classification competitions were only achieved by very deep neural networks, implementing up to hundreds of layers. These networks got much deeper in the last years and their focus was mostly shifted into the direction of classification accuracy without respecting constraints like processing complexity and speed. Big implementations nowadays often are accelerated by multiple GPUs to bring their processing time to an acceptable time-frame. Devices that are implementing these techniques in future generations are becoming smaller and the trend points to even more integrated devices and processors, not capable of processing very deep networks in an acceptable time frame.

Personal assistants implemented in smart phones, production plants and self-driving cars are becoming the main application for neural networks. They are usually equipped with embedded processors and are sold in such big quantities that limiting the costs during production is becoming one of the manufacturers highest priorities. With cost savings in mind artificial neural networks are required that are less computational expensive and can easily be executed on low power embedded hardware.

A promising solution to lower processing complexity is the binarization of artificial neural networks proposed by Courbariaux et al [5], which allows the substitution of computational complex operations by simple bitwise operations. They showed in their work, that binarized neural networks can achieve state-of-the-art results like conventional deep neural networks do. Additionally, they prove in theory their methodology is showing enough potential to accelerate calculations much faster than conventional neural networks could. They also give first indications that binarized neural networks can be implemented on embedded hardware and expedite the image classification process in real-world scenarios by implementing their work experimentally on GPU devices using high level frameworks.

The usage of mainly binary operations allows the networks to be executed and used in real-time image classification tasks on small and less powerful embedded devices or even programmable hardware like FPGA devices, which could map these bitwise operations directly without computational overhead.

This thesis explores the ability to execute these binarized neural networks on FPGA hardware and make effective usage of the binary logic operations introduced by binarized networks without the need of developing highly optimized hardware description code but instead abstracting these functionalities by using the OpenCL framework.

## 1.3 Work Assignment

This master thesis is related on machine learning, especially image classification by neural networks. Typically, deep neural networks (DNNs) are used for for these tasks. DNNs are computationally expensive and require large amounts of memory to store high precision weights.

A new approach for working on image classification tasks with neural networks are Binarized Neural Networks (BNNs) [5, 6, 7]. BNNs only use binary weights and activation which need much less memory than high precision weights for DNNs. So BNNs are suited for an implementation on FPGA hardware.

In this thesis, the classification of handwritten digits or characters from the EMNIST dataset [8] provided by NIST, the National Institute of Standards and Technolgy of US Department of Commerce, has to be used for the evaluation of BNN usage in an SoC device equipped with one Cortex A9 ARM core and an Altera Cyclone IV FPGA. A second task is to compare the utilization of FPGA logic and memory cells for this implementation with other comparable FPGA implementations for image classification [9, 10, 11]

## 1.4 Thesis Structure

In this master thesis a method to run binarized neural networks on SoC platform consisting ARM processor cores and FPGA logic will be evaluated. The thesis is mainly divided into three parts: Theoretical background, methodic and implementation details and the evaluation and analysis of the achieved results.

Collecting theoretical background information will help to understand the fundamentals of state-of-the-art technologies available at this time and will assist to select the best technology suited for the task at hand.

- **Chapter 2** analyzes the state-of-art in image processing and the usage of deep neural networks.
- **Chapter 3** introduces deep learning methods and discusses the advancement of the usage of artificial neural networks for image classification tasks. Additionally, key performance indicators are defined that will enable further discussion on the topic.
- In **chapter 4** the system architecture is presented
- **Chapter 5** discusses the implementation of the binarized neural network as well as its representation in the SoC
- In **chapter 6** performance measurements are presented and analyzed according to the previously defined Key Performance Indicators
- **Chapter 7** summarizes the key aspects of this work and gives an overview on possible future work building up on this thesis.

# 2 State of Art Review

The last years have been a renaissance to deep learning. With processing capabilities evolving, the concept developed in the 20th century is gaining more and more interest in the industry. It is progressing into consumer devices like smart phones or tablets [4] and became the foundation of home assistance systems [3].

A field nowadays highly dominated by deep learning is image classification, the basement for computer vision. The technique is reaching highest scores in classifying competitions like MNIST [12] or ImageNet [13].

These competitions were formerly approached with conventional image analysis algorithms like Scale-Invariant Feature Transformation (SIFT) [14] or Bag-of-Visual-words (BOVW) [15, p.101 ff.]. These conventional image processing algorithms were providing mitigated results and were showing error rates only down to 26.2 percent.

Advances in the ImageNet Large Scale Visual Recognition Challenge were achieved starting in 2012. Authors proposed several architectures based on neural networks that were setting the groundwork on recent image classification research. As visualized in figure 2.1, the error rates have been reduced to outperform even human capabilities:

**AlexNet** was the first big achievement in image classification. The paper describing the algorithm is one of the most influential papers concerning image classification. It reduced the error rate commonly achieved at this time by 50 percent to an error rate of 15.4 percent. In 2012 the authors of [16] were the first to use convolutional neural networks on image classification tasks. This approach was ground-breaking and brought the concepts of deep learning into public focus. It was a very basic implementation of 8 layers, using mainly 11x11 filters. Because of the huge computational complexity, AlexNet was run in parallel on two GPUs.

While AlexNet focused on implementing just a few layers that were equipped with rather complex filters, **VGG Net** was implemented by using only 3x3 filters [17]. The authors argue that combining two 3x3 filters simulates the effect of a larger 5x5 filter, by still providing the computational benefits of a small filter size. They implemented a total of 19 layers which turned out to perform well not only on image classification, but also on localization tasks [17, p. 10]. The model was trained on

four Nvidia Titan Black GPUs over the period of three weeks and could achieve an error rate of 7.3 percent.

The next advancement on classification tasks was provided by **GoogleNet** in 2014, a project of several Google research groups focusing on self-driving cars, image research and understanding content provided in YouTube videos [18]. It proposed a 22 layer convolutional network which achieved an error rate of 6.7 percent in the ImageNet competition. The approach is different from all other approaches to this date, because it does not focus on stacking layers on top of each other in a sequential path, but is introducing parallel structures of 1x1, 3x3 and 5x5 filters. The authors state the performance as "trained on a few high-end GPUs within a week".

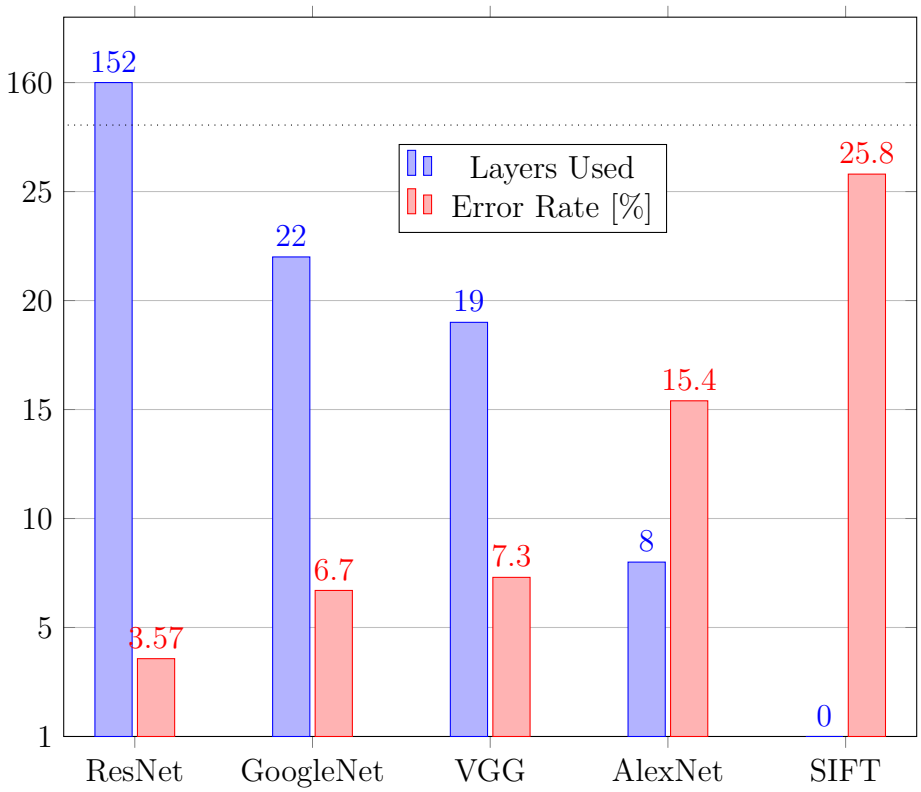



Figure 2.1: Advances in the ImageNet competition. While the error rate is is reduced with each iteration, networks are growing very big.

In 2015 the Microsoft Research Asia group proposed the architecture of **ResNet**, a 152-layer network leading to an error rate of 3.6 percent, which even outperforms human capabilities [19]. The main idea behind ResNet is the "identity shortcut connection", a shortcut path within the network which enables the use of a massive amount of layers, without losing control over it. It was trained on eight GPUs within three weeks.

A side effect on the popularity deep neural networks gained since the proposal of AlexNet in 2012 is the possibility to use them for different tasks besides image classification, by using the proposed networks in new ways: Instead of classifying images, researchers could produce new images by mixing attitudes of different images. An example is the transfer of one artworks style to another image which enables the production of new artworks [20].

All the previously discussed approaches are following the trend of implementing (very) deep convolutional networks. While this leads to higher classification accuracy, the networks are not the most efficient in terms of size and computational speed. They usually require high-end hardware and very long time to train and execute the network.

In real-life applications like the usage in self-driving cars, robotics or in assembly lines, the tasks executed by those networks are supposed to be processed in a very limited time frame and run on limited hardware. With the appearance of deep learning on low power computing devices like smart-phones, home assistants and other embedded devices, the demand to reduce the computational complexity of applications implementing deep learning algorithms is even growing faster. This motivated researchers to investigate on approaches to minimize the computational cost of those networks and find possibilities to reduce the need for expensive hardware based on multiple high-end GPUs.

While one approach on devices with access to the internet is to outsource complex calculations to cloud service providers [21], research is focusing on approaches to reduce the complexity the calculations of neural networks have by researching on techniques that lower the needs of processing power and memory consumption and thus, are speeding up the execution time without sacrificing too much accuracy:

**Network pruning** is a technique of identifying and removing parts inside neural networks that have the least important influence on the outcome [22]. Methods on identifying those parts can be based on the magnitude of weights and activations or by identifying mutual information within different paths of the network, like feature maps that filter similar criteria. The theory of pruning is modeled after the human brains capability to maintain functionality even after suffering damages within big parts of the brain [23, 24].

**MobileNets**, as proposed by a Google research group, combine recent techniques that allow big convolutional networks being computed on small embedded devices like mobile phones or tablets [25]. Depthwise separable convolutions, previously discovered in the Xception network [26], allow the thinning of convolutional filters. In a 2D environment with multiple channels, e.g. an image with RGB color, a filter is as deep as the number of channels used on the input. In depthwise convolution each channel is presented to a separate filter. This technique is combined with a

1x1 convolution across channels, which allows computation with less parameters as regular convolutional layers have and thus, is leading to less operations to compute. This makes the network cheaper and faster. The technique of depthwise separable convolutions is combined with the two 'hyper parameters' called width- and resolution multiplier. The width multiplier is scaling the amount of inputs and outputs for each layer, while the resolution multiplier is a parameter scaling the size of the input image and the internal representation of every layer, allowing to reduce the size and latency of the network.

The **ShuffleNet** convolutional network "is designed specially for mobile devices with very limited computing power" [27]. By introducing two new operations, pointwise group convolution and channel shuffle, the computational cost of a network can be reduced drastically while trading off only a reasonable amount of the networks accuracy. Group convolution is a combination of techniques used in AlexNet and the previously discussed depthwise separable convolution by generalizing them to work on groups of input channels. The channel shuffle operation is extracting information from feature maps and shuffles them among channel groups, leading to increased accuracy and lowered computational speed.

While these implementations focus on implementing neural networks on mobile processor architectures like the ARM architecture, other researchers focus on limiting and optimizing the complexity of the underlying calculations to allow mapping the networks algorithms on programmable logic like FPGA and ASIC technology.
**Binarized neural networks** evolved from neural networks with low precision weights [28] and reduce the networks complexity by removing the need for high-precision and computational expensive floating-point calculations when processing the networks weights and activations [5]. Weights and activations are represented as binary values, allowing to replace formerly expensive multiplications and trigonometric functions being replaced by simple bitwise operations like XNOR or adders.

The usage of FPGA hardware is a reasonable tradeoff between the computational power of GPUs and the flexibility and efficiency of mobile CPU architectures like ARM and can build the basis for specialized machine learning chips. The hardware design synthesized for the FPGA can be converted to an ASIC design efficiently without the need for redesigning the architecture. These custom designs have the advantage of draastically reducing energy consumption.

The trend of using FPGA or custom ASIC chips for neural network computation has already been applied to big scale data center operations and operators that were previously relying exclusively on CPU-GPU architectures [29] are moving towards FPGA or ASIC architectures: Google is developing custom ASIC accelerators for machine learning tasks using neural networks, called Tensor Processing Units (TPU), which they deploy to data centers [30]. Microsoft and Amazon follow different approaches in their data centers by deploying FPGA based accelerators or SoCs consisting of

CPU and FPGA cores to their data centers [31, 32], which are the basis to their cloud services catapult [33] and AWS F1 [32].

# 3 Theoretical Background

This chapter will introduce the fundamental theory on which later chapters are based. It introduces the theory of neural networks and discusses parallel computing platforms that are suited for neural network computing and image classification. To be able to measure the efficiency of the model developed in this work and compare it to different models, performance measuring units will be defined and discussed.

## 3.1 Artificial Neural Networks

Artificial neural networks are algorithms loosely modeled after the human brains neuronal structure. They are usually composed out of multiple layers of artificial neurons, a concept this section will introduce. The concept of artificial neurons evolved from the perceptron, a concept introduced in the second half of the twentieth century.

### 3.1.1 Perceptron

The concept of perceptron was developed in the early 1960's [34, Ch.2, p. 12]. It is a simplified representation of a biological neuron. While modern neural network use more complex models of neurons, the perceptron is introducing the fundamental concepts of nowadays neurons. The purpose of a perceptron is making decisions on problems by weighting evidence.

A perceptron takes several binary inputs $x0, x1, x2, ...$ and produces a single binary output $o_j$, as illustrated in figure 3.1. The inputs are weighted and a sum over all weighted inputs is calculated. This sum is fed into a function called activation function, which produces a binary output [35, pp. 55-75].

The activation function is a step function 3.2 with the zero-point shifted up or down by a threshold. It produces an output of 1 for all inputs $x_n \geq \theta_j$ and 0 for all inputs $x_n < \theta_j$. The threshold can also be described as an input parameter of the perceptron

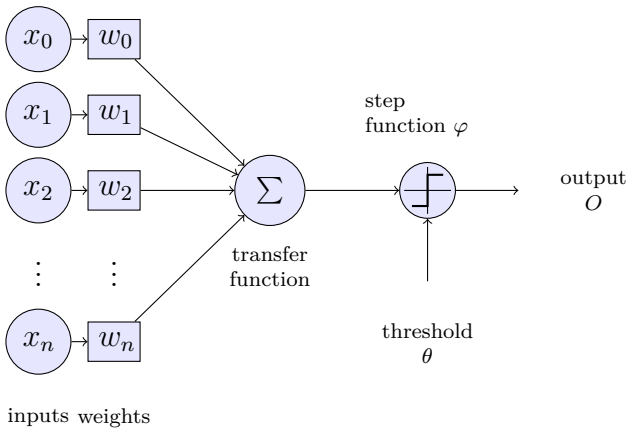



Figure 3.1: A schematic representation of a perceptron.

and moved to the other side of the equation. This input parameter is called bias. $b_j \equiv -\theta$. Thus, the activation function can be described with equation 3.1.

$$f(x) = \begin{cases} 0 & \text{if } x + b_j \geq 0 \\ 1 & \text{if } x + b_j < 0 \end{cases} \tag{3.1}$$

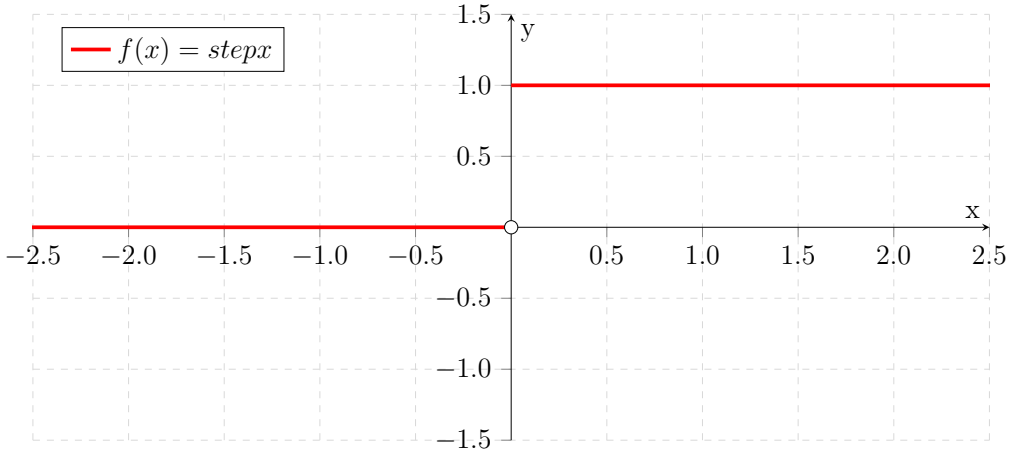



Figure 3.2: Plot of the step function.

Varying the weights will result in different models of the perceptron and so in different paths of the decision making. Inputs and weights are both vectors of equal length. Thereby the sum of all weighted inputs can be described as the dot product of the input vector $x$ and the weights vector $w$. Equation 3.2 is showing the relation between

the weighted sum and dot product.

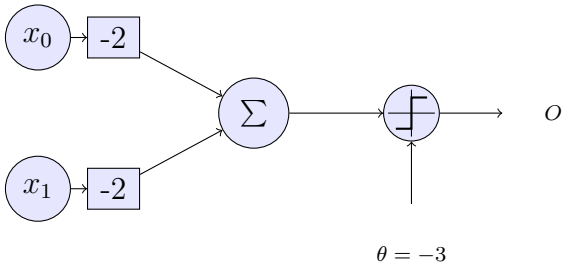
$$w \cdot x \equiv \sum_j w_j * x_j \tag{3.2}$$

The perceptron can be described by the equation 3.3

$$O = \begin{cases} 0 & \text{if } w \cdot x + b \geq 0 \\ 1 & \text{if } w \cdot x + b < 0 \end{cases} \tag{3.3}$$

## Logic Functions

With the correct parameters selected for weights and threshold, the concept of the perceptron can be used to implement logic functions like OR, AND or NAND (Figure 3.3). A NAND function has universal completeness, meaning all boolean function can be converted to a combination of NAND gates. With enough NAND gates available, every combinatorial logic function can be realized [35, p. 62].




| **x0** | **x1** | $O$ |
|---|---|---|
| 0 | 0 | 1 |
| 1 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

Figure 3.3: Parameterization of a perceptron to implement the NAND logic function. Following [36].

This fact makes it possible to build any combinatorial logic function out of perceptrons. Since those combinatorial logic functions are the fundamentals of computer science, all concepts of computer science can be represented through the usage of perceptrons, which makes the concept of perceptrons universal for computation. A network of perceptrons can represent any other computing device like an adder, calculator or a modern CPU.

Figure 3.4 is showing an implementation of a half adder using only NAND logic gates. By building a network out of perceptrons parameterized to fulfill the NAND logic function, a half adder can also be composed from perceptrons (see Figure 3.5)

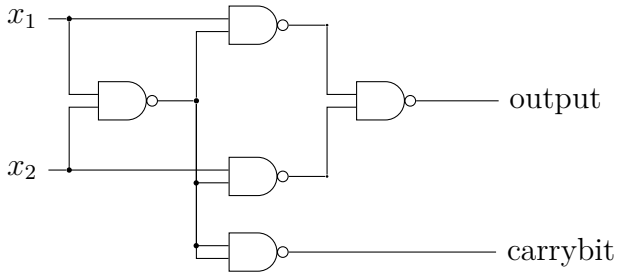



Figure 3.4: Implementation of a full adder using only NAND logic gates. Adapting [36].

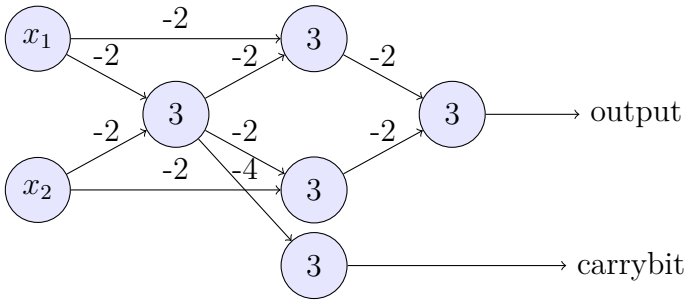



Figure 3.5: Implementation of a full adder with perceptrons. For simplification, the bias is written into the perceptron and the weights are above the connections. Adapting [36].

The network of perceptrons comes with the advantage of being able to adapt to any other logic function by simply changing its weights and biases. Thus, a network of perceptrons can be parameterized to response to any external stimulus given.

## Learning Perceptrons

With the perceptron being adaptable to any logic function, the use of linear algebra enables us to determine the parameters needed to fulfill any given logic function [35, pp. 78-84]

By converting the function describing the perceptron to have a fixed input and be variable on the weights and bias, an input vector and desired output can be applied to the equation and weights and bias can be calculated. The perceptron can learn a new function by adjusting its weights and bias in situations where a complex network of conventional logic functions might be suited.

The biggest drawback in learning with a perceptron is the binary output of the step function. Small changes in the weights and bias will not result in any changes of

the output. Only reaching the threshold will result in changes of the output. Small changes will either result in no change at all, or they will result in a full change of the perceptron's behavior. In a network of perceptrons small changes on weights might cause the network to work in an unpredictable way.

### 3.1.2 Artificial Neurons

Modern neural networks use a modified version of a perceptron which shares the same fundamental concepts, but because of modifications to the algorithm it can be adapted, so that small changes in weights and bias will result in changes to the output value.

The threshold of a perceptron is converted into a bias with $b_j \equiv -\theta$. The bias can be implemented as one additional input $x_{n+1} = b$ and an attached weight with value $w_{n+1} = 1$, so all calculations in form $w \cdot x + b$ can be simplified to $w \cdot x$. The neuron is also modified to accept non-binary input data.

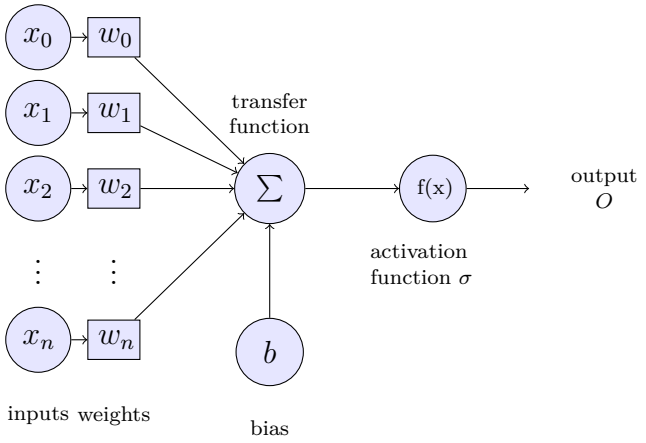



Figure 3.6: A schematic representation of an artificial neuron

This changes the threshold from being an input variable of the step function to a normal weighted input of the neuron. With this modification, the activation function can be replaced by any other function in the form $\sigma = f(x)$. With any function working as activation function, small changes in weights $\Delta w$ can result in changes in the output $\Delta O$.

An activation function that is commonly used within artificial neurons is the sigmoid function which is defined by equation 3.4. With the sigmoid function used as

activation function, the equation for the output of an artificial neuron is defined by equation 3.5

$$\sigma(x) = \frac{1}{1 + e^{-x}} \tag{3.4}$$

$$O = \frac{1}{1 + \exp\left(-(\sum_j (x_j * w_j) + b)\right)} \tag{3.5}$$

The sigmoid function is behaving similar to the step function, so the behavior of the neuron is equal to a perceptron. The higher the exponent value in the sigmoid function gets, the more it is approximating a step function, as shown in figure 3.7 the higher $x \cdot w + b$ gets, the more a neuron is converging to a perceptron.

Small changes in $\Delta w_j$ shall result in small changes in $\Delta O$. Using calculus $\Delta O$ can be determined by equation 3.6. This shows that $\Delta O$ is a linear function of $\Delta w$ and $\Delta b$. The learning process of an artificial neuron is thus aiming towards determining $\Delta w$ by measuring $\Delta O$.

$$\Delta O = \sum_j (\frac{\partial O}{\partial w_j} \Delta w_j) + \frac{\partial O}{\partial b} \Delta b \tag{3.6}$$

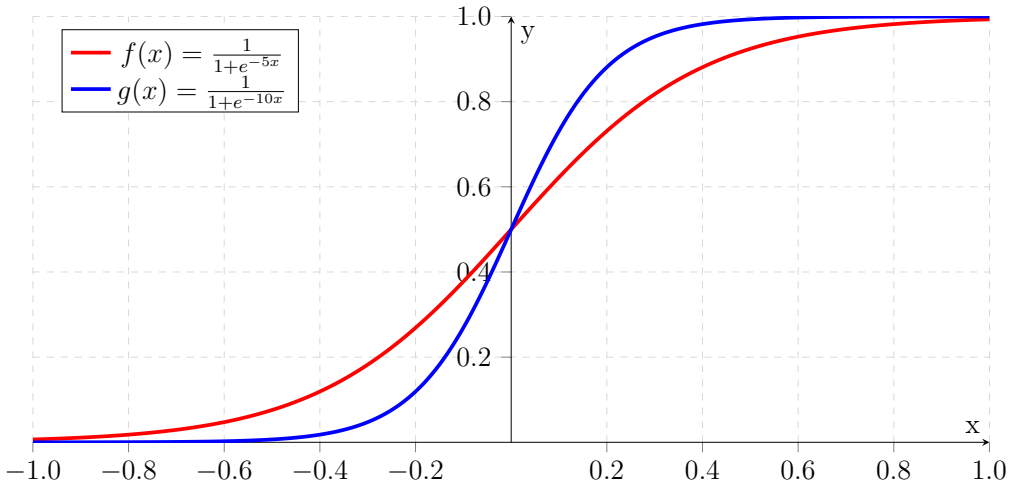



Figure 3.7: Plot of sigmoid function

### 3.1.3 Networks of Neurons

Analogous to the concept of connecting perceptrons illustrated in figure 3.5, neurons can be connected to build a network. The neurons can be connected in different combinations. The following paragraphs will introduce the most common network architectures in recent research.

#### Feed Forward Network

A common combination is to organize the neurons in layers, where all outputs of one layer are connected to all inputs of the next layer, as depicted in figure 3.8 . This type of network is called a *feed forward neural network*. The first layer is containing the input values of the problem that will be solved, the last layer represents the output. They are called *input layer* and *output layer*. The layers between input- and output layer are called *hidden layers* [35, pp. 125-148].

The organization of input- and output layer is directly mapping to the problems input and output variables. The design of the hidden layers can be more complex. A layer $A$ consisting of $n$ neurons connected to a layer $B$ consisting of $m$ neurons is resulting in a total of $n * m$ connections between those layers. Each neuron in layer $B$ is connected to $n$ neurons from layer $A$, resulting in $n$ inputs and thus $n$ weights connecting to this neuron.

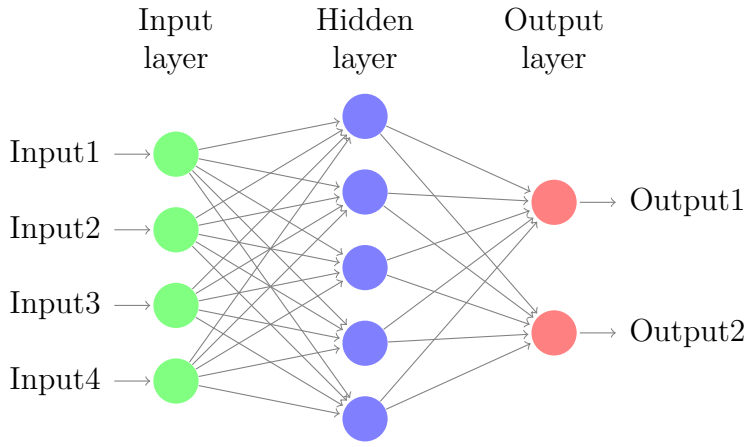



Figure 3.8: Graph of a feed forward neural network with a single hidden layer.

## Recurrent Networks

While in feed forward networks the connections from one neuron only connect to neurons of the next layer, recurrent networks also implement feedback loops, where the output of one neuron can be fed back as input of another neuron of a previous layer, or even connect to its own input, as illustrated in figure 3.9.

Their structure of applying feedback loops makes them useful on long-short-term-memory based models, like acoustic analysis [37] However, in practice recurrent network researchers face problems in applying learning algorithms like the back-propagation algorithm to them, because the feedback loops can easily lead to exploding or vanishing gradients [38]. Thus, their behavior is very complex and can become unpredictable.

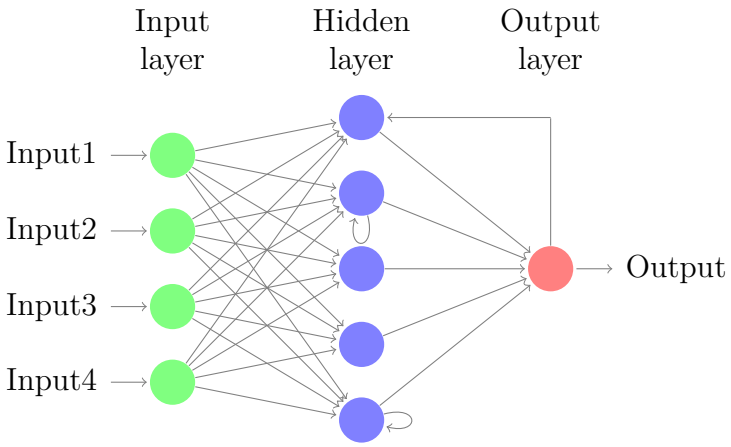



Figure 3.9: Graph of a recurrent neural network.

## Hidden Layer Organization

The number of hidden layers and the amount of neurons in those layers is dependent on the available data, meaning the number of input and output neurons, the amount of training data available, the complexity of the underlying problem and the algorithm used for training the network [35, pp. 125-148].

### Number of Hidden Layers

Linear separable problems can be solved without the need for any hidden layers, but more complex problems require at least one hidden layer to be solved. As the universal approximation theorem states, a feed forward network using only a single

hidden layer with enough neurons can approximate to any function given [39]. While this allows to theoretically solve any given problem with just one hidden layer, the theorem does not state the possibility of training the network. If it empirically proves to enhance the performance for the given problem, the usage of 2 or more hidden layers can be useful. A network with multiple hidden layers is illustrated in figure 3.10.

When multiple hidden layers are present in a neural network, a layer can be understood as pre-filtering the data fed to the next layer, where groups of neurons of one layer are filtering specific characteristics of the dataset. The next layers will make decisions based on these characteristics, not the underlying data.

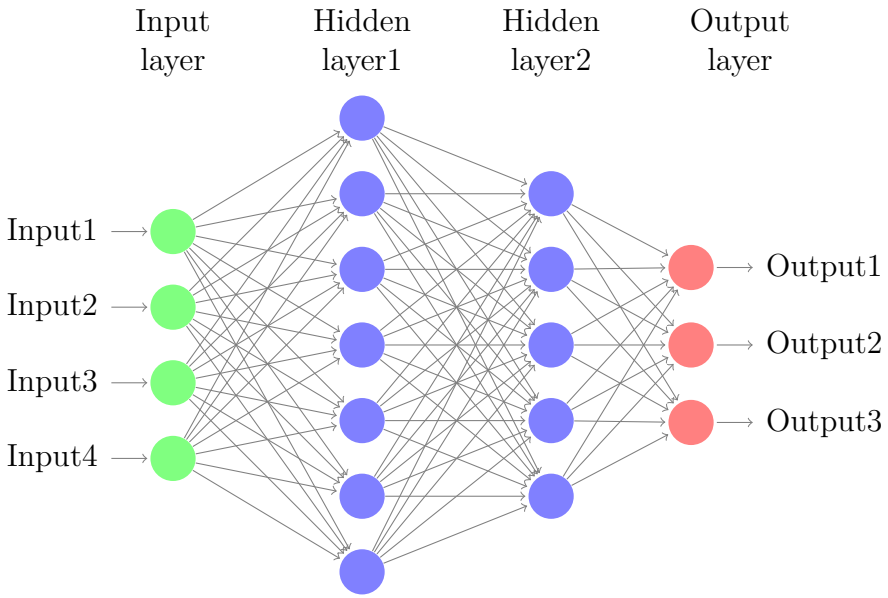



Figure 3.10: Graph of a feed forward network with multiple hidden layers.

### Number of Neurons in the Hidden Layers

Involving too few neurons in the hidden layers will cause the network to not correctly approximate the problems solution, leading to an underfitting network. In contrast involving too many neurons will cause the network to overfit by solving problems that were not required to solve and thus can cause the network to not adapt to data not present in the training sets. The relation between under- and overfitting networks is depicted in figure 3.11.

While some rules of thumb exist that set the number of neurons around the range of $\sqrt{n_{in} * n_{out}}$, where $n_{in}$ and $n_{out}$ describe the number of input- and output units

connected to the layer [40], the exact number cannot be estimated mathematically and must be solved empirically by observing the networks error during training and validation [41].

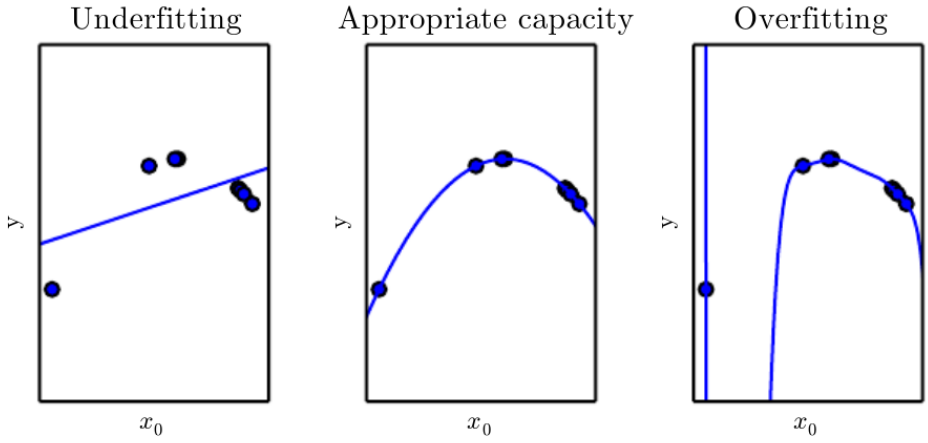



Figure 3.11: 3 models of a neural network solving the problem of mapping points from a quadratic function. **Left**: A linear model is leading to underfitting by not accurately capturing the given data samples. **Right**: A polynomial model can suffer from overfitting by passing all points, but not leading to the optimal structure. **Center**: A quadratic model is fitting the data well, even to unseen points. Image and description reused from [42, Ch. 5, p. 111].

## Convolutional Neuronal Networks

Convolutional neural networks were introduced in the 20th century [43, 44] and were widely adapted for classification tasks [45] They got public attention after the proposal of AlexNet, a convolutional network used for image classification [16]. Convolutional neural networks are a specialized version of conventional neural networks, based on the assumption that the input data is made of images.

While ordinary neural networks, like feed forward networks work well on small images, they do not scale well for bigger image size with multiple colors, as their number of weights from input layer to the first hidden layer will grow very large, which would increase the overall network size as well, which eventually will lead to become unpracticable.

Convolutional neural networks arrange their neurons in a 3-dimensional structure, based on the images width, height and depth, where the layers are not fully connected: A layer only connects to a small part of its previous layers neurons as illustrated in

figure 3.12. This area is slided over the full surface of the image, a mathematical operation called convolution.

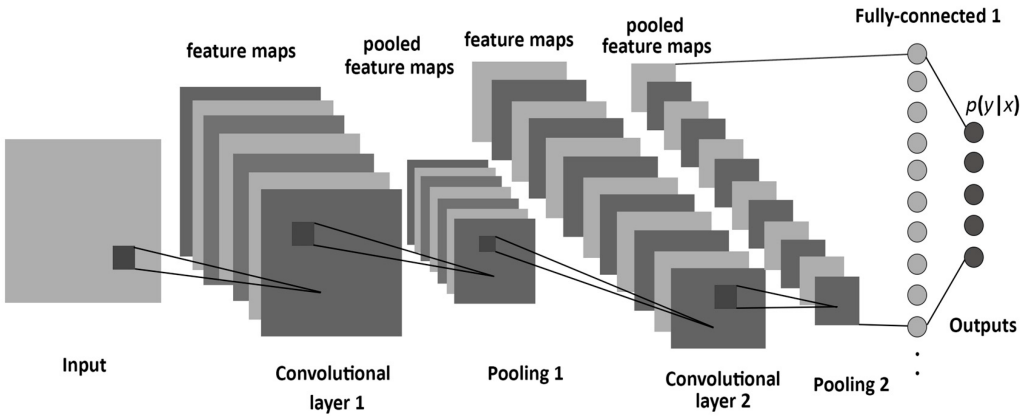



Figure 3.12: Architecture of the LeNet-5 [45], a convolutional neural network. Image reused from [46].

The layers in a convolutional network can vary in their function and parameters to control them: While convolutional layers slide over all neurons of the previous layer, pooling layers combine the output of multiple neurons into a single output neuron.

### 3.1.4 Learning as a Gradient Descent

As shown in previous sections, neural networks are capable of approximating a wide range of functions by implementing hidden layers with an adequate amount of neurons parametrized by their weights and biases. This section will discuss a method that enables the network to learn those parameters by feeding inputs and the desired outputs into the network.

To quantify the correctness of the networks approximation, the cost function defined by equation 3.7 is used. This equation is called the *mean squared error* or *quadratic cost function* $C$. It is a function dependent on the weights and biases of the network. The difference between the desired output $a$ and the networks output $y$ is built for all inputs $x$. Their number is determined by $n$.

$$C(w,b) = \frac{1}{2n} \sum_{x} |y(x) - a|^2 \tag{3.7}$$

The goal of a learning the parameter to approximate any function correctly is to change the weights and biases in a way that the cost functions output is as small as possible. The closer $C(w,b)$ gets, the better the function will be approximated.

This can be achieved by a technique called *stochastic gradient descent* or SGD, an iterative method to solve minimization problems [47].

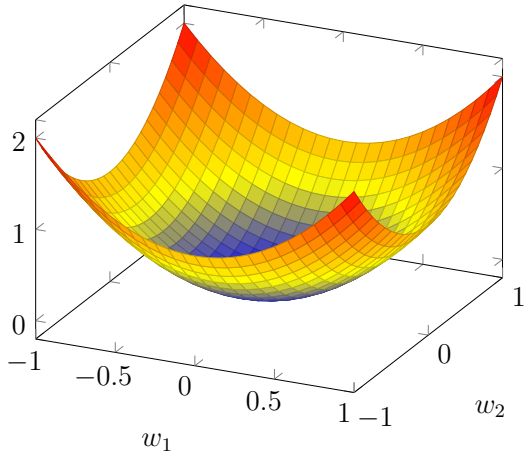



Figure 3.13: The surface plane of the Mean Squared Error with 2 inputs

With stochastic gradient descent small changes $\Delta w$ and $\Delta b$ are incrementally applied to the cost functions parameters $w$ and $b$, until its output is minimized to satisfaction. Figure 3.13 illustrates the surface pane of the cost function for 2 inputs. The SGD method can be imagined as rolling a ball on the pane until it reaches the bottom. The lowest point is called the global minimum of the cost function and is considered the optimal solution to the minimization problem.

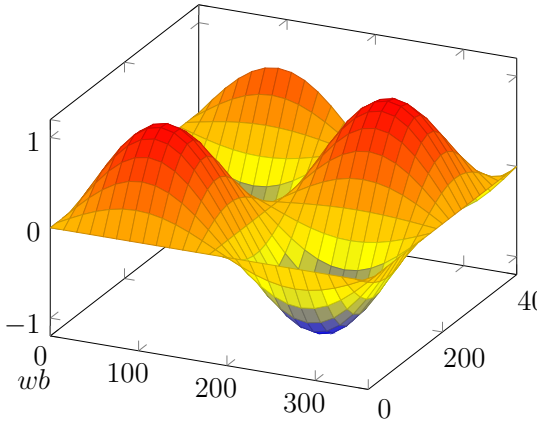



Figure 3.14: A surface plan of a network showing local and global minima.

While 3.13 depicts a function dependent on just 2 input parameters and the global minimum is easily comprehensible, the cost for complex problems is dependent on thousands of parameters, resulting in a more complex pane as illustrated in 3.14. This pane can have multiple local minima, which could lead to incorrect assumptions of

the best solution [35, pp. 101-119]. Avoiding local minima is a common problem during the training of neural networks.

A common technique to overcome this problem is to change the neurons activation function. While the sigmoid function will map negative inputs to outputs close to zero, the hyperbolic tangent, as illustrated in figure 3.15, will map negative inputs to negative outputs, which leads to a wider range of output values. As figure 3.16 illustrates the hyperbolic tangent produces a stronger gradient, which will lead to bigger $\Delta w$ that are leading the SGD less likely to remain in a local minimum [48].

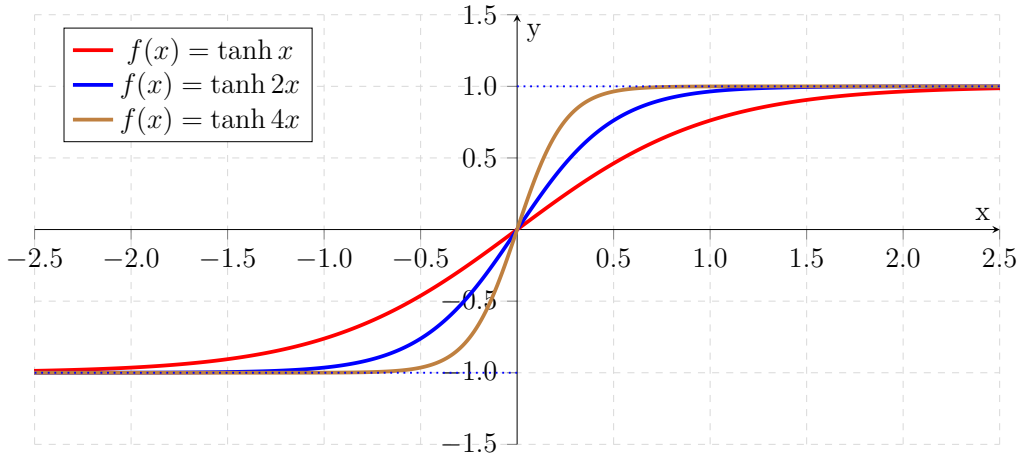



Figure 3.15: Plot of the hyperbolic tangent.

The cost $C$ can be minimized by the iterative process of the gradient descent for which the gradient of $w$ is calculated as shown in equation 3.8. Each weight is updated by $\Delta w$, defined in equation 3.9, where $\eta$ is a constant for the learning rate, a parameter that scales the learning process [35, pp. 101-119].

$$\Delta C = \frac{\partial C}{\partial w_1} + \frac{\partial C}{\partial w_2} + ... \frac{\partial C}{\partial w_n} \tag{3.8}$$

$$\Delta w = -\eta \frac{\partial C}{\partial w_i} \text{ for } i = 1, 2, 3, ..., n \tag{3.9}$$

As shown in equation 3.9 $\Delta w$ is depending on $\Delta C$, while $C$ is depending on the activation functions output $a$. To determine $\Delta w$ the activation function must be differentiable. Equation 3.10 is defining derivative of the sigmoid activation, 3.11 the derivative of the hyperbolic tangent.

$$s'(x) = \frac{\partial}{\partial x} s(x) = s(x)(1 - s(x)) \tag{3.10}$$

$$tanh'(x) = \frac{\partial}{\partial x} tanh(x) = 1 - tanh(x)^2 \tag{3.11}$$

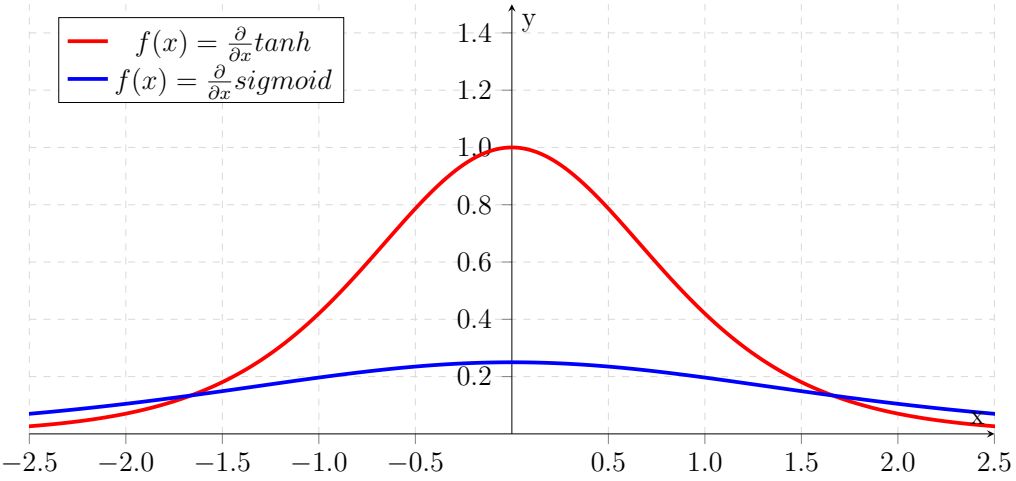



Figure 3.16: Partial derivative of sigmoid and hyperbolic tangent.

### 3.1.5 Backpropagation Algorithm

The method of stochastic gradient descent, described in 3.1.4, is applying small changes to the networks weights in order to approximate a desired output. The exact amount of change to each weight is defined by the partial derivative of the networks cost function with respect to the weight examined. It is called the *gradients* of the weights vector.

The backpropagation algorithm, developed in 1988, is a method to determine these partial derivatives by calculating the error for each output neuron and propagating it backwards through the network [49].

The network can be understood as a chain of function compositions, where each node of the network is given a composite structure: The right side of the node computes the neurons activation function, while the left side computes the activation function derivative for the same input, as depicted in figure 3.17. The result computed at the right side is considered the output of the neuron and sent to the neurons connected to it, while the result computed at the left side is stored in the neuron itself [35, pp. 152-182]. During a second step, the backpropagation, where the network is propagated backwards, the previously stored results are used as the neurons outputs.

A simplification to a neuron during backpropagation can be made: A neuron is a composition of an integrator +, summarizing the weighted inputs and an activation

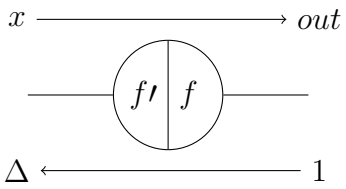



Figure 3.17: Function derivatives during network propagation. Image following [35, p. 158].

function $s$, as shown in figure 3.18. When building the derivative of both nodes, the activation functions derivative get $s'$, while the integrators derivative is 1 for all inputs, allowing us, during backpropagation, to reduce the neurons calculation to only its activation function.

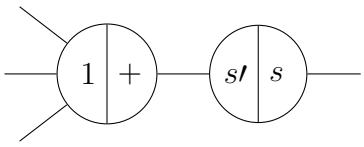



Figure 3.18: Function derivatives during network propagation. Image following [35, p. 158].

To apply backwards propagation through the composited network, three cases have to be considered: Function composition, addition and inputs weighting must be proven to be generalizable in order to be able to formulate the algorithm for backpropagation. They will be discussed in the following paragraph.

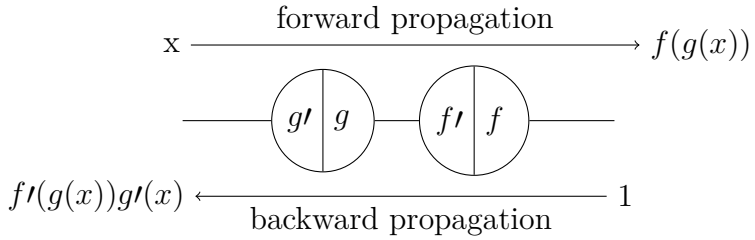



Figure 3.19: Function composition during forward and backward propagation. Image following [35, pp. 159-160]

The **composition** of the functions $f$ and $g$ is visualized in figure 3.19. During forward propagation the input $x$ is fed to the left side of the network. The nodes output and its derivative are computed. The derivative is stored in the node, while the output is fed to the next node. The networks output is the composition $f(g(x))$.

Running the network backwards by feeding 1 to the right side of the network, in each node the input is multiplied by the value stored in the unit. Thus, the derivative of the function composition $f(g(x))$ is $f'(g(x))g'(x)$. Since the backpropagation implements the chain rule this result can be generalized to any sequence of function compositions [35, pp. 159-160].

Figure 3.20 depicts the **addition** of the functions $f$ and $g$. As already shown for the simplification of the neurons integrator, the node implementing the addition has a partial derivative of 1 for all inputs. When forward propagating the network, the network computes $f(x) + g(x)$ and the derivatives $f'(x)$ and $g'(x)$ are stored inside the nodes. During the backpropagation step a 1 is fed to the right side of the network, multiplied with the partial derivatives stored in the nodes, the result of the step is $f'(x) + g'(x)$. As shown in [35, pp. 159-160], the result can be generalized to any addition of functions.

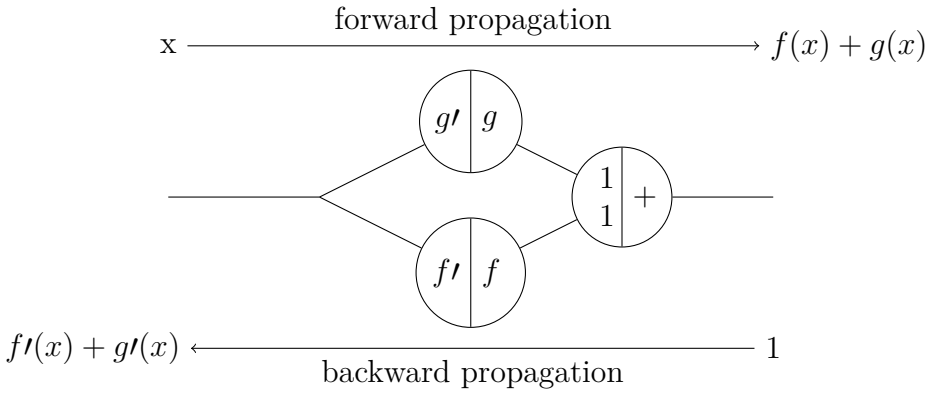



Figure 3.20: Function addition during forward and backward propagation. Image following [35, pp. 159-160]

**Weighted inputs** can be understood as multiplying each sides input with $w$: While during forward propagation the nodes input $x$ is multiplied with the weight $w$, resulting in $wx$, during backward propagation the weight is multiplied by 1 fed from the right side of the node, resulting in $w$. This is the derivative of $wx$ with respect to $x$. Thus, it can be concluded that weights are modulating the nodes information from both sides, by multiplying each sides input with $w$ [35, p. 161].

### Steps of the Backpropagation Algorithm

The backpropagation method is defined by algorithm 1.

**Algorithm 1** Backward propagation.
$N$ is the amount of training data, $L$ the amount of layers and $t$ the training vector. $a$ is the vector of activations, $w$ the vector of weights. $C$ is the cost and $\eta$ the learning rate.

**for** $n = 1$ to $N$ **do**
  1. propagate forward
  **for** $k = 1$ to $L$ **do**
    $a_k \leftarrow dotproduct(a_k, w_k)$
  **end for**
  2. backpropagate to output layer
  $\Delta w_L \leftarrow -a'_L * (a - t)$
  3. backpropagate to hidden layers
  **for** $k = L - 1$ to $1$ **do**
    $\Delta w_k \leftarrow -\frac{\partial C}{\partial w_k}$
  **end for**
  4. update weights
  $w \leftarrow w - \eta \Delta w$
**end for**

### 1. Feed forward

In a first step, the network is propagated forwards by applying the dot product of weights and activations for each layer as described in 3.2. Additionally, the evaluated derivatives of each neurons activation function for are stored in the unit [35, p. 161].

### 2. Backpropagate output layer

During backpropagation to the output layer, the partial derivatives $\frac{\partial C}{\partial w}$ are calculated and for each neuron multiplied by its evaluated error as shown in in equation 3.12 [35, pp. 167-168].

$$\Delta w_{i,j} = \frac{\partial C}{\partial w_{i,j}} * c_i \tag{3.12}$$

### 3. Backpropagate Hidden Layer

While the error of output neurons can easily be calculated, for the hidden layers, this error for each neuron is dependent on all neurons connected to it. The neurons error

is calculated by equation 3.13 [35, p. 169].

$$\Delta w_{i,j} = \frac{\partial C}{\partial w_{i,j}} * \sum_{q} w_{i+1,q} * C_q \tag{3.13}$$

#### 4. Update Weights

In the last step all neurons weights are updated by applying the previously calculated $\Delta w$ to them. A constant, the learning rate, is used to scale the step size of the correction to the weights [35, p. 170].

If there is more training data available, the process can be repeated and weights corrections can be applied to each set of training data, resulting in the network being able to approximate all patterns available in the training data.

### 3.1.6 Binarized Neural Networks

Binarized neural networks (BNN) are a recently introduced variant of neural networks with weights and activations represented as binary values with the intention to drastically reduce memory size and accesses and as well reduce the computational cost by replacing arithmetic operations by bit-wise operations [5].

#### 3.1.6.1 Binarizing Weights and Activations

In binarized neural networks weights and activations are constrained to the values -1 or 1. Binarizing the networks weights can be performed by transforming the real-valued variables into the binary values [50].

This can be performed in either a deterministic or a stochastic way. While the stochastic binarization shown in paragraph is resulting in a higher accuracy during classification, it requires the generation of random values to calculate the probability. In a computational inexpensive environment, the generation of random values is not desired, since it is lowering the networks performance. The deterministic approach avoids the generation of randomness and thus, is the preferred implementation.

### Deterministic Binarization

With deterministic binarization the networks weights are transformed by the *sign* function shown in equation 3.14. Figure 3.21 is showing the graphical representation of the sign function.

$$f(x) = \begin{cases} 1 & if\ x \geq 0 \\ -1 & if\ x < 0 \end{cases} \tag{3.14}$$

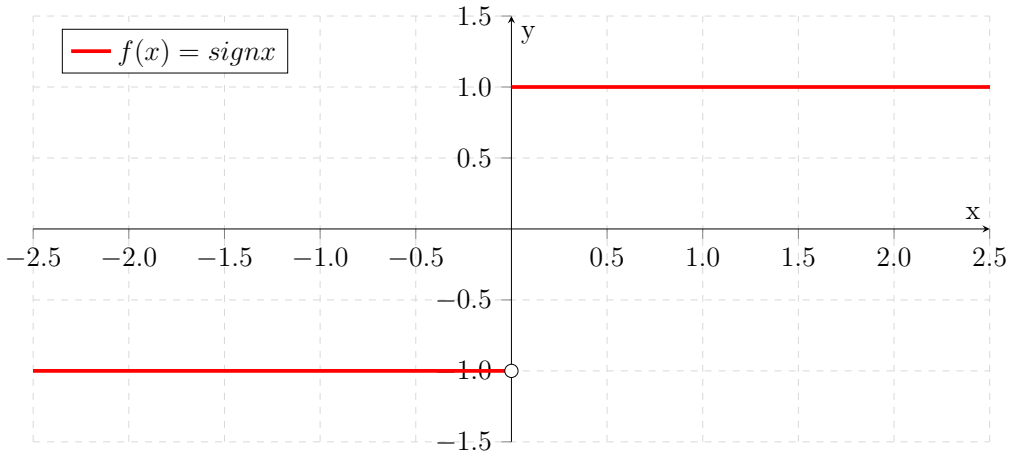



Figure 3.21: Plot of sign function

### Stochastic Binarization

In stochastic binarization the variables are transferred with equation 3.15, where $\sigma(x)$ is the *hard sigmoid* function, defined in equation 3.16.

$$f(x) = \begin{cases} 1 & \text{with probability } p = \sigma(x) \\ -1 & \text{with probability } 1 - p \end{cases} \tag{3.15}$$

$$\sigma(x) = clip(\frac{x+1}{2}, 0, 1) = max(0, min(1, \frac{x+1}{2})) \tag{3.16}$$

#### 3.1.6.2 Binarized Forward Propagation

In a binarized neural network all weights and activations are constrained to $-1$ or 1. This leads to possible optimizations in the calculation of a neurons activation. While in conventional networks the activation vector is multiplied with the weight matrix by applying the dot product, the operation can be simplified to a bitwise XNOR operation [6] in binarized neural networks as shown in table 3.1.

| **x0** | **x1** | $\bigotimes$ |
|---|---|---|
| 0 | 0 | 1 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

(a) XNOR LUT with inputs 0 and 1.

| **x0** | **x1** | $\bigotimes$ |
|---|---|---|
| -1 | -1 | 1 |
| -1 | 1 | -1 |
| 1 | -1 | -1 |
| 1 | 1 | 1 |

(b) XNOR LUT with inputs -1 and 1.

Table 3.1: Lookup-Table for the XNOR function.

Equation 3.17 defines the forward propagation procedure on binarized neural networks [7], where $l$ is the index of the layer, $i$ is the index of the neuron calculated, $a$ is the neurons activation and $b$ the neurons bias. $K$ is the amount of neurons in the layer, $w$ the neurons weights. The $\bigotimes$ symbol describes the bitwise XNOR operation.

$$a_i^l = b_i^l + \sum_j^{K^{l-1}} w_{i,j}^l \bigotimes a_j^{l-1} \tag{3.17}$$

The forward propagation step is similar to those of conventional networks, but with the exception that the dot product operation on activations and weights is replaced by the XNOR operation. Since the first layer is containing real valued inputs derived from the dataset under investigation, it must be preprocessed by binarizing the values. Depending on the range the input parameters are, the exact way of binarization is not defined and must be adapted to the given range of input parameters.

In the case of the MNIST dataset, the input parameters range from 0 to 255, representing grey-scale images. The inputs can be binarized by applying a threshold of 80, following equation 3.18.

$$a_i^b = \begin{cases} 1 & \text{if } a_i \geq 80 \\ -1 & \text{else} \end{cases} \tag{3.18}$$

#### 3.1.6.3 Binarized Backward Propagation

During backpropagation the activation functions derivative is used. For the *sign* functions the derivative is zero on all points, except the zero point of the function, where the derivative is undefined, which makes it incompatible with the backpropagation algorithm.

Algorithm 2 defines the process for backward propagation of a binarized neural network. It is structured analogous to the backward propagation of conventional networks, but is applying modifications in order to overcome the disadvantages binarized weights and activations have to the algorithm.

**Algorithm 2** Backward propagation in a binarized neural network.
Model derived from [5].
$N$ is the amount of training data, $C$ is the cost function for the dataset, $\eta$ is the learning rate, $L$ the number of layers. $w$ are weight vectors, $b$ are biases vectors, $a$ the activation output.

**for** $n = 1$ to $N$ **do**
  1. binarize input
  **for** $i = 1$ to $n$ **do**
    $a_i^b \leftarrow thresh(a_i)$
  **end for**
  2. forward propagation
  **for** $k = 1$ to $L$ **do**
    $w_k^b \leftarrow binarize(w_k)$
    $a_k \leftarrow xnor(a_k, w_k^b)$
    $a_k^b \leftarrow sign(a_k)$
  **end for**
  3. compute gradients
  **for** $k = L$ to $1$ **do**
    Compute $\frac{\partial C}{\partial w_k^b}$
    Compute $\frac{\partial C}{\partial b_k^b}$
  **end for**
  4. update parameters
  **for** $k = 1$ to $L$ **do**
    $\Delta w_k \leftarrow \eta \frac{\partial C}{\partial w_k^b}$
    $w \leftarrow clip(w_k - \Delta w_k^b, -1, 1)$
    $b \leftarrow b_k - \eta \frac{\partial C}{\partial b_k^b}$
  **end for**
**end for**

The networks input layer is binarized and forwards propagation is applied to the network as defined in 3.1.6.2. The gradients are calculated with respect to the partial derivatives of cost and weights as described in 3.1.5. In this step both, the binarized and real-valued gradients are kept for further processing. When updating the parameters the gradients are applied to the real-valued weights and biases. To avoid exploding gradients, the updated parameters are clipped at $-1$ and 1 [50].

## 3.2 Parallel Computing Platforms

Neural networks can be implemented on different platforms that offer sequential or parallel computing structures. This section gives an overview of the platforms commonly used for processing neural networks and other computer vision tasks.

### 3.2.1 Overview of Platforms

The most generic approach to compute artificial neural networks is the usage of a **CPU** (Central Processing Unit). Recent CPU devices implement multi-core design and can have limited vector operations, like SIMD allowing up to 16 operations per clock cycle, for small signal processing tasks and usually have a high operational frequency. CPUs are a very generic and flexible solution, capable of achieving many different tasks. The number of cores is limited to up to 4 to 16 cores which can run on a high frequency of up to 4-5 GHz. CPUs suited for embedded solutions usually implement up to 4-8 cores and have a lower operational frequency of up to 2-3 GHz. Since CPUs are involved in almost every electronic device, prices are generally cheap and solutions for every budget exist. However, the small amount of cores available is limiting parallel computing capabilities and thus CPUs are not competitive with other parallel computing platforms.

In recent research **GPUs** (Graphics Processing Unit) are the preferred solution to implement artificial neural networks [16]. Using a GPU for general purpose processing is an approach called GPGPU (General-Purpose computing on Graphics Processing Units), where one or a cluster of multiple cards is connected to a generic CPU which is feeding data to the GPUs and is reading back computation results.

As shown in figure 3.22 the capability of high parallelism results from more internal hardware dedicated to data processing, compared to a CPU design: GPUs support a big number of cores ranging from a couple hundred to up to thousands, depending on the price range. GPUs are capable of native floating-point calculations. Cores are usually clustered to thread processor clusters that consist of several multi-processors,

cached memory and special texture processing units, where each multi-processor implements one instruction unit with local memory and multiple stream-processors connected to those units. Those stream-processors can execute the same instruction on different data paths or work-items, a mechanism called single-instruction, multiple-data (SIMD). It can be understood as the foundation of data-parallel programming model, where the same program is executed multiple times in parallel on different data elements. To work around memory access delays blocking the cores for execution, the streaming-processors must access successive addresses in local memory. Accessing arbitrary addresses is resulting in blocked execution [51, pp. 1-3].

While the multi-processors each implement their own small amount of local memory with fast access time, they are all sharing the same global memory which has limited bandwidth. This requires the need to implement modified algorithms matching the underlying architecture and providing the correct amount of threads and address accesses for continuous operation.

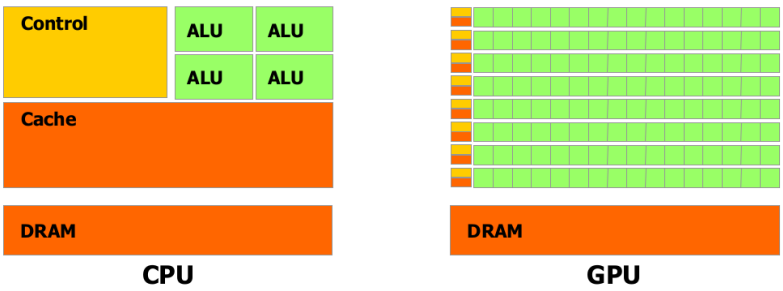



Figure 3.22: Architectural differences between CPU and GPU. The GPU uses more hardware for data processing. Image reused from [51, p. 3].

The design with a high number of cores and fast local memory is resulting in a fast and flexible solution but the enormous performance power is traded for high energy consumption [52].

A **FPGA** (Field Programmable Gate Array) is a highly configurable architecture that can be designed with the usage of hardware description languages like VHDL or Verilog. A FPGA usually consists of multiple modular building blocks like lookup table blocks, adaptive logic modules (ALT), DPS- and memory blocks, which can be arranged in almost any desired configuration, as illustrated in figure.3.23. ALTs can be configured to represent any logic, register or arithmetic function. DSP blocks can implement floating point calculating capabilities in variable precisions and memory blocks can be used for embedded high bandwidth, low latency volatile storage. These programmable resources allow the implementing of highly specialized algorithms by still providing very high performance per watt.

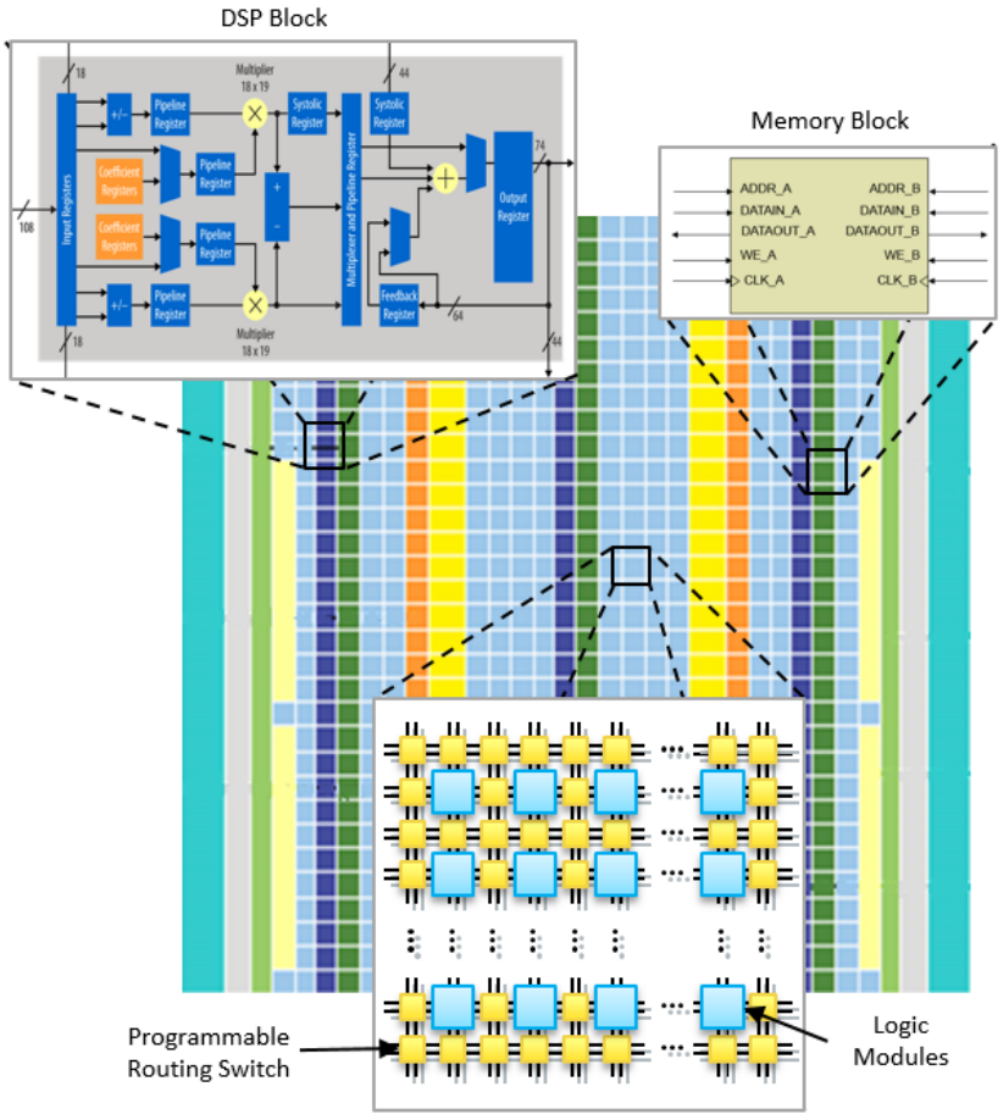



Figure 3.23: FPGA logic resources architecture. Inner blocks can be used as modular building blocks. Image reused from [53, p. 34].

In a FPGA, usually all tasks are performed simultaneously by synthesizing the exact hardware architecture needed for this task. Parallel tasks are usually designed in a pipeline strategy, where different stages of the instruction are applied to the work-items concurrently [54, Ch. 6, pp. 515-530].

The high parallelism and low energy consumption FPGAs offer makes them a good solution for parallel computing [31, 32]. However, compared to GPU and CPU devices the FPGA is the least accessible solution, since programming them by a hardware

description language requires a long time to be implemented and compiled [55].

This problem can be overcome by using one of the frameworks for parallel programming, presented in the next section.

### 3.2.2 Parallel Computing Frameworks

Implementing algorithms on GPU and FPGA is historically done with their native programming interfaces. For GPUs this is native machine code and for FPGAs its the description of hardware synthesized from VHDL or Verilog code. While these interfaces provide the most efficient way of using the underlying hardware, lots of knowledge and insights about the hardware are needed to produce and maintain efficient code. Thus, implementing complex algorithms can result in very time-consuming tasks.

In the last decade, new programming frameworks were introduced to allow faster access to the underlying hardware by abstracting machine language and hardware description to C-Typed languages and wide-spread programming concepts. These frameworks will be discussed in this section.

#### 3.2.2.1 CUDA

Cuda is a general-purpose parallel computing framework, introduced in 2006, that allows the usage of GPUs by the manufacturer NVIDIA for general-purpose computing. It allows the usage of a C-Style interface for developing programs that make use of the GPU as a coprocessor.

With the CUDA parallel programming model the challenge of dealing with parallelism on multiple cores is overcome by providing abstractions to the developer.

There are three key abstractions exposed to the programmer: A hierarchy of thread groups, shared memories and barrier synchronization, providing data- and thread parallelism. The developer is guided to partition algorithmic problems into subproblems that can be solved independently in parallel blocks of threads. These sub-problems can even be decomposed into smaller groups that may be solved cooperatively by threads contained in a block [51, pp. 3-5].

In a CUDA program, C-functions can be defined that, when called are executed multiple times in parallel on different threads which are identified by a 3-dimensional *threadIndex*. This allows the computation of vectors or matrices in a natural way, resulting in a native decomposition of algorithms. Multiple threads are grouped together into equally sized *threadBlocks*.

Data can be accessed from multiple sources: Each thread has local memory assigned and each block contains a shared memory component that is accessible by all threads in the block. There are also additional special-purpose memories available, such as global memory, texture memory and constant memory [51, pp. 8-12].

There are various synchronization mechanisms available that allow synchronization between host and kernel and between different kernels or even between multiple devices executing different kernels [56, pp. 233-237].

#### 3.2.2.2 OpenCL

Like CUDA, OpenCL is a framework, introduced in 2009, assisting the creation of application for parallel processing platforms. While its functionality is similar to the CUDA language, it is not restricted to GPUs and thus can be used on a variety of processing platforms like CPUs GPUs DSPs and FPGAs. With the usage of OpenCL it is even possible to run one program on heterogeneous systems made of different hardware accelerators [57, pp. 2-4]. The abstraction provided by OpenCL is easily portable to different platforms, by simply linking the compiled program against libraries for the underlying hardware platform [58].

The main programming models used with OpenCL are data-parallelism and task-parallelism. Data-parallelism is similar to the SIMD model described for CUDA, where the same task is executed on different work-items, like executing the same operation for multiple cells of a vector in parallel. Task-parallelism describes the technique of executing different threads on multiple processing elements for efficient load-balancing [59, pp. 8-10]. The programming model is derived from the underlying platform model, which consists of the host connected to one or multiple devices. The devices can be heterogeneous where each device has its dedicated execution model.

While the host is creating the OpenCL platform and its context and manages execution of the kernels, the devices run the actual implementation of the kernels. Similar to the kernel execution in CUDA, multiple instances of the same kernel can be allocated usually arranged in two- or three-dimensional arrays of instances, pointed to by an index, called NDRange-index [60, pp. 45-46].

The kernels can have different memory types assigned. Global memory is accessible by all kernels, local memory is only accessible by the device, executing the kernel and private memory is only accessible to the kernel itself [60, pp. 53-54].

Special attention was brought to the implementation of OpenCL running on Intel FPGAs. It allows to use the C-Style language to implement applications on FPGA fabric that formerly was only accessible by using hardware description languages like VHDL or Verilog. Historically OpenCL was mostly used on GPUs, DSPs or other

hardware implementing a SIMD architecture, leading to programming styles similar to those of CUDA.

OpenCL kernels, the C-Style representation of a function executed with OpenCL, were executed in parallel on different data regions. This type of kernel is called NDRange kernel within the OpenCL terminology. With the availability of OpenCL being executed on FPGA fabric, theoretically any hardware architecture can be synthesized [53, pp. 123-133]. This makes single work-item kernels possible, which could theoretically execute a full algorithm within a single clock cycle [53, pp. 105-121]. In contrast to traditional implementations on CPUs, GPUs and DSPs, on FPGAs parallelism is not achieved by duplicating the same generic computation hardware, but rather only the logic the algorithm exercises .

### 3.2.3 Challenges for Parallel Hardware Architectures

GPU and FPGA based accelerators are working well for processing parallel tasks like the computation of neural networks. While both perform well in accelerating parallel processes, they implement parallelism very differently.

The following examples are showing main differences in common use cases met with parallel processing applications on embedded hardware.

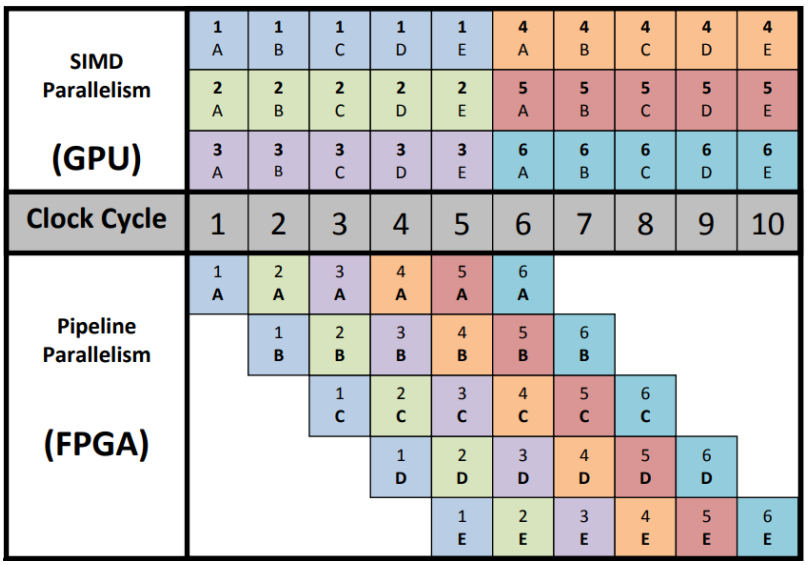



Figure 3.24: SIMD versus pipelined architecture. Image reused from [61].

Figure 3.24 shows the difference between SIMD and pipelined implementations in hardware. Five instructions (A-E) are executed on six work items (1-6). The SIMD can handle 3 work items at the same time, while the pipeline structure adapts to

the amount of instructions and work-items needed. Processing the example, the GPU implementing SIMD strategy completes 3 work items every 5 clock cycles. The pipelined FPGA structure can, after initial 5 clock cycles, finish one work item each clock cycle by executing multiple instructions in parallel.

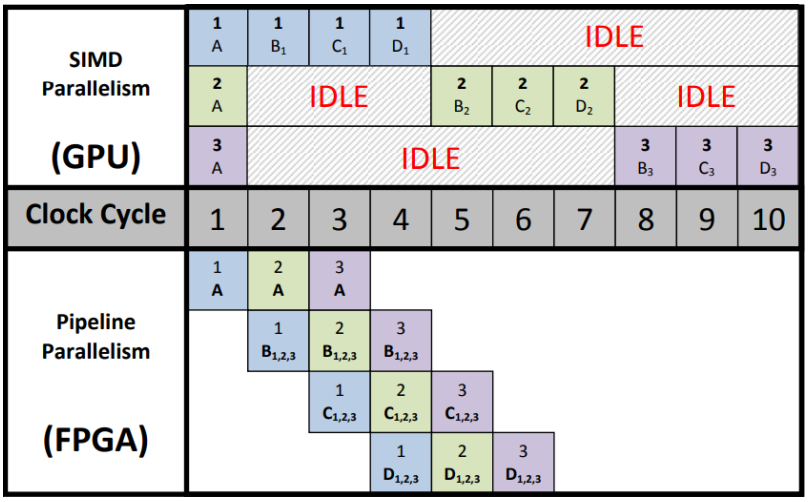



Figure 3.25: SIMD versus pipelined branching. Image reused from [61].

Figure 3.25 illustrates the difference in handling branching. While within the FPGA all possible code paths are already established in hardware before execution, the execution is not different from the previous example and one work item each clock cycle can be achieved. The GPU on the other hand only handles a single instruction at the same time, which leads to conditional execution of the single branches and thus can lead to conditional enabling or disabling of specific work items during execution. This behavior can in the worst-case result in an execution time that is equal to one without parallel strategies implemented.

Figure 3.26 depicts an advantage SoC integrated FPGA can offer compared to GPUs. While with a GPU usually all accesses to IO peripherals go through the host system, a FPGA embedded in a SoC device has direct access to the connected peripherals and thus, the FPGA fabric can access IO peripherals without dealing with the host system.

Even in executing non-parallel tasks, like single work-item kernels, FPGAs can profit from their pipelined structure. As Figure 3.27 shows how single instructions can be pipelined while executing a loop. All instructions can be executed in parallel, where the data executed in loop iterations work as separate work items.

However with the usage of OpenCL FPGA hardware is also partly used in a SIMD structure: When a kernel is implemented as a NDRange kernel, the execution of the instances of this is also performed in a SIMD structure, as it is done with GPUs.

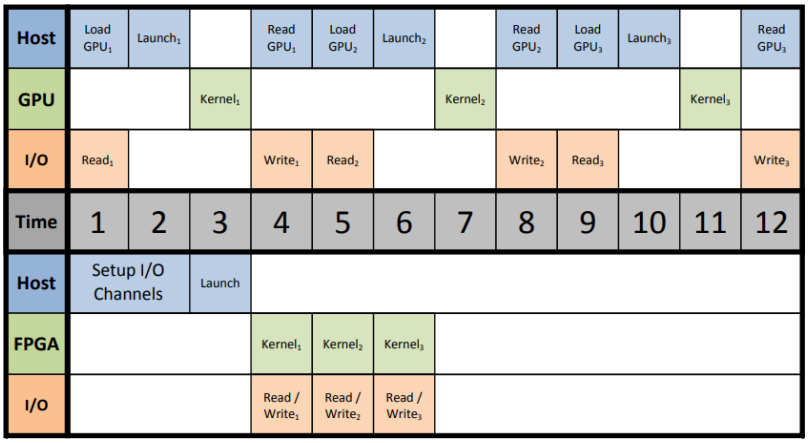



Figure 3.26: SIMD versus pipelined IO access. Image reused from [61].

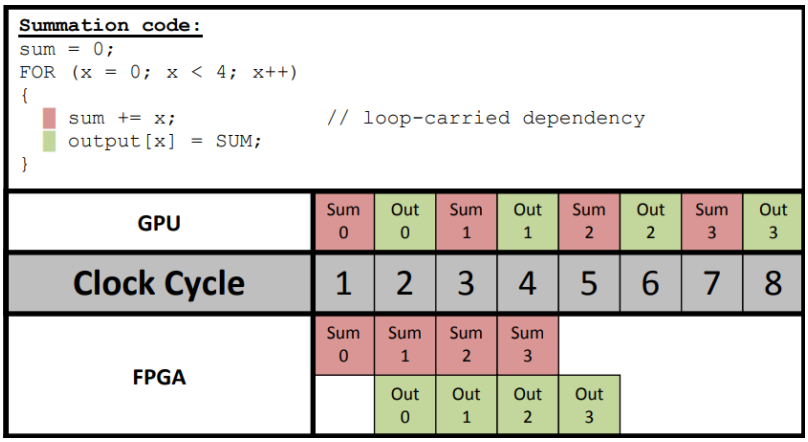



Figure 3.27: SIMD versus pipelined loop execution. Image reused from [61].

## 3.3 Performance Indicators

Comparing different neural networks and their implementation can become a difficult task. They may vary in the number of neurons and hidden layers or measurements can be taken on different architectures. To be able to compare different implementations processing the same task, it is needed to define measurable indicators that can distinguish different features of the network.

### 3.3.1 Classification Accuracy

The most important factor in the process of classifying images is the percentage of images classified correctly. The classification accuracy is highly dependent on the

training data provided, so it is important to only compare networks that were trained with the same datasets.

A good accuracy is considered 80%. An accuracy of 84.4% was achieved in the ImageNet competition by the AlexNet network [16], which is the basis of all modern convolutional network implementations.

### 3.3.2 Execution-Time per Frame

Besides how accurate a network is, another important factor is the time the network needs to process one image. This gives a rough estimation on the implementations performance. Since different authors may implement their work on different platforms, it should only work as a measurement for implementations on the same platform and the same training dataset.

### 3.3.3 FPGA Logic Gates used

One big factor for any algorithm that is supposed to run in an FPGA device is: Will it fit into the FPGA? FPGAs have a limited amount of different resources, like lookup tables, logic gates or DSP blocks. After synthesizing the FPGA logic, the clock might not be sufficient anymore, or the logic needs more resources as the chip offers. Since FPGA prices are growing exponentially with the amount of resources they offer, it becomes important to be able to fit the logic into a chip that is in the projects budget. In mass production it is useful to keep the hardware costs as minimal as possible, because those costs will reoccur and will also increase the more devices are sold.

### 3.3.4 Development Cost

In most commercial projects their owner wants to push the costs to a minimum. Cost of a software project can be measured in lines of code (LOC) that are needed to finish the project. An amount of LOC can be estimated that a developer can write during one day. This amount is different for different programming languages: Programmers using C or C++ usually write less code a day as programmers writing in Python do [62, pp. 55-56].

### 3.3.5 Training Cost

Since training is usually done prior to the shipment to the end user, the time needed to train the network does not affect the experience the user will have with the network. However long training times increase the costs and are getting a big factor when it comes to learning new datasets after deployment of the network or even during runtime.

Thus, it is useful to have low training cost. The cost of training can be estimated by the time the model needs for training.

Since the time can vary enormously between the implementation on different processor architectures, measurements should only be compared, if they were performed on the same hardware.

### 3.3.6 Energy Consumption

Energy consumption of the system is very important to the end user. Usually, implementations on FPGAs are consuming less energy as the same implementation would consume on a GPU. Since the author does not have access to all implementations discussed in this thesis, only comparisons between the used architectures can be chosen for the comparison.

# 4 System Architecture

This chapter discusses the hardware and software architecture used in the implementation of this work.

## 4.1 Intel De0-Nano SoC

The hardware on which the artificial neural network is supposed to be executed determines the operational speed and thus, is the biggest influence for the performance of the network.

As shown by Nurvitadhi et al., the implementation of networks using FPGA devices has superficial benefits over GPU and CPU. This is not only true for conventional neural networks [63], but also for binarized neural networks. While the bitwise structure binarized neural networks offer is suited to be implemented in custom hardware like FPGAs, the implementation also shows a better performance to watt ratio, outperformed only by custom ASIC implementations [64].

Intel's SoC FPGAs integrate FPGAs with systems based around ARM processors and peripherals into a single device. This enables the developer to run complex parallelized tasks on the FPGA while the ARM processor handles normal system IO operations and even may run an operating system like Linux. Usually, in the SoC design the FPGA and ARM cores share system bus and memory and thus can access the same peripherals. This enables the developer to share data between programs running on the ARM cores and parallel FPGA designs. Thus, complex routines can be outsourced to the FPGA without the overhead of transferring it to another processor.

The SoC consists of a dual-core ARM Cortex-A9 and a FPGA part, implementing the Cyclone V SoC (55CSEMA4U23C6N). The hard-processor-system (HPS) consists of a microprocessor unit with dual ARM-Cortex-A9 core processors, flash memory controllers, SDRAM interconnect, on-chip memories, several peripherals like support- and interface peripherals, debug functionality and phase-locked-loops.

The FPGA part contains the FPGA fabric (lookup tables, multipliers, routing, RAMs), PLLs, a control-block, high speed interface transceivers, PCIe controller and

memory controllers. The FPGA fabric is a Cyclone V FPGA with 40k programmable logic elements and 2460 Kbits embedded memory [65, p. 42]. As shown in figure 4.1, the Cyclone V also consists of several (64) DSP blocks that offer floating-point adder and multiplier with IEEE 754 single precision, allowing designs to be synthesized that make use of floating-point calculations. Both, the FPGA and ARM cores have access to all peripherals the SoC offers and all peripherals that are connected to the board.

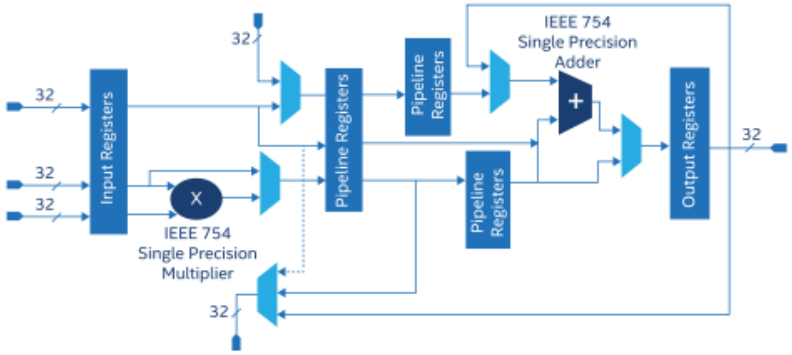



Figure 4.1: Block diagram of a Cyclone V DSP Block. The DSP block contains IEEE 754 floating-point adder and multiplier. Image reused from [65, p. 54].

Both, HPS and FPGA part, are connected through the HPS-FPGA-Interconnect, a BUS system that will be explained in the following section.

### 4.1.1 HPS-FPGA Interconnect

To enable data being transferred between the HPS- and FPGA portion, the Cyclone V SoC implements the HPS-FPGA interconnect, shown in figure 4.2.

The **FPGA-to-HPS bridge** is a high-performance bus offering configurable data width of 32, 64 or 128 bits. It enables transactions form FPGA master to HPS slaves. It also enables full visibility of the HPS address space [65, pp. 602-605].

The **HPS-to-FPGA bridge** is a high-performance bus with configurable data width of 32, 64 or 128 bits. It enables transactions from HPS master to FPGA slaves. It consists of a heavy- and light-weight implementation. The light weight implementation only offers a 32-bit fixed data width, which leads to faster setup times as the full implementation [65, pp. 602-605].

The **interrupt system** from FPGA to HPS enables IP cores inside the FPGA to signal interruptions up to the HPS operating system [65, p. 45].

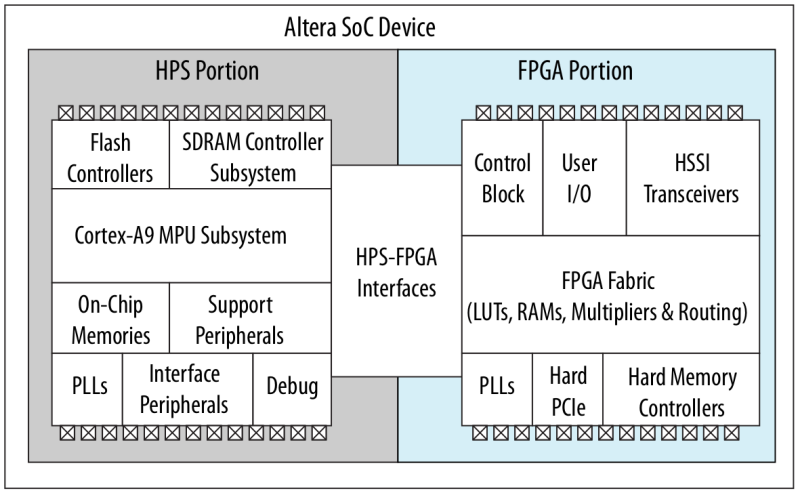



Figure 4.2: Cyclone V HPS-FPGA interconnect. Image reused from [65, p. 40].

The **FPGA manager interface** is an interface used for boot and configuration of the FPGA fabric [65, pp. 166-175].

The **HPS debug bridge** extends the debug capabilities of the HPS to the FPGA fabric [65, p. 45].

While the HPS portion can boot form external flash memory or JTAG, the FPGA must be configured through the HPS or an external programmer. In the configuration that is applied in this work, the ARM core is programming die FPGA by placing the hardware design on an SPI Flash memory exclusively connected to the FPGA. During boot, the FPGA is sourcing the design when the ARM core is booting.

## 4.2 Network Structure

The neural network implemented during the research of this thesis is based on a feed forward network structure.

While usually convolutional networks are used for image classification tasks and show best performances, they tend to show best performances when implemented as very deep networks, like AlexNet [16] or even ResNet [19]. The implementation of such big networks in low-level programming languages as well as the huge training periods they require, would consume more time as what is allocated for the practical work of this thesis. All expected results that can be derived from investigating binarized feed forward networks can easily be transferred to other network types. Considering the limited hardware on which the practical work of this thesis is supposed to run the decision was made towards the implementation of a feed forward network.

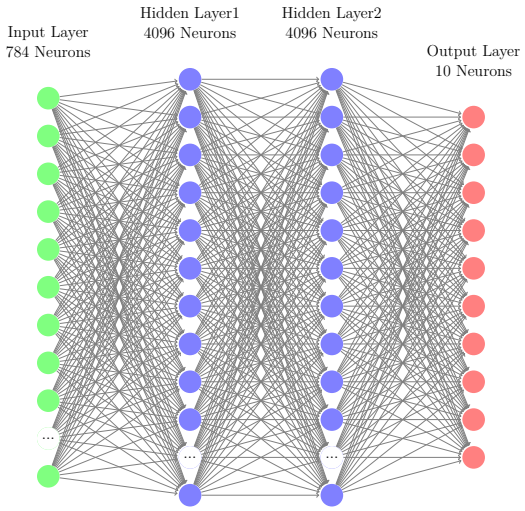



Figure 4.3: Graph of the implemented network. The network consists of 384 input neurons, 2 hidden layers with each 4096 neurons and an output layer of 10 neurons.

### Network Layout

Figure 4.3 illustrates the network architecture. The **input layer** consists of 384 neurons. The EMNIST dataset is made of images of $27 * 27$ pixels of grey-scale values ranging from 0 to 255. Each pixel can be translated to one input neuron in a network, which results in 784 input neurons. The pixels grey-scale value is scaled into a range from $-1$ to 1 and is afterwards binarized to the values $-1$ and 1.

The **output layer** is made from 10 neurons with binarized output values of $-1$ or 1. The goal of the network is to be able to classify number from 0 to 9, resulting in 10 classes, each represented by a single neuron in the output layer.

Input- and output layer are connected through one or more **hidden layers**. While with a conventional network one hidden layer with less than 1000 hidden neurons is needed show acceptable accuracy when classifying digits, binarized networks require a minimum of 2 layers with 2048 neurons to reach an accuracy of more than 50%. Results with an accuracy over 85% could only be reached by using 2 layers of 4096 neurons.

## 4.3 OpenCL Integration

The usual work-flow to synthesize algorithms for FPGAs is the usage of the tool-chain provided by the FPGA manufacturer, which involves designing the algorithm with a hardware description language like VHDL or Verilog with respect to hardware resources offered by the specific FPGA that is used.

An interface between host processor and FPGA has to be designed as described in 4.1.1 to transfer data between the two entities. A behavioral simulation of the design has to be executed and potential changes to the design must be made until the simulation behaves as described. If the simulation passes, the design is **synthesized** to a netlist format, describing the final circuit with logic elements and available FPGA resources.

After synthesis of the hardware description the design is implemented by **translating** the netlist to an intermediate format with respect to timing constraints and physically available pins of the actual chip.

The translation is followed by **mapping** this format to available building blocks of the underlying FPGA fabric and a final **place and route** step, where the mapped blocks are placed and connected on the real FPGA fabric in a way that respect previously created timing constraints. With a functional simulation the designs functionality can be verified end eventually a static timing analysis is performed to provide a comprehensive timing report [66, Ch.4, pp. 79-104].

Since these steps require a deep knowledge of the exact FPGA fabric being used, formulating complex algorithms with help of a hardware description language by connecting logic blocks is a time-consuming task. With the possibility to use Intel's SoC platforms with OpenCL through the SDK provided by Intel, the OpenCL design process is drastically decreasing the time needed for implementing complex algorithms.

The ARM core on the Cyclone V SoC is booting an embedded Linux operating system provided by Intel, that is started from SD-card. The operating system provides an OpenCL driver and Run-Time environment (RTE) that allows the usage of the FPGA fabric from any application.

The application executing the binarized neural network is composed of two parts: The host application and the OpenCL kernel running in the FPGA fabric. Figure 4.4 is illustrating the OpenCL programming model for FPGAs: The OpenCL kernel is compiled from the kernel source file developed by the author with the help of the OpenCL SDK compiler. It is developed in the language C and translated to optimized VHDL or Verilog Code by the compiler, which then gets synthesized and mapped to an FPGA image with the FPGA tool-chain and under respect of the

board specific design files provided by the board manufacturer and FPGA supplier [1, p. 7-12].

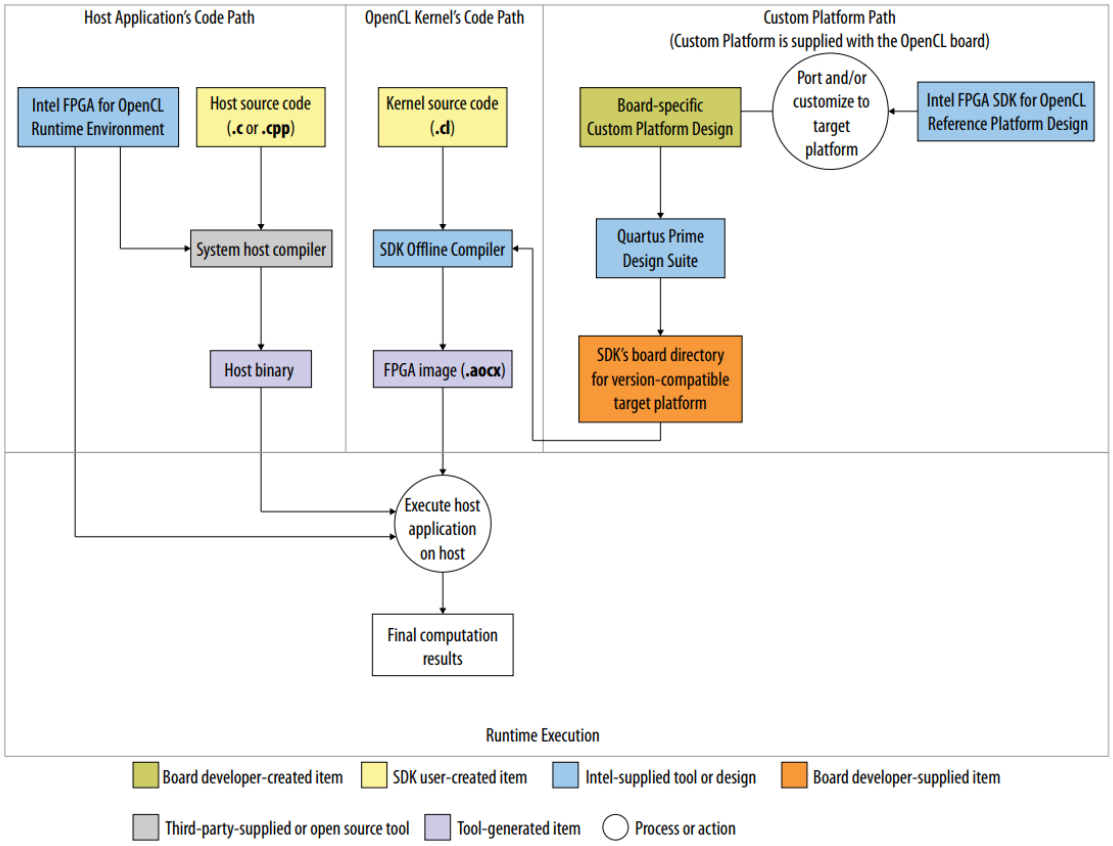



Figure 4.4: Schematic diagram of the OpenCL programming model when programming Intel FPGAs. Image reused from [1, p. 7].

The host application, developed by the author, is making use of the Intel FPGA for OpenCL SDK to provide all the functionality needed to make use of the OpenCL kernel within the FPGA image and configuring the OpenCL RTE. It is reading the data needed to compute the binarized neural network from SD card, allocating buffers that can be used form the FPGA and is providing them to the OpenCL kernel.

Figure 4.5 illustrates the program flow of classifying one image. The ARM core starts by reading the image from DDR SDRAM memory, and filling the input layer of the neural network. For each layer it writes the data needed for processing that layer to SDRAM and initiates the FPGA acceleration of the calculation of the dot product.

For simplification, this illustration depicts the processes executed inside the FPGA as one single process, while in fact multiple processes are executed in parallel inside the FPGA. A more detailed view of those processes is illustrated in 4.6.

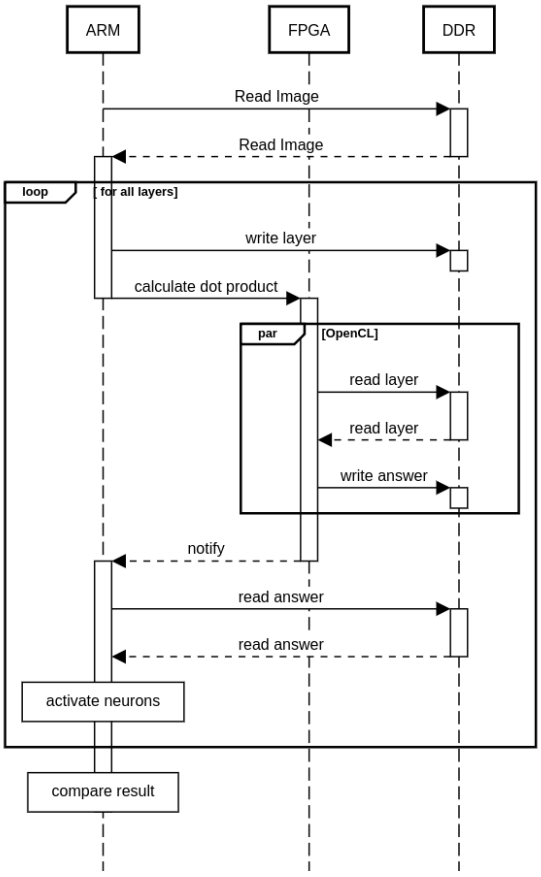



Figure 4.5: Sequence Diagram for ARM and FPGA cores accessing DDR SDRAM.

The FPGA is reading the required data and is writing back the answer to DDR SDRAM memory. Afterwards it informs the ARM core about finishing the processing. The ARM core reads back the answer and is finishing applying the activation function to the dot product calculated by the FPGA core. After all layers of the network are processed, the result in the output layer is verified to be the correct representation of the data.

While in figure 4.5 the processing done in the FPGA is represented as a single thread, figure 4.6 is providing a deeper view into the scheduling inside the FPGA. The FPGA is calculating the dot product of a matrix of weights and a vector of activations. To achieve this, each row of the matrix is multiplied with the activation vector and the results are added.

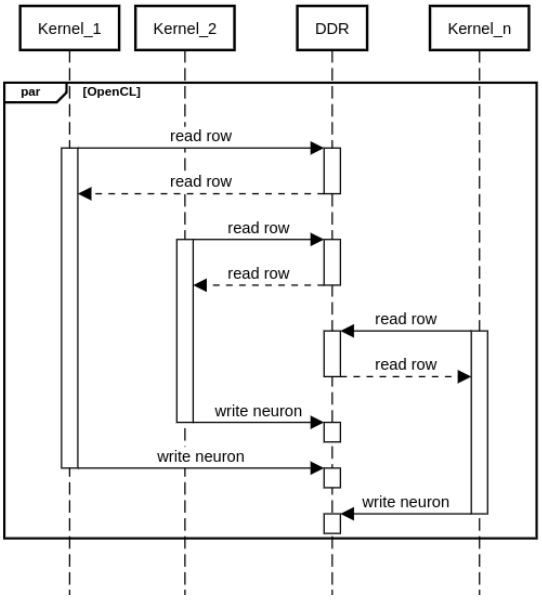



Figure 4.6: Sequence Diagram for OpenCL Kernels accessing DDR SDRAM.

Internally, for each row of the matrix an OpenCL kernel is executed, executing the multiplication and addition and writing the result back to the corresponding cell in the output vector. Those threads are executed in parallel and are scheduled by the OpenCL scheduler subsystem. Each thread is accessing only the row in the matrix that it needs access to, by executing its own SDRAM memory accesses.

# 5 Implementation Details

In this chapter the implementation of the neural network developed in this thesis will be discussed. Furthermore, the transition from a conventional to a binarized network and details about the implementation in the SoC using OpenCL are presented.

## 5.1 Network Implementation

The Neural network is implemented as a multilayer network where all neurons of each layer are connected to all neurons of the next layer. Several implementations were tested that differantiate in the number of hidden layers.

### 5.1.1 Input Preprocessing

While data fed into the network can be any scale, a better accuracy can be achieved by scaling input data according to the activation functions point of operation. In the case of the EMNIST dataset there are 28x28 pixel of grayscale images. The values range from 0 to 255. This is resulting in 784 fields of input data. If the hyperbolic tangent function is chosen it is useful to scale the input values between -1 and 1. Also lots of different types of pre-filtering can be done in this step. The dynamics of the image could easily be changed by introducing a threshold from which pixels are stared being recognized.

### 5.1.2 Hidden Layers

The number of hidden layers in a neural network is usually growing with its complexity. An additional layer of input neurons can be understood as an additional step of filtering attitudes out of the image. The network is implemented in a way that unlimited hidden layers can be implemented during initialization. Each hidden layer can have a distinct number of neurons.

While conventional neural networks usually work with a smaller amount of neurons inside the hidden layer, after initial testing it turned out useful to have at least 2048

neurons in each layer when using binarized neural networks. State-of-the-art results can be achieved starting with 2 layers, each consisting of 4096 neurons.

### 5.1.3 Activation Function

The activation function of the network reads the sum of all activations and is applying a mathematical function f(x) to it. This function can theoretically be any function imaginable. This network was tested with hyperbolic tangent, sigmoid, hardsigmoid and the step function.

In binarized networks, best results were achieved with the hyperbolic tangent function. Since hyperbolic tangent has a faster changing gradient around -1 and 1 as for example the sigmoid function does, during backpropagation the biggest changes of the weights can be achieved. This is needed in binarized networks, since they only support output values of -1 or 1 and need big changes in weights to make a switch from one output value to the other happen.

Since the derivative of the hyperbolic tangent shows bigger gradients on the points (1,1) and (-1,-1) as for example the sigmoid functions derivative, it also has bigger potential to learn binarized networks. This justifies by the fact that during training the back-propagation algorithm is used, where the error-signal is used as an input for the derivation of the activation function to determine the weights deltas. Binarized networks need big deltas in weights to change their output values from 1 to –1 or vice versa,

With conventional networks it is rather useful to stay inside the area around the zero-point of the activation function, since it allows intermediate instead of discrete values, small changes in the weights are needed to work best with the SGD algorithm.

```
1
2  float act_sigmoid(float x) {
3      return 1.0 / (1.0 + (exp(double)-x));
4  }
5
6  float der_sigmoid(float x) {
7      float s = act_sigmoid(x);
8      return s*(1.0-s);
9  }
10
11 float act_tanh(float x) {
12     return tanh(x);
13 }
14 float der_tanh(float x) {
15     return 1-pow(tanh(x),2);
16 }
17
18 float act_hard_sigmoid(float x) {
19     return max(0.0, min(1.0, (double)((x+1)/2)));
20 }
21 float act_hard_tanh(float x) {
22     return (2*act_hard_sigmoid(x))-1;
23 }
24 float act_step(float x) {
25     return x>=0?1:-1;
26 }
```

Listing 5.1: Implementation of the activation functions and their derivatives

### 5.1.4 Weights Scaling

As described in the previous paragraph, binarized neural networks show the best performance around the points (1,1) and (-1,-1). To guarantee this, it is required to directly hand the weighted inputs to the activation function after integrating them.

In an artificial neuron a sum over all weights is built. While with conventional neural networks it is useful to scale this sum by dividing it by the number of weights, in binarized networks it turned out to decrease the performance drastically. In conventional networks, linearity of an activation function is wanted, because it allows

for small in the output while only providing small changed to the input value. As introduced in chapter 3.1.4, during backpropagation a high gradient is needed

### 5.1.5 Forward Propagation

During forward propagation, a loop over all layers is performed, where in each iteration an OpenCL kernel is executed to calculate the dot product of the weights-matrix and the activation-vector. The implementations of all OpenCL Kernels implemented are described in detail in section 5.2. After calculating the dot product, the neurons activation function described in 5.1.3 is called.

### 5.1.6 Gradient Calculation

Courbariaux is implementing his work with help of the Theano framework, which enables gradient calculation using complex algorithms and thus resulting in a long training time. Since this works implementation is intended to run on embedded systems with limited resources, the choice was made to implement a simplified gradient calculation.

The cost $\Delta Cost$ of an output neuron is calculated by $Output_{desired} - Output_n$. The output layers neuron output is used as an input for the activation functions derivative and is then scaled by the factor $\Delta Cost * LearningRate$, resulting in $\Delta Weight$

### 5.1.7 Binarizing Weights

Courbariaux is implementing weights binarization in the Thenao implementation in a way that each node in the networks processing graph will binarize the weights prior to processing. This results in redundant binarization steps, calculating the same values for each node.

This work implements weight binarization in a way that prior to processing a new matrix of weights is allocated, where the binary values are stored. As chapter 3.1.6.3 introduces, during training the original weights of floating point type, as well as the binarized weights are needed, so allocating the new binarized weight matrix is no redundant step. During each step in the training process, binarized and non-binarized weights are updated, if needed. This reduces the amount of binarization steps drastically and is resulting in a much faster training process.

The binarization itself is implemented by looping over all weights, while testing if the weight is greater or equal to zero.

### 5.1.8 Network Optimizations

During implementation several optimizations were applied to the binarized network to gain better results or lower the processing time. The following paragraphs will present and discuss those optimizations.

#### 5.1.8.1 Parameter Initialization

The networks weights are initialized with random numbers between -1 and 1. While this is showing a good learning performance on networks implementing 1 hidden layer, the performance decreases with the number of hidden layers. If the weights of a network are too small, passing multiple layers could make the signal too small until it reaches the output neurons. If they are too big, while passing through the network, they may grow too big and lose their usefulness.

In [67] Glorot is presenting a method of initializing the networks parameters in a reasonable range by making them dependent on the amount of inputs and output connected to the layer. $Var(w) = \frac{2}{n_{in}+n_{out}}$

Glorot initialization, originally proposed by Xavier Glorot and Yoshua Bengio in [67], is an initialization technique for weights where the variance of a layers output neuron is tried to be made equal to the variance of its inputs.

While small networks, consisting only of a few layers, can be initialized by normal distribution with a mean of 0.0, for deep neural network with many layers other initialization methods allow a more efficient training.

Initializing deep networks with normal distribution and $\sigma = 0.01$ leads to the neurons activations very close to zero. By traversing through the network and applying for example the tanh activation for all neurons, the activations will get smaller the closer they come to the output layer, which leads to the "vanishing gradients" problem. During back propagation this leads to gradients close to zero in all layers which will result in inability to learn properly.

This could be avoided by using $\sigma = 1.0$ for initialization with normal distribution. This can easily lead to saturation inside the activation function, so the neurons output will be either +1 or -1, which, again, makes gradient calculation tough, because without change in output values, the gradients will get close to zero, like they do with small variance.

Also, the exploding gradients problem can occur then weights and thus activations get too big.

$$weight_{scale} = 1/\sqrt{\frac{2}{n_i + n_{i+1}}} \tag{5.1}$$

Equation 5.1 describes how the weights are initialized.

#### 5.1.8.2 Learning Rate Decay

The learning rate is one of the most important parameters to configure for neural networks.

It is responsible for scaling $\Delta w$ and $\Delta b$ in the direction of the gradient. Low training rates are in general more reliable because the steps taken for weight changes are very small and the neuron's error is recalculated with every step. The smaller these changes, the more steps are needed to reach towards the minimum of the loss function, so, the overall training time increases. If the learning rate is small and is reaching a local minimum of the error function, it might not be able to leave this minimum anymore, because the step size is too small [68]. While big changes can speed up the training process by bringing the parameters very fast close to the desired target, they may overshoot it and lead to a diverging behavior [69].

After each training cycle, the learning rate is multiplied by a static factor. Equation 5.2 is showing the calculation of the step size for one cycle, where $n$ is the number of training cycles and $\eta_{start}$ and $\eta_{end}$ are the start- and end-point of the learning rate.

$$\Delta\eta = (\eta_{end}/\eta_{start})^{1.0/n} \tag{5.2}$$

#### Glorot Learning Rate Scaling

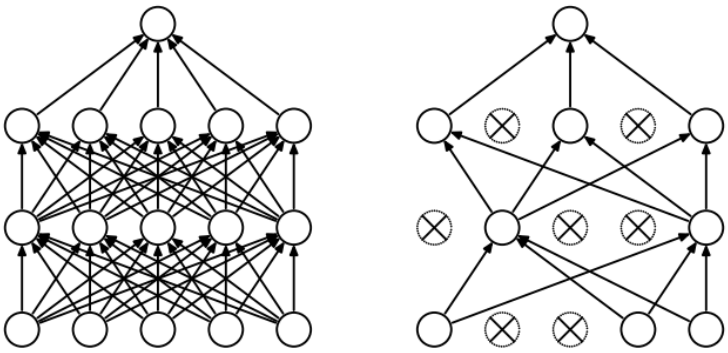
Analogous to the initialization of the networks weights presented in chapter 5.1.8.1, it is useful to scale weights initialization according to Glorots parameters [67]. To maintain those scaled values through the entire training process, also the learning rate can be scaled by the parameters Glorot defined. This process guarantees that the input to the activation function will stay in the range that Glorot proposes. The final scaling parameter is calculated by equation 5.3

$$lr_{scale} = 100/\sqrt{\frac{1.5}{n_i + n_{i-1}}} \tag{5.3}$$

In this implementation the factors slightly differ from Glorots proposal and the weight initialization chosen. During implementation other network optimizations were implemented and those parameters have statistically proven the best results.

#### 5.1.8.3 Dropout

Dropout is a technique proposed in 2014 [70] that can overcome overfitting of large networks. It evolved from the technique of adding random noise to the networks parameters during the training process. Instead of adding noise, the network is thinned out by randomly dropping neurons together with their connections from the network, as figure 5.1 visualizes.



(a) Standard Neural Net (b) After applying dropout.

Figure 5.1: Visualization of the dropout technique. A is depicting a default neural network. B depicts the same network thinned out by applying dropout. Crossed units are dropped. Image reused from [70].

#### 5.1.8.4 Shuffle Dataset

For all methods implementing the SGD algorithms it is important for all training samples to be trained independently. While visiting the training samples in fixed order, as they are represented on the storage media, reduces random access to the disk, faster convergence of the networks approximation can be observed when accessing the training samples in random order. For a better convergence, the samples can be pre-shuffled on disk to increase convergence [48] [71, pp. 123].

#### 5.1.8.5 Exaggerated Targets

In the training process of neural networks, the cost for each output neurons error is calculated and backpropagated through the network to adjust the weights. While with conventional networks an error of 0 is almost never reached, because the output neurons activations are slowly converging towards the desired output, binarized networks do not show this behavior. Their output neurons activation function only has binary output values of [-1, 1] or [FALSE, TRUE], which leads to the fact that the error for some output neurons activation is 0, but the result still is wrong. The network paths leading to this result will not be touched during the learning process. Since in the training of binarized networks with the stochastic gradient descent algorithm, weights are, like in the training of conventional networks, adjusted by small changes to reach the global minimum of their error function.

For classification problems this can lead to problems: Most output activations are supposed to be FALSE, while only one output is supposed to be TRUE. In a situation where, during classification of numbers from 0 to 9, the wrong class is detected, only the path of 2 output neurons would be adjusted, while 8 are considered to have reached their global minimum. This can lead to long training times, because after the network made some initial learning progress, only few weights are adjusted at each training cycle.

In neuroscience, the Hebbian theory [72] describes a basic mechanism for the adaptation of neurons in the brain during a learning process. Hebb describes:

> Let us assume that the persistence or repetition of a reverberatory activity (or "trace") tends to induce lasting cellular changes that add to its stability.[...] When an axon of cell A is near enough to excite a cell B and repeatedly or persistently takes part in firing it, some growth process or metabolic change takes place in one or both cells such that A's efficiency, as one of the cells firing B, is increased.

This theory can be summarized as "What fires together wires together" and is a mechanism that can be observed often in nature: People tend to adapt behavioral pattern that work well in certain situations. A view on the same theory on a meta-scale. The theory can be interpreted and transferred to the problem in a way that connections between neurons should get stronger if they lead to the desired output.

Applying this theory to the problem described can be done by exaggerating the training data in the case of not detecting any error as shown in listing 5.2. By scaling the exaggeration with the learning rate, it can be adjusted through the learning process. During the end of the training, exaggeration will have less impact on the network, to avoid vanishing gradients.

```
float CostFunction(float input, float target)
{
    if(input == target) {
        float scale = scaleParameter * target;
        target += scale * learningRate;
    }
    return input - target;
}
```

Listing 5.2: Exaggerating the error of a neurons activation.

#### 5.1.8.6 Converging Activation Function Replacement

To the user, the time needed for executing forward propagation is directly influencing the experience the user has with the network. The faster a network can classify an image, the more responsive it is.

One factor that can influence the forward propagation time is the type of activation function used in all neurons. On most hardware computational expensive functions like trigonometric functions are implemented with a combination of lockup tables and interpolation, which can lead to high computation times. Since this function is executed once for every neuron in the network, optimizations show a great influence on execution time.

As shown in chapter 3.1.5, the backpropagation algorithm requires differentiable functions to show best performance. Since easy to compute functions, like the step function or a saturated linear function are not differentiable or have a derivative of 1 for certain input ranges, they should be avoided to be used with the backpropagation algorithm.

An optimization to overcome the problematic, while still being able to make proper use of the backpropagation algorithms advantages is to use functions that converge around the desired input range. Examples of converging functions are presented in 5.2. While during backward propagation the computationally expensive function $f(x)$ is used, during forward propagation, it can be replaced by a simpler function $g(x)$ that converges to $f(x)$ in the desired input range.

This modification only shows good performance with binarized networks, because of the binarized output of the activation function, which allows a big input range to produce no change the functions resulting output value.

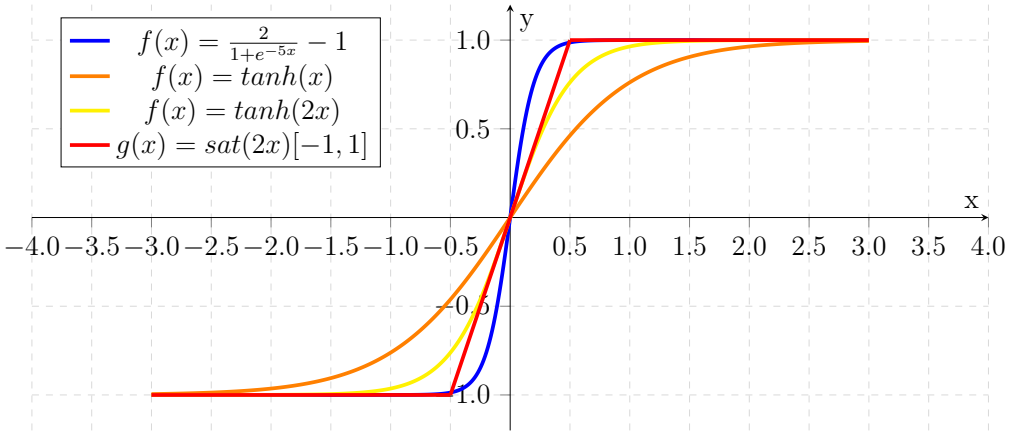



Figure 5.2: Converging functions that can replace each other.

## 5.2 OpenCL Implementations

To be able to record and compare measurements of different version of he neural network, several OpenCL kernels were created. All the implementations use a mixture of a NDRange kernel and a single-work-item kernel. The weights matrix consists of $n$ rows and $n_{-1}$ columns, where $n$ is the amount of neurons of the layer that will be calculated and $n_{-1}$ is the previous layers number of neurons.

Each row of a matrix calculation is implemented as a single-work-item kernel, meaning all the operations executed for processing this row are executed within one execution cycle of the kernel. Processing of multiple rows is done by executing them inside a NDRange kernel. In NDRange kernels multiple threads of the same kernel-code are pipelined and executed by a thread scheduler.

Inside the single work item kernel all loop-carried dependencies were removed, by simply pre-caclulating them before the loop executes [53, p. 106]. All loops and nested loops have integer bound exit conditions to speed up execution and pointer aliasing is avoided by making proper use of the restrict keyword [53, p. 121].

### 5.2.1 Matrix Dot Product

Listing 5.3 is presenting an OpenCL kernel implementing a matrix dot product, which is used for processing inside conventional neural networks.

```
__kernel void vectorMult
  (__global const float * restrict weight,
   __global const float * restrict input_neuron,
   __global float * restrict output, int input_length)
{
  // get index of the work item
  // this equals the row we are processing
  int index = get_global_id(0);
  // row iterator
  int it_r = index * input_length;
  int sum = 0;
  // iterate through cols (it_r + i)
  // apply weight to input and sum up
  for(int i=0;i<input_length;i++)
  {
    sum += weight[it_r + i] * input_neuron[i]);
  }
  // write sum into output buffer
  output[index] = sum;
}
```

Listing 5.3: Implementation of a matrix dot product in OpenCL.

The kernel receives 2 input buffers for the neurons weights and the previous layers activation output, as well as an output buffer for this layers activation output. To allow optimization, the pointers to those buffers have the restrict type qualifier set. This indicates to the compiler that the memory the pointer points to is only accessed through this pointer. This is avoiding pointer aliasing and allows the OpenCL compiler to implement caching optimizations.

Multiple threads of this kernel are instanced and scheduled by the OpenCL framework to process all the matrix's rows. An index is assigned to each thread, representing the rows index. This index can be read inside the kernel by receiving the global id from the OpenCL framework. This is done by calling the function `get_global_id(0)`.

The offset of the row to be processed in this kernel instance is calculated by the number of matrix columns multiplied by the amount of neurons inside the previous layer: `it_r = index * input_length;`

The sum of all weighted input is built by looping over all elements of the row that is processed and multiplying them with the last layers output of the corresponding neuron: `sum += weight[it_r + i] * input_neuron[i];` After the sum is build, it is written back to the output buffer at the index of the thread.

### 5.2.2 XNOR Product

Listing 5.4 is showing the processing of the XNOR product within an OpenCL kernel. It is implemented analogous to the previously discussed dot product.

```
__kernel void vectorMult
  (__global const unsigned char * restrict weight,
   __global const unsigned char * restrict input_neuron,
   __global float * restrict output, int input_length)
{
  // get index of the work item
  // this equals the row we are processing
  int index = get_global_id(0);
  // row iterator
  int it_r = index * input_length;
  int sum = 0;
  // iterate through cols (it_r + i)
  // apply weight to input (xnor) and sum up
  for(int i=0;i<input_length;i++)
  {
    sum += (weight[it_r + i] == input_neuron[i])?1:-1;
  }
  // write sum into output buffer
  output[index] = sum;
}
```

Listing 5.4: Implementation of the XNOR product in OpenCL.

Since the XNOR product is used in binarized neural networks, the floating point input buffers can be replaced by buffers of the data type `char`. For optimization purposes, the output buffer still has the data type `float`. This is done, because the output values of the kernel still have to be used as the activation functions input parameters which are supposed to be floating point values.

The main difference to the kernel computing the dot product is how the weighted inputs are calculated: The multiplication from the dot product is replaced by the XNOR logic function: `weights[it_r + i] == input_neuron[i];`. In C-Style programming languages, the XNOR function can be represented by the equality operator `==`. As shown in table 3.1 the output of the XNOR function is only 1 if the input values are equal.

### 5.2.3 Compressed XNOR Product

```
__kernel void vectorMult
  (__global const unsigned int * restrict weight·,
   __global const unsigned int * restrict input_neuron·,
   __global int * restrict output, int input_length,
   int process_max)
{
  // get index of the work item
  // this equals the row we are processing
  int index = get_global_id(0);
  // row iterator
  int it_r = index * input_length;
  int sum = 0;
  int to_process = process_max;
  // iterate through cols (it_r + i)
  // apply weight to input and sum up
  for(int i=0;i<=input_length;i++)
  {
    for(int bit=0;bit<32;bit++)
    {
      if(to_process)
      {
        char a = (weight[it_r + i] >> bit) & 0x1;
        char b = (input_neuron[i] >> bit) & 0x1;
        sum += (a == b)?1:-1;
        to_process--;
      }
    }
  }
  // write sum into output buffer
  output[index] = sum;
}
```

Listing 5.5: Implementation of the compressed XNOR product in OpenCL.

For calculating the compressed XNOR product, as shown in listing 5.5, input values for weights and the output of last layers neurons are compressed by only using 1 bit for each values representation. 32 bits are stored inside an unsigned integer.

To process the XNOR operation from the compressed buffers, the bits must be un-

compressed by using the shift operation `(x >> bit) & 0x01`. Since it can not be guaranteed that the amount of weights or inputs are multiples of 32, the exact amount must be set as the input parameter `process_max`.

Usually the XNOR operation could be implemented as a bitwise operation over the whole 32 bit integer by negating a XOR operation: `xnor = ~(A^B)`. It was chosen not to use this implementation, because still the sum of the results of the XNOR operation has to be built by looping over all bits. Doing the bitwise operation while still implementing the loop would use more resources.

# 6 Analysis

In this chapter performance measurements of practical outcome of this thesis will be presented. They will be discussed according to the performance indicators defined in this thesis and compared to measurements of conventional neural networks as well as to other authors results.

## 6.1 Performance Measurements

The following section will present measurements of implementations of a neural network predicting the EMNIST classification benchmark

The implemented network consists of one input layer, one output layer and one or two hidden layers with 4096 neurons each. Measurements on processing times of the single layers are taken and presented.

### 6.1.1 Classification Accuracy

The classification accuracy is the percentage of images classified correctly. Figure 6.1 lists the accuracy for different implementations of MNIST handwritten number classification. Compared are four implementations derived from this thesis and two implementations implemented by other authors.

| **Implementation** | **Classification Accuracy** |
|---|---|
| 1 Hidden Layer ANN | 97% |
| 1 Hidden Layer XNOR | 87% |
| 2 Hidden Layer ANN | 98% |
| 2 Hidden Layer XNOR | 92% |
| Huynh | 97% |
| Courbariaux | 96% |

Table 6.1: Classification Accuracy for different network models.

The networks implemented in this paper are 1 and 2 hidden layer implementations of the XNOR network and a conventional network with its operating point close to (0,0). While the conventional implementation gives an accuracy 0f 97-98%, the BNN implementation gives accuracy of 87-92%. Huynh states and accuracy of 97% [9], Courbariaux 96% [8].

### 6.1.2 Execution-Time Per Frame

Besides the accuracy of the network, the execution time is a good measurement for the networks performance. The following tables show processing times for ARM-only implementations compared to FPGA-accelerated implementations of the same network. The measurements only contain the kernels processing time, excluding the time needed to transfer the images from ARM to FPGA core. This is done, because in a real-life scenario images are usually coming from an outside source, like a camera. Since the FPGA core has direct access to all the SoCs peripherals, the measurements of interest are the raw kernel processing times.

#### ARM Implementation

Table 6.2 is showing the processing time for the layers involved in performing the classification. The column **ANN** lists timings for the conventional artificial neural network implementation, using floating-point multiplication for weights and activations. Column **XNOR** shows timings for the XNOR implementation of the network using 8-bit integers to store weights and activations. The column **Compressed** lists timings for the compressed XNOR network, where weights and activations are stored as 1-bit values.

| **Resource** | **ANN** | **XNOR** | **Compressed** |
|---|---|---|---|
| Hidden Layer 1 (784x4096) | 308.066 ms | 103.102 ms | 144.344 ms |
| Hidden Layer 2 (4096x4096) | 1654.475 ms | 687.858 ms | 963.014 ms |
| Output Layer (10x4096) | 4.047 ms | 1.683 ms | 2.641 ms |
| 1 Hidden Layer Network | 312.145 ms | 131.783 ms | 147.028 ms |
| 2 Hidden Layer Network | 1966.782 ms | 819.191 ms | 1110.042 ms |

Table 6.2: Timings for ARM-based implementation of a conventional artificial neural network.

## SoC ANN Implementation

Table 6.3 lists processing times using the ARM core and accelerating matrix operations with the FPGA. The column **ANN** lists timings for the conventional artificial neural network implementation, using floating-point multiplication for weights and activations. Column **XNOR** shows timings for the XNOR implementation of the network using 8-bit integers to store weights and activations. The column **Compressed** lists timings for the compressed XNOR network, where weights and activations are stored as 1-bit values.

| Resource | ANN | XNOR | Compressed |
|---|---|---|---|
| Hidden Layer 1 (784x4096) | 45.812 ms | 45.315 ms | 32.966 ms |
| Hidden Layer 2 (4096x4096) | 458.964 ms | 439.686 ms | 205.096 ms |
| Output Layer (10x4096) | 2.529 ms | 1.802 ms | 1.612 ms |
| 1 Hidden Layer Network | 48.341 ms | 46.952 ms | 34.578 ms |
| 2 Hidden Layer Network | 507.305 ms | 486.803 ms | 239.674 ms |

Table 6.3: Timings for SoC-based implementation of a conventional artificial neural network.

### 6.1.3 FPGA Logic Gates used

Table 6.4 lists the amount of FPGA resources used in different implementations of image classification networks running on FPGA hardware. Listed are other authors implementations compared to 3 different implementations derived from this work: An artificial neural network using floating point weights and activations and implementing floating point calculation, the XNOR implementation using 8-bit weights and activations with the values -1 or 1 and the compressed version, using 1-bit representation of weights and activations. A detailed usage of resources from these implementations is listed in table 6.5.

| Resource | ALUTs | Registers | Logic | DSP Blocks | Memory Blocks |
|---|---|---|---|---|---|
| Multiplication | 8,992 | 10,679 | 38% | 3 | 62 |
| XNOR | 8,316 | 10,236 | 37% | 2 | 60 |
| Compressed | 8,570 | 10,470 | 38% | 2 | 63 |
| Huynh | 63,454 | 44,080 | 92% | 64 | N/A |
| Park-Sung | 121,173 | 130,802 | N/A | 900 | 232 |
| wang-et-al | N/A | 16,356 | 84% | 93 | 65 |

Table 6.4: FPGA Resources allocated compared between different implementations.

Park and Sun [10] implement a network using only 3 bits for storing the networks weights. The network consists of 784-1022-1022-1022-10 neurons, so 3 hidden layers are implemented. 100 images are preloaded in the FPGAs internal memory and are iterated during runtime.

Wang et al. [11] implement a discrete model in FPA hardware only. The images are preprocessed by de-noising, edge amplifying, binarization and removing redundant information. Afterwards the images are fed into a network consisting of 784-500-500-2000-10 neurons, so 3 hidden layers are implemented. The image classified is preloaded into the FPGA's internal memory and there is no mechanism for changing the picture during runtime.

Huynh [9] implements different version of a network, ranging from 784-40-10 to 784-126-126-126-10. For this measurement an implementation of 784-40-40-40-10 is considered. The network uses 16-bit floating point weights and activations.

| **Resource** | **Multiplication** | **XNOR** | **Compressed** | max |
|---|---|---|---|---|
| ALUTs | 8,992 | 8,316 | 8,570 | N/A |
| Registers | 10,679 | 10,236 | 10,470 | N/A |
| Logic | 38% | 37% | 38% | 15,880 |
| I/O Pins | 103 | 103 | 103 | 314 |
| DSP BLOCKS | 3 | 2 | 2 | 84 |
| Memory Bits | 347,360 | 346,240 | 362,880 | 2,764,800 |
| M10K blocks | 62 | 60 | 63 | 270 |
| max fanout | 615 | 606 | 609 | N/A |

Table 6.5: Resources allocated in FPGA for different kernels

### 6.1.4 Training Cost

Table 6.6 is presenting the average training times for different network implementations. Compared are the implementation described in [5], implemented in the Theano framework using the Python programming language and the implementations developed in this thesis, all implemented in C++.

The training was performed on an Intel(R) Core(TM) i5-2520M CPU clocked at 2.50GHz. The measurements were taken after performing 60000 training cycles on 28x28 pixel grey-scale images.

| Resource | 1h layer training | 2h layer training |
|---|---|---|
| Courbariaux | 9682 min | 21547 min |
| Multiply | 1263 min | 8154 min |
| XNOR | 242 min | 843 min |
| Compressed | 256 min | 867 min |

Table 6.6: Training Times for different network implementations.

### 6.1.5 Development Cost

The development cost of a software project can be measured in lines-of-code (LOC) the finished product implements [62, pp. 55-56]. Table 6.7 is listing the average amount of lines-of-code derived from several GitHub projects implementing the MNIST dataset benchmark and from the project described in this thesis.

| Resource | Lines-Of-Code |
|---|---|
| VHDL | 25470 |
| Verilog | 12684 |
| OpenCL(ARM+FPGA | 2026 |
| OpenCL(FPGA-only) | 47 |

Table 6.7: Training Times for different network implementations.

## 6.2 Evaluation of KPI

In this section measurements of key performance indicators listed in the previous section are evaluated.

### 6.2.1 Classification Accuracy

As seen in table 6.1, all models under investigation show state-of-the-art results $> 85\%$ classification accuracy. While conventional neural networks already show results $> 90\%$ with just one hidden layer with only, binarized neural networks drastically improve by involving a second hidden layer, both containing at least 2048 neurons. This can be explained by the fact that conventional networks implement the activation function being able to output all possible values between -1 and 1. Like this it is quite easy to identify a trend the output will follow: The output value will shift more and more towards the correct value and already by seeing a result of 0.2 instead of 1.0, a trend of the outcoming value can already been foreseen and

a lower threshold can be applied. With a lowered threshold, the training time can drastically be shortened compared to training models of the XNOR network.

Compared with other authors implementations, the XNOR implementation presented in this thesis is showing slightly less classification accuracy. While other authors implementations require a training time ranging from several days up to weeks, this model is trained in less than 1 day. Additionally, optimizations implemented in other authors implementations are not implemented in this thesis's work. Those include batch normalization and complex gradient calculation.

### 6.2.2 Execution-Time per Frame

The execution-time per frame is drastically lowered by accelerating the network with the FPGA core compared to an ARM-only implementation. Due to the FPGAs capability of parallelizing tasks, all weights of a neuron can be processed at once, while on ARM all calculations are executed serialized.

While the XNOR implementation is showing a high accelerating potential over the floating-point multiplication on an ARM core implementation, it has almost no impact using OpenCL running on FPGA fabric. Modern FPGA systems contain DSP blocks that implement native floating-point accelerators [73]. Using floating-point accelerators, a floating-point calculation can be processed within one clock cycle which is the same timing a XNOR operation stored in a flipflop has. By compressing the network to only use 1-bit per weight and activation, even more operations can be parallelized. Since OpenCL uses C-style datatypes, 32 weights are stored into a 32-bit integer, instead of one weight per integer.

Thus, the biggest difference can be achieved when accelerating conventional artificial networks with OpenCL running on FPGA fabric.

| **Resource** | **784x4096** | **4096x4096** | **10x4096** | **1h - 4096** | **2h - 4096** |
|---|---|---|---|---|---|
| ARM-mult | 308.066 ms | 1654.475 ms | 4.047 ms | 312.145 ms | 1966.782 ms |
| ARM-xnor | 103.102 ms | 687.858 ms | 1.683 ms | 131.783 ms | 819.191 ms |
| ARM-comp | 144.244 ms | 963.014 ms | 2.641 ms | 147.028 ms | 1110.042 ms |
| SoC-mult | 45.812 ms | 458.964 ms | 2.529 ms | 48.952 ms | 507.305 ms |
| SoC-xnor | 45.315 ms | 439.686 ms | 1.802 ms | 46.952 ms | 486.803 ms |
| SoC-comp | 32.966 ms | 205.096 ms | 1.612 ms | 34.578 ms | 239.674 ms |

Table 6.8: Speed for matrix operations with different kernels

As table 6.8 shows, the processing time on ARM architecture is growing linear with the networks size. It takes 7.8 times longer to process a kernel with 784 neurons, as it

takes processing one with 10 neurons. Using the FPGA architecture, it is possible to parallelize processing, leading to non-linear growth of processing time over network sizes.

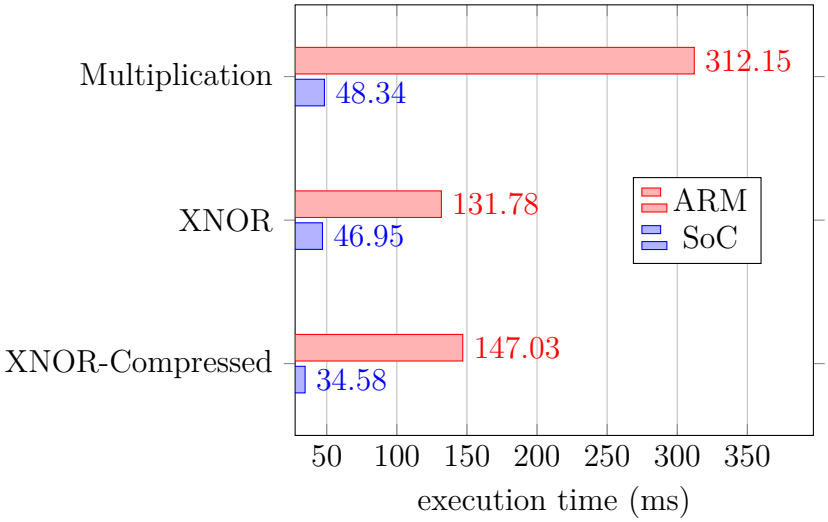



Figure 6.1: SoC kernel timings in comparison to ARM kernel timings for 1 hidden layer network.


Figure 6.2: SoC kernel timings in comparision to ARM kernel timings for 2 hidden layers network.

### 6.2.3 FPGA Logic Gates used

As shown in figure 6.3, using OpenCL to implement the compressed XNOR implementation is drastically saving FPGA resources compared to all other authors implementations. This is mainly due to the fact that the algorithm is only partly executed inside the FPGA. The FPGA does not take care of loading images or iterating through single layers. Only the matrix multiplication is accelerated inside the FPGA which is implemented in a way that the same implementation can be reused for all sizes. A downside of this implementation is that for small kernel sizes, which usually cold be parallelized to execute in just 1 clock cycle, the execution time is slightly higher as expected and thus saving almost no time compared to the ARM implementation.

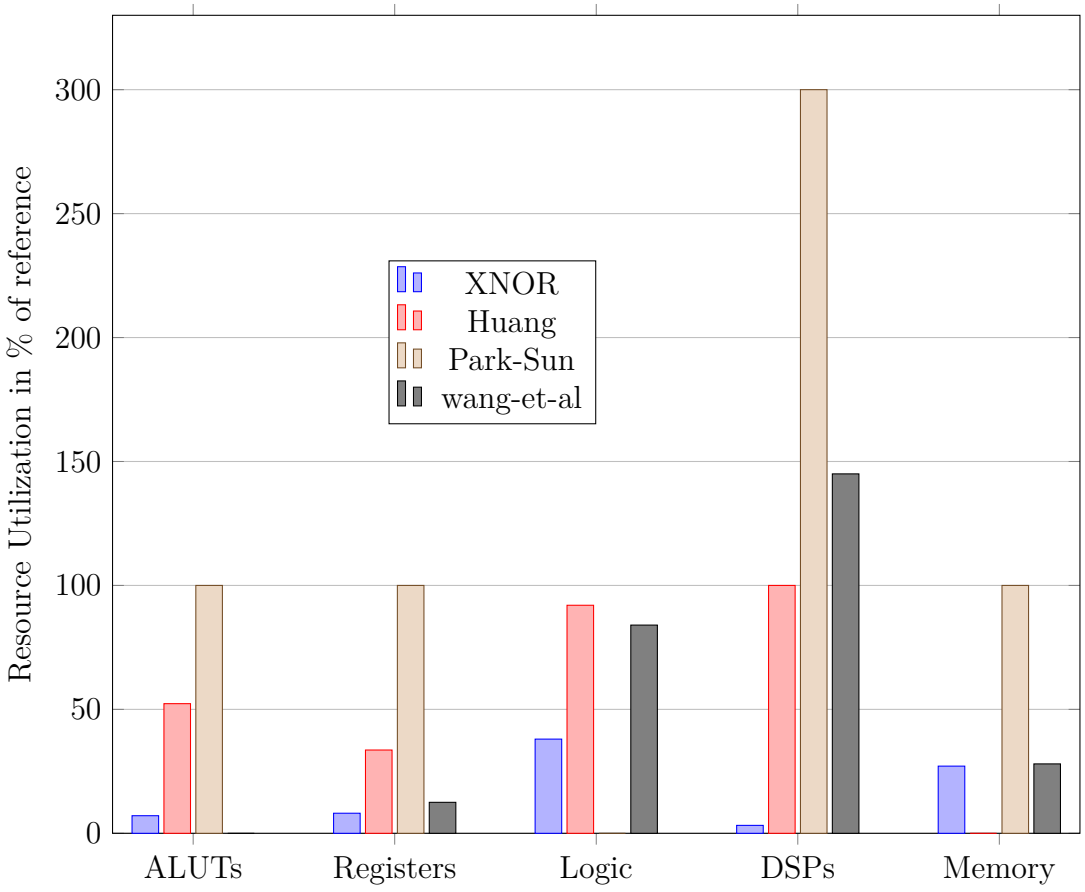



Figure 6.3: Comparison of logic utilization for different implementations.

### 6.2.4 Training Cost

As shown in Table 6.6, a native implementation in C++ is drastically decreasing the training time needed for learning a dataset of 60000 images, compared to an implementation using the Python programming language and the Theano framework. While Courbariaux's implementation needs up to 2 weeks for learning the dataset, the same operation can be achieved in 14 hours using a native C++ implementation.

This can mostly be explained by the way the Theano framework is performing network operations. Theano is internally building a network graph where all operations are an entity fully able to work on its own. The graphs nodes are executed serialized, where for each operation the network is binarized before and de-binarized after the operation. Also, the implementation includes additional steps to optimize the learning, like building a mean over a matrix of weights. This can be visualized by having each node in the graph performing a full loop over the networks neurons, executing just one operation. Together with the (de-)binarization, for each node in the graph there are 3 loops over all networks neurons.

The native C++ implementation is not implemented as a graph, and thus can make use of optimizations like loop unrolling, where multiple operations can be performed in just one loop over the networks neurons.

Also, the usage of the programming language itself is an explanation for the low training-time: Python is an interpreted dynamic language, meaning in its basic form, each line of code is interpreted and executed during runtime, while in C++ the compiler can produce highly optimized code during compile time. The implementation of basic features in the language differ a lot. A loop in Python for example is, when using the *range* implementation, very different from the for loop in C++: First, all elements described in the *range* function are pre-calculated to build an input vector. This vector is then used in a for-each-style loop to execute the operations on it's elements. In C++ at the end of each loop the next elements in calculated, which eliminates the pre-calculation of the input vector. When using a big range of inputs, like it is usually done in neural networks, this pre-caclulation can have a big impact on execution time. Iterating through a simple hidden layer of 4096 neurons is basically resulting in $4096 * 4096 = 16777216$ operations of multiplying weights with inputs.

Another big factor for the savings in training cost the the usage of the training optimization method of exaggerated targets that is drastically reducing the number of training cycles need until a neuron shifts towards its optimum value.

### 6.2.5 Development Cost

The difference in lines-of-code, listed in Table 6.7, can be explained by the choice of programming language and the level of hardware abstraction OpenCL offers. As Table 6.9 shows, the intermediate VHDL and Verilog code, the OpenCL compiler generates for this project has 581494 lines of code. Compared to the 47 lines of code needed for writing the FPGA kernel itself, this is scaled by a factor of approximately 10000. The resulting VHDL or Verilog code contains functionalities for transferring memory from the ARM cores to the FPGA fabric and back and brings state machines for executing the main software's threads, as well as helper threads, executing the data transfer or pre-calculating resources.

| **Language** | **Lines-Of-Code** |
|---|---:|
| VHDL | 430634 |
| Verilog | 150860 |
| Sum | 581494 |

Table 6.9: Resulting lines-of-code in the intermediate code OpenCL generates.

Besides just comparing the lines of code needed for solving the problem, the development cost also includes the complexity of the act of writing the code. Writing VHDL and verilog code forces the developer to bring hardware architectural knowledge, since all operations have to be implemented on a register level. In comparison writing OpenCL code let the developer focus on solving the problem or algorithm on a higher abstraction level, without taking care of FPGA resources. So not only the time needed to write 1 lines of code is considered longer for writing VHDL code, also the salary of a VHDL developer is usually higher, so overall the usage of OpenCL is drastically lowering the development costs.

# 7 Conclusions

This chapter concludes the thesis with a discussion of its findings and contributions. Limitations, as well as outlines will be pointed out and directions for future research will be discussed.

## 7.1 Summary

The achievements of this work show that the implementation of binarized neural networks on FPGA devices is possible in real-life scenarios. It is measurably faster and more resource-friendly as the usage of conventional neural networks without trading the networks precision. In terms of classification accuracy state-of-the-art results could be achieved by using a binarized network without significantly increasing the networks size. Thus, this work confirms the theoretical assumptions and first proofs of concept of Courbariaux et al.

The matured implementation unfolds additional advantages: Binarized neural networks are not simply calculating faster, they also can be implemented with less complexity by using OpenCL and SoCs with integrated FPGA fabric. With the usage of OpenCL the development process could be speed up, resulting in less lines of code, that are easily maintainable, because of C-Style language. Compared to a native FPGA implementation less knowledge of the underlying hardware architecture is needed to write and maintain the algorithm. This leads to decreased development costs compared to an implementation using only hardware description languages. Not only the amount of code is reduced, also the resulting number of logic gates within the FPGA is drastically smaller as with other implementations.

The OpenCL framework allows development of the algorithm and source code on normal hardware like a generic laptop's CPU and after testing verification, the algorithm can be easily ported to any supported hardware architecture by linking against the OpenCL FPGA SDK. Using a SoC as a target platform, that is combining ARM cores with an FPGA core is a good alternative to using ARM-only or FPGA-only implementations. All parallelizable tasks can be outsourced to the FPGA, while still having the convenience of a general-purpose processor and the easy access it offers to lots of peripherals.

The binarized network maps very well on FPGA logic and only needs up to 30 percent of the FPGAs resources. Since modern FPGA fabrics offer DSP blocks implementing IEEE 754 floating point calculation, the biggest improvement over conventional networks is not the reduction to XNOR calculation, but the possibility to compress the networks weights to only a single bit, reducing data transfers and the amount of logic needed to process the network. This is showing big potential for storing and transferring pre-trained networks on small-footprint devices or when sharing network models with cloud providers.

Not only the execution time of the network is drastically lowered by accelerating it with a FPGA. Compared to other authors, this implementation uses less FPGA logic elements and due to OpenCL technology, the development time is drastically lowered, compared to an FPGA-only implementation. The time to train a network is reduced by accelerating the feed forward step inside the FPGA. A big reduction of training times could be achieved by the method of exaggerated targets during training, where a disadvantage of the binarization is compensated: While the outputs of binarized networks can only be true or false without any intermediate values, no cost can be evaluated if the neuron is showing the correct result, which leads to very long training times. The method of exaggerated targets if overcoming this disadvantage by overdrawing the aimed target values, leading to a big reduction of training times and thus, a reduction of overall cost.

While binarized networks tend to grow bigger as conventional networks, the overall computational complexity is reduced, because of easier accessible calculations and compressed weights, resulting in an overall reduced power consumption due to less operations needed to process the network, which also leads to less need of cooling the systems and thus, they are a perfect fit for usage in big data-centers.

During implementation it was noticed that without any tweaks to the learning algorithm, binarized networks do not perform very well: Learning rate scaling is very important. Binarized networks need to start with a big learning rate. Otherwise it takes too much training cycles to reach an acceptable result. Also, Glorot scaling of the initialization parameters is very important. Like this, certain neurons have bigger impact on the output value and good results can be achieved faster. By exaggerating the networks error, correct results can still lead to proper learning and thus decrease training time.

## 7.2 Future Work

This work could prove the feasibility of implementing binarized neural networks using OpenCL on FPGA fabric.

Outsourcing other steps performed within the backpropagation algorithm could improve training times drastically. Steps within the backpropagation algorithm are very similar to the dot product accelerated in this work. They are implemented as multiplications of matrices and vectors. and thus, could easily be ported to be executed within the FPGA core.

This thesis did not examine the problem of memory bandwidth between ARM core and FPGA. This should also be addressed by further research. The full network initialization data could be placed in memory cells within the FPGA to achieve higher bandwidth and avoiding redundant transfers of the networks weights.

While this work is using dynamic kernels that can be reused with any network size, the algorithm could be even further parallelized by implementing static kernels for specific kernel sizes. Multiple kernels implementing memory pipelines could be used to transfer ones kernels result to the next kernels which can use them as input parameters. This would even lower processing times further and would avoid unneeded memory transfers between ARM- and FPGA core.

The next iteration step should implement convolutional neural networks using this technique, which would enable classification of sub-images inside a big image that can be classified in any angle. Also, the image source should not be a static picture that is fed from ARM core to the FPGA, but rather be a live image from a camera source, directly fetched by the FPGA. By training the network with the CIFAR-100 dataset, classes of real-life objects can be classified instead of handwritten numbers. Implementing multiple kernels in parallel would enable a real-time classification of all detectable objects within a camera stream.

Attaching the FPGA driven convolutional network to a cloud service, images of unidentified object can be automatically sent to the cloud provider and be identified by other ways, for example by humans. These new datasets can then be used to train an updated model of the network which is sent back to the processing system. Like this the network could easily adapt to changes in the environment.

# Bibliography


[1] *Intel FPGA SDK for OpenCL Pro Edition - Programming Guide*, Intel.

[2] P. Sermanet and Y. LeCun, "Traffic sign recognition with multi-scale convolutional networks," in *IJCNN*. IEEE, 2011, pp. 2809–2813.

[3] L. Zhang, X. Wu, and D. Luo, "Real-time activity recognition on smartphones using deep neural networks," in *2015 IEEE 12th Intl Conf on Ubiquitous Intelligence and Computing and 2015 IEEE 12th Intl Conf on Autonomic and Trusted Computing and 2015 IEEE 15th Intl Conf on Scalable Computing and Communications and Its Associated Workshops (UIC-ATC-ScalCom)*, Aug 2015, pp. 1236–1242.

[4] A. Sehgal and N. Kehtarnavaz, "A convolutional neural network smartphone app for real-time voice activity detection," *IEEE Access*, vol. 6, pp. 9017–9026, 2018.

[5] M. Courbariaux and Y. Bengio, "Binarynet: Training deep neural networks with weights and activations constrained to +1 or -1," vol. abs/1602.02830, 2016. [Online]. Available: http://arxiv.org/abs/1602.02830

[6] M. Rastegari, V. Ordonez, J. Redmon, and A. Farhadi, "Xnor-net: Imagenet classification using binary convolutional neural networks," *Computer Vision – ECCV 2016*, vol. 9908, pp. 525–542, 10 2016.

[7] M. Kim and P. Smaragdis, "Bitwise neural networks," *International Conference on Machine Learning (ICML) Workshop on Resource-Efficient Machine Learning*, 2015.

[8] G. Cohen, S. Afshar, J. Tapson, and A. van Schaik, "Emnist: Extending mnist to handwritten letters," in *2017 International Joint Conference on Neural Networks (IJCNN)*, May 2017, pp. 2921–2926.

[9] T. V. Huynh, "Deep neural network accelerator based on FPGA," in *2017 4th NAFOSTED Conference on Information and Computer Science*, Nov 2017, pp. 254–257.

[10] J. Park and W. Sung, "FPGA based implementation of deep neural networks using on-chip memory only," in *2016 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, March 2016, pp. 1011–1015.

[11] Wang, Yang, Xu, and Fu, "Design of FPGA-based handwriting image recognition system," in *International Information and Engineering Technology Association, Advances In Modelling and Analysis B*, vol. 60, no. 2, 2017, pp. 493–504.

[12] L. Deng, "The mnist database of handwritten digit images for machine learning research [best of the web]," *IEEE Signal Processing Magazine*, vol. 29, no. 6, pp. 141–142, Nov 2012.

[13] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, A. C. Berg, and L. Fei-Fei, "Imagenet large scale visual recognition challenge," *Int. J. Comput. Vision*, vol. 115, no. 3, pp. 211–252, Dec. 2015. [Online]. Available: http://dx.doi.org/10.1007/s11263-015-0816-y

[14] D. G. Lowe, "Distinctive image features from scale-invariant keypoints," *Int. J. Comput. Vision*, vol. 60, no. 2, pp. 91–110, Nov. 2004. [Online]. Available: https://doi.org/10.1023/B:VISI.0000029664.99615.94

[15] R. T. Ionescu and M. Popescu, *Knowledge Transfer between Computer Vision and Text Mining*, 01 2016.

[16] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in Neural Information Processing Systems 25*, F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, Eds. Curran Associates, Inc., 2012, pp. 1097–1105. [Online]. Available: http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf

[17] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," in *Proceeding of the International Conference on Learning Representations (ICLR)*, 2015, pp. 1–14.

[18] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going deeper with convolutions," in *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2015, pp. 1–9.

[19] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2016, pp. 770–778.

[20] L. A. Gatys, A. S. Ecker, and M. Bethge, "Image style transfer using convolutional neural networks," in *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2016, pp. 2414–2423.

[21] E. Schikuta and E. Mann, "N2sky — neural networks as services in the clouds," in *The 2013 International Joint Conference on Neural Networks (IJCNN)*, Aug 2013, pp. 1–8.

[22] S. Anwar, K. Hwang, and W. Sung, "Structured pruning of deep convolutional neural networks," *J. Emerg. Technol. Comput. Syst.*, vol. 13, no. 3, pp. 32:1–32:18, Feb. 2017. [Online]. Available: http://doi.acm.org/10.1145/3005348

[23] P. Molchanov, S. Tyree, T. Karras, T. Aila, and J. Kautz, "Pruning convolutional neural networks for resource efficient transfer learning," in *Proceeding of the International Conference on Learning Representations (ICLR)*, 2017.

[24] Y. L. Cun, J. S. Denker, and S. A. Solla, "Advances in neural information processing systems 2," D. S. Touretzky, Ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1990, ch. Optimal Brain Damage, pp. 598–605. [Online]. Available: http://dl.acm.org/citation.cfm?id=109230.109298

[25] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam, "Mobilenets: Efficient convolutional neural networks for mobile vision applications," *CoRR*, vol. abs/1704.04861, 2017.

[26] F. Chollet, "Xception: Deep learning with depthwise separable convolutions," in *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, July 2017, pp. 1800–1807.

[27] X. Zhang, X. Zhou, M. Lin, and J. Sun, "Shufflenet: An extremely efficient convolutional neural network for mobile devices," in *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2018.

[28] M. Courbariaux, Y. Bengio, and J. David, "Low precision arithmetic for deep learning," *CoRR*, vol. abs/1412.7024, 2014. [Online]. Available: http://arxiv.org/abs/1412.7024

[29] K. Hazelwood, S. Bird, D. Brooks, S. Chintala, U. Diril, D. Dzhulgakov, M. Fawzy, B. Jia, Y. Jia, A. Kalro, J. Law, K. Lee, J. Lu, P. Noordhuis, M. Smelyanskiy, L. Xiong, and X. Wang, "Applied machine learning at facebook: A datacenter infrastructure perspective," in *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, Feb 2018, pp. 620–629.

[30] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, R. Boyle, P.-l. Cantin, C. Chao, C. Clark, J. Coriell, M. Daley, M. Dau, J. Dean, B. Gelb, T. V. Ghaemmaghami, R. Gottipati, W. Gulland, R. Hagmann, C. R. Ho, D. Hogberg, J. Hu, R. Hundt, D. Hurt, J. Ibarz, A. Jaffey, A. Jaworski, A. Kaplan, H. Khaitan, D. Killebrew, A. Koch, N. Kumar, S. Lacy, J. Laudon, J. Law, D. Le, C. Leary, Z. Liu, K. Lucke, A. Lundin, G. MacKean, A. Maggiore, M. Mahony, K. Miller, R. Nagarajan, R. Narayanaswami, R. Ni, K. Nix, T. Norrie, M. Omernick, N. Penukonda, A. Phelps, J. Ross, M. Ross, A. Salek, E. Samadiani, C. Severn, G. Sizikov, M. Snelham, J. Souter, D. Steinberg, A. Swing, M. Tan, G. Thorson, B. Tian, H. Toma, E. Tuttle, V. Vasudevan, R. Walter, W. Wang, E. Wilcox, and D. H. Yoon, "In-datacenter performance analysis of a tensor processing unit," in *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ser. ISCA '17. New York, NY, USA: ACM, 2017, pp. 1–12. [Online]. Available: http://doi.acm.org/10.1145/3079856.3080246

[31] A. M. Caulfield, E. S. Chung, A. Putnam, H. Angepat, J. Fowers, M. Haselman, S. Heil, M. Humphrey, P. Kaur, J. Y. Kim, D. Lo, T. Massengill, K. Ovtcharov, M. Papamichael, L. Woods, S. Lanka, D. Chiou, and D. Burger, "A cloud-scale acceleration architecture," in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, Oct 2016, pp. 1–13.

[32] P. Sundararajan and C. Kulkarni, "Data store acceleration-as-a-service on amazon fpga instances," 2017. [Online]. Available: https://www.xilinx.com/support/documentation/product-briefs/reniac-aws-f1.pdf

[33] A. Putnam, A. Caulfield, E. Chung, D. Chiou, K. Constantinides, J. Demme, H. Esmaeilzadeh, J. Fowers, J. Gray, M. Haselman, S. Hauck, S. Heil, A. Hormati, J.-Y. Kim, S. Lanka, E. Peterson, A. Smith, J. Thong, P. Y. Xiao, D. Burger, J. Larus, G. P. Gopal, and S. Pope, "A reconfigurable fabric for accelerating large-scale datacenter services," in *Proceeding of the 41st Annual International Symposium on Computer Architecture (ISCA)*. IEEE Press, June 2014, pp. 13–24. [Online]. Available: https://www.microsoft.com/en-us/research/publication/a-reconfigurable-fabric-for-accelerating-large-scale-datacenter-services/

[34] F. Rosenblatt, *Principles of neurodynamics: perceptrons and the theory of brain mechanisms*, ser. Report (Cornell Aeronautical Laboratory). Spartan Books, 1962.

[35] R. Rojas, *Neural Networks: A Systematic Introduction*. Berlin, Heidelberg: Springer-Verlag, 1996.

[36] J. Raju, S. Kumar, and L. Sneha, "Realization of logic gates using mcculloch-pitts neuron model," in *International Journal of Engineering Trends and Technology*, vol. 45, 03 2017, pp. 52–56.

[37] X. Li and X. Wu, "Constructing long short-term memory based deep recurrent neural networks for large vocabulary speech recognition," in *2015 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, April 2015, pp. 4520–4524.

[38] R. Pascanu, T. Mikolov, and Y. Bengio, "On the difficulty of training recurrent neural networks," in *Proceedings of the 30th International Conference on International Conference on Machine Learning - Volume 28*, ser. ICML'13. JMLR.org, 2013, pp. III–1310–III–1318. [Online]. Available: http://dl.acm.org/citation.cfm?id=3042817.3043083

[39] K. Hornik, "Approximation capabilities of multilayer feedforward networks," *Neural Netw.*, vol. 4, no. 2, pp. 251–257, Mar. 1991. [Online]. Available: http://dx.doi.org/10.1016/0893-6080(91)90009-T

[40] T. Masters, *Practical Neural Network Recipes in C++*. San Diego, CA, USA: Academic Press Professional, Inc., 1993.

[41] G. E. Hinton, S. Osindero, and Y.-W. Teh, "A fast learning algorithm for deep belief nets," *Neural Comput.*, vol. 18, no. 7, pp. 1527–1554, jul 2006. [Online]. Available: http://dx.doi.org/10.1162/neco.2006.18.7.1527

[42] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016, http://www.deeplearningbook.org.

[43] K. Fukushima and S. Miyake, "Neocognitron: A self-organizing neural network model for a mechanism of visual pattern recognition," in *Competition and Cooperation in Neural Nets*, S.-i. Amari and M. A. Arbib, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 1982, pp. 267–285.

[44] W. Zhang, K. Itoh, J. Tanida, and Y. Ichioka, "Parallel distributed processing model with local space-invariant interconnections and its optical architecture," *Appl. Opt.*, vol. 29, no. 32, pp. 4790–4797, Nov 1990. [Online]. Available: http://ao.osa.org/abstract.cfm?URI=ao-29-32-4790

[45] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, Nov 1998.

[46] S. Albelwi and A. Mahmood, "A framework for designing the architectures of deep convolutional neural networks," *Entropy*, vol. 19, no. 6, 2017. [Online]. Available: http://www.mdpi.com/1099-4300/19/6/242

[47] L. Bottou and O. Bousquet, "The tradeoffs of large scale learning," in *Advances in Neural Information Processing Systems*, J. Platt, D. Koller, Y. Singer, and S. Roweis, Eds. NIPS Foundation (http://books.nips.cc), 2008, vol. 20, pp. 161–168. [Online]. Available: http://leon.bottou.org/papers/bottou-bousquet-2008

[48] Y. LeCun, L. Bottou, G. B. Orr, and K.-R. Müller, "Efficient backprop," in *Neural Networks: Tricks of the Trade, This Book is an Outgrowth of a 1996 NIPS Workshop*. London, UK, UK: Springer-Verlag, 1998, pp. 9–50. [Online]. Available: http://dl.acm.org/citation.cfm?id=645754.668382

[49] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Neurocomputing: Foundations of research," J. A. Anderson and E. Rosenfeld, Eds. Cambridge, MA, USA: MIT Press, 1988, ch. Learning Representations by Back-propagating Errors, pp. 696–699. [Online]. Available: http://dl.acm.org/citation.cfm?id=65669.104451

[50] M. Courbariaux, Y. Bengio, and J.-P. David, "Binaryconnect: Training deep neural networks with binary weights during propagations," in *Advances in Neural Information Processing Systems 28*, C. Cortes, N. D. Lawrence, D. D. Lee, M. Sugiyama, and R. Garnett, Eds. Curran Associates, Inc., 2015, pp. 3123–3131. [Online]. Available: http://papers.nips.cc/paper/5647-binaryconnect-training-deep-neural-networks-with-binary-weights-during-propagations.pdf

[51] *CUDA API REFERENCE MANUAL*, NVIDIA Corporation, 4 2012, version 4.2.

[52] C. Grozea, Z. Bankovic, and P. Laskov, "Facing the multicore-challenge," R. Keller, D. Kramer, and J.-P. Weiss, Eds. Berlin, Heidelberg: Springer-Verlag, 2010, ch. FPGA vs. Multi-core CPUs vs. GPUs: Hands-on Experience with a Sorting Application, pp. 105–117. [Online]. Available: http://dl.acm.org/citation.cfm?id=1980597.1980612

[53] *Intel FPGA SDK for OpenCL Pro Edition - Best Practices Guide*, Intel.

[54] D. A. Patterson and J. L. Hennessy, *Computer Organization and Design, Fifth Edition: The Hardware/Software Interface*, 5th ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2013.

[55] S. Asano, T. Maruyama, and Y. Yamaguchi, "Performance comparison of fpga, gpu and cpu in image processing," in *FPL 09: 19th International Conference on Field Programmable Logic and Applications*, 08 2009, pp. 126–131.

[56] *CUDA RUNTIME API*, NVIDIA Corporation, 7 2017, vRelease Version.

[57] B. Gaster, L. Howes, D. R. Kaeli, P. Mistry, and D. Schaa, *Heterogeneous Computing with OpenCL*, 1st ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2011.

[58] *The OpenCL Specification Version 1.2*, Khronos OpenCL Working Group, 11 2012, rev. 19.

[59] A. Munshi, B. Gaster, T. G. Mattson, J. Fung, and D. Ginsburg, *OpenCL Programming Guide*, 1st ed. Addison-Wesley Professional, 2011.

[60] R. Banger, B. Bhattacharyya, and K. Bhattacharyya, *OpenCL Programming by Example*, ser. Community experience distilled. Packt Publishing, 2013. [Online]. Available: https://books.google.de/books?id=1O80ngEACAAJ

[61] *OpenCL on FPGAs for GPU Programmers*, Accelware, 6 2014, version 1.0.

[62] S. McConnell, *Software Estimation - Demystifying the Black Art*. Microsoft Press, 2006.

[63] E. Nurvitadhi, G. Venkatesh, J. Sim, D. Marr, R. Huang, J. Ong Gee Hock, Y. T. Liew, K. Srivatsan, D. Moss, S. Subhaschandra, and G. Boudoukh, "Can fpgas beat gpus in accelerating next-generation deep neural networks?" in *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '17. New York, NY, USA: ACM, 2017, pp. 5–14. [Online]. Available: http://doi.acm.org/10.1145/3020078.3021740

[64] E. Nurvitadhi, D. Sheffield, J. Sim, A. Mishra, G. Venkatesh, and D. Marr, "Accelerating binarized neural networks: Comparison of fpga, cpu, gpu, and asic," in *2016 International Conference on Field-Programmable Technology (FPT)*, Dec 2016, pp. 77–84.

[65] *Cyclone V Reference Manual*, Intel, 6 2018, version 2.

[66] D. Bailey, *Design for Embedded Image Processing on FPGAs*, 01 2011.

[67] X. Glorot and Y. Bengio, "Understanding the difficulty of training deep feedforward neural networks," in *In Proceedings of the International Conference on Artificial Intelligence and Statistics (AISTATS'10). Society for Artificial Intelligence and Statistics*, 2010.

[68] Y. Bengio, N. Boulanger-Lewandowski, and R. Pascanu, "Advances in optimizing recurrent networks," in *2013 IEEE International Conference on Acoustics, Speech and Signal Processing*, May 2013, pp. 8624–8628.

[69] L. N. Smith, "Cyclical learning rates for training neural networks," in *2017 IEEE Winter Conference on Applications of Computer Vision (WACV)*, March 2017, pp. 464–472.

[70] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: A simple way to prevent neural networks from overfitting," *Journal of Machine Learning Research*, vol. 15, pp. 1929–1958, 2014. [Online]. Available: http://jmlr.org/papers/v15/srivastava14a.html

[71] Y. Bengio, "Practical recommendations for gradient-based training of deep architectures," in *Neural Networks: Tricks of the Trade*. Springer,Berlin,Heidelberg, 2012, pp. 437–478.

[72] D. Hebb, *The Organization of Behavior: A Neuropsychological Theory*. Taylor & Francis, 1949.

[73] *Cyclone V Device Datasheet*, Intel, 5 2018.