TAL TECH

DOCTORAL THESIS

Novel Neural Network Accelerator Architectures for FPGAs

Madis Kerner

TALLINNA TEHNIKAÜLIKOOL
TALLINN UNIVERSITY OF TECHNOLOGY
TALLINN 2024

TALLINN UNIVERSITY OF TECHNOLOGY
DOCTORAL THESIS
16/2024

Novel Neural Network Accelerator Architectures for FPGAs

MADIS KERNER

TAL TECH PRESS

TALLINN UNIVERSITY OF TECHNOLOGY
School of Information Technologies
Department of Computer Systems

The dissertation was accepted for the defence of the degree of Doctor of Philosophy on 27 March 2024

Supervisor: Prof. Dr. Jaan Raik,
Department of Computer Systems, School of Information Technologies,
Tallinn University of Technology,
Tallinn, Estonia

Co-supervisor: Assoc. Prof. Dr. Kalle Tammemäe,
IT College, School of Information Technologies,
Tallinn University of Technology,
Tallinn, Estonia

Co-supervisor: Prof. Dr.-Ing. Thomas Hollstein,
Research Center Future Aging,
Frankfurt University of Applied Sciences,
Frankfurt, Germany

Opponents: Prof. Dr. Alberto Bosio,
Institute of Nanotechnology Ecully,
Ecully, France

Prof. Dr. Jari Nurmi,
Faculty of Information Technology and Communication Sciences,
Tampere University,
Tampere, Finland

Defence of the thesis: 10 April 2024, Tallinn

Declaration:
Hereby I declare that this doctoral thesis, my original investigation and achievement, submitted for the doctoral degree at Tallinn University of Technology, has not been submitted for any academic degree elsewhere.

Madis Kerner

signature

Copyright: Madis Kerner, 2024
ISSN 2585-6898 (publication)
ISBN 978-9916-80-130-7 (publication)
ISSN 2585-6901 (PDF)
ISBN 978-9916-80-131-4 (PDF)
DOI https://doi.org/10.23658/taltech.16/2024
Printed by Koopia Niini & Rauam

Kerner, M. (2024). *Novel Neural Network Accelerator Architectures for FPGAs* [TalTech Press]. https://doi.org/10.23658/taltech.16/2024

TALLINNA TEHNIKAÜLIKOOL
DOKTORITÖÖ
16/2024

Uudsed närvivõrkude kiirendite arhitektuurid FPGAdele

MADIS KERNER

TAL TECH
KIRJASTUS

Contents

List of Publications

The present Ph.D. thesis is based on the following publications that are referred to in the text by Roman numbers.

I M. Kerner, K. Tammemae, J. Raik, and T. Hollstein, “An Efficient FPGA-based Architecture for Contractive Autoencoders,” in 2020 IEEE 28th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM), pp. 230–230, Institute of Electrical and Electronics Engineers Inc., 5 2020

II M. Kerner, K. Tammemae, J. Raik, and T. Hollstein, “Novel Architectures for Contractive Autoencoders with Embedded Learning,” in 2020 17th Biennial Baltic Electronics Conference (BEC), vol. 2020-October, pp. 1–6, IEEE Computer Society, 10 2020

III M. Kerner, K. Tammemae, J. Raik, and T. Hollstein, “Triple Fixed-Point MAC Unit for Deep Learning,” in 2021 Design, Automation & Test in Europe Conference & Exhibition (DATE), vol. 2021-February, pp. 1404-1407, Institute of Electrical and Electronics Engineers Inc., 2 2021

Author's Contributions to the Publications

I In I, I was the main author, wrote the Verilog HDL, conducted simulations, prepared the figures, wrote the manuscript, and presented the work.

II In II, I was the main author, wrote the Verilog HDL, conducted simulations, prepared the figures, wrote the manuscript, and presented the work.

III In III, I was the main author, wrote the Verilog HDL, conducted MATLAB and HDL simulations, prepared the figures, wrote the manuscript, and presented the work.

Abbreviations

AE	Autoencoder
ANN	Artificial Neural Network
ASIC	Application Specific Integrated Circuit
BFP	Block Floating-Point
BL	baseline
CAE	Contractive Autoencoder
CCom	CAE with efficient Communication
CCom-RO	Resource Optimised CAE with efficient Communication
CNN	Convolutional Neural Network
CPU	Central Processing Unit
DFxP	Dual Fixed-Point
DL	Deep Learning
DNN	Deep Neural Network
DSP	Digital Signal Processing
ECG	Electrocardiogram
EEG	Electroencephalogram
FP	Floating-Point
FPGA	Field Programmable Gate Array
FSM	Finite State Machine
FxP	Fixed-Point
GPS	Global Positioning System
GPU	Graphical Processing Unit
HDL	Hardware Description Language
HW	Hardware
IOU	Intersection over Union
IP	Intellectual Property
LSTM	Long short-term memory
LUT	Look Up Table
MAC	Multiply-Accumulate
mAP	Mean Average Precision
MSE	Mean Squared Error
NN	Neural Network
PE	Processing Element
PL	Programmable Logic
RAM	Random Access Memory
ReLU	Rectified Linear Unit
SoC	System On Chip
SVM	Support Vector Machine
SW	Software
TFxP	Triple Fixed-Point

1 Introduction

The Deep Learning (DL) systems have made their way to different domains nowadays and enhance the contemporary information age in many ways. Examples of more or less successful deployments of DL can be found in almost any field, including self-driving cars [1, 2], monitoring the operation of the human heart [3], classifying the human activity [4], including the activity related to sport [5], and monitoring the industrial machinery [6] to mention few. Not only that, the different algorithms and methods are so widespread that often the user of the system is not even aware of the existence of a DL network excelling in the background. Due to its increasing popularity, it is no longer explicitly mentioned or advertised but considered the normality.

While the increasing number of successful deployments can give the impression that DL networks and algorithms are relatively new inventions, it is not so: the earliest multi-layer network was published quite a while ago, in 1965: the network described in [7] can be considered the first of its kind. However, the authors in the cited work did not use backpropagation to train the network. Instead, they trained the network layer-by-layer using least-squares fitting.

Speaking of Convolutional Neural Networks (CNNs), this type of network is a familiar invention, too, and can be dated back to 1979: authors in [8] published a network with convolutional and pooling layers similar to the ones deployed nowadays. However, they did not use backpropagation for training but relied on reinforcement learning: they increased the weight values of neuron connections firing together on different layers. Figure 1 presents the network as was initially proposed by the authors: the layered structure and feature extraction scheme resembles that of the contemporary CNNs.

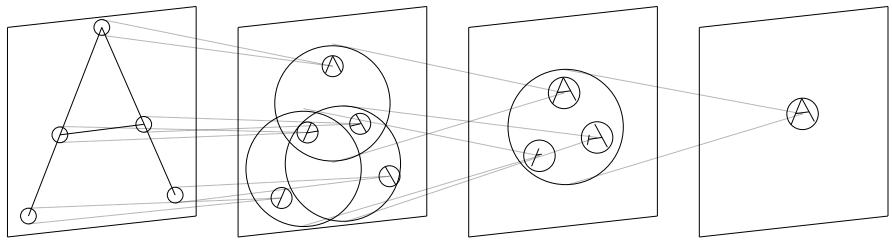


Figure 1 – Architecture of the CNN like network, published in 1979 ([8]). The layered structure and feature extraction scheme are similar to what is used in contemporary algorithms.

However, the backpropagation methods also have a long history and date back to the 1960s. The work published in [9] is the first implementation similar to the training scheme used in contemporary networks, although the author did not mention the neural networks as the target application for the provided method.

The first implementation of backpropagation for training the DL network is presented in [10], where the network was trained to recognize hand-written digits. The system also got successfully deployed to read the hand-written checks.

Despite the existence of methods and successful applications like [10], the DL gained little popularity. Instead, the Support Vector Machine (SVM) introduced in [11] enjoined the attention of researchers. Figure 2 presents a simple separable problem in two-dimensional space as was presented by the authors.

The true advent of DL still had to wait for its opportunity, and it arrived with increased computational power in personal computers. Moreover, the appearance of Graphical Processing Units (GPUs) played an even more prominent role here. With the more reasonable

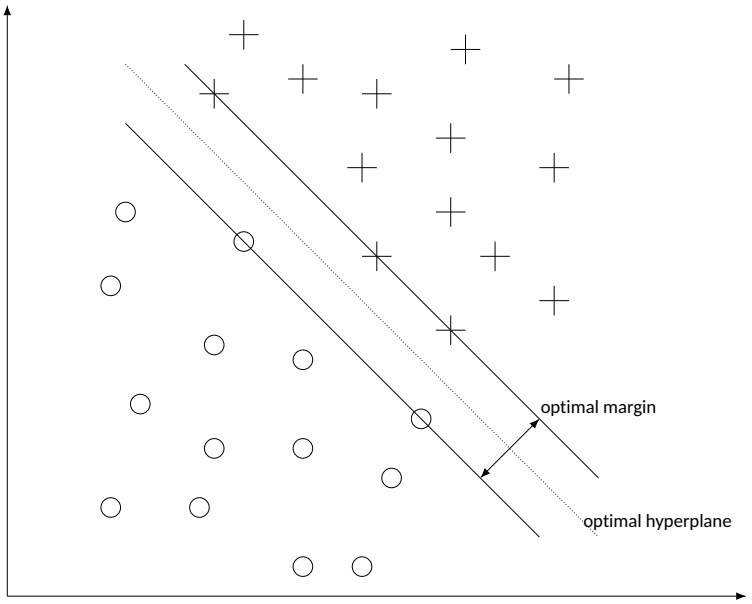


Figure 2 – An example of a separable problem in a 2-dimensional space, [11].

training times and suitable platforms, it turned out that DL models can achieve better results than other algorithms if trained enough, and this becomes possible using Hardware (HW) like GPUs with enough computational power.

1.1 Motivation

Coming the long way, applications of DL networks are really ubiquitous nowadays. Therefore, researchers also address issues running these algorithms on battery-powered resource-constraint systems to expand the usage even more. Naturally, the interest in running these algorithms on less powerful devices follows successful deployments on more powerful HW: there are no arguments about the usefulness of DL.

Training methods, or the loss function of the network, can roughly be divided into supervised- or unsupervised ones. In supervised learning, there is a need for labeled data, and the network output is analyzed based on that, i.e., the loss function is constructed using the actual network response to the input and labels on that specific input. These kinds of networks require pre-training, and due to the need for labeled data, it has to be carried out before deployment; therefore, there are no restrictions to the HW for the training phase. For example, training the supervised DL networks can freely be carried out using GPUs.

Unsupervised DL networks can also be pre-trained using GPUs. However, it has to be noted that these kinds of networks do not rely on labeled data. Therefore, the training process can autonomously carry on during the entire lifespan of a system. An excellent scenario example requiring constant network training is provided in [12]. Authors use the pre-trained DL network to detect analog trojans in the manufactured integrated circuits. They claim that there might be a need to change the network weight values based on the environmental noise level. This need could be handled by letting the system train itself and autonomously handle the drifts of normality. Therefore, accelerators, which also address the training phase of the network, are needed to run unsupervised Deep Neural Networks

(DNNs) on resource-constraint systems.

Regarding resource-constrained systems, the majority of research focuses on executing the pre-trained DL models on these platforms. So, naturally, there are issues to tackle: storage of the network weights and other parameters, levels of possible parallelisms, and replacing the floating point data representations with something less HW hungry variants, to mention a few. However, it has to be emphasized that the focus is precisely on running pre-trained networks, i.e., on forward pass, or inference phase, and the network training has to be performed on more powerful HW. This is the field this thesis focuses on: enable the HW based training of unsupervised networks and provide a suitable data-type to replace the floating point representations.

Elaborating on data-types, the common practice in resource-constraint systems is to binarize the network hyperparameters or rely on fixed-point data representations. However, both of these methods require retraining of the network: existing floating-point based already trained DL implementations can not be directly converted to resource-constraint targets. Therefore, the study of possible data representation that can directly replace the floating point values is needed.

1.2 Problem Formulation

Field Programmable Gate Arrays (FPGAs) have gained a lot of attention among researchers to accelerate DL networks ([13, 14]). This is caused by a relatively rapid development cycle compared to the specialized integrated circuits and relatively low power consumption compared to the GPUs. Also, FPGA is a classical and proven approach for rapid prototyping circuits.

First, this thesis focuses on executing an unsupervised DL network Contractive Autoencoders (CAEs), a flavor of an Autoencoder (AE) with an additional regularization term, on FPGA. The provided implementations also address the HW based training. Also, as previously noted, the floating point data type is very HW hungry. Therefore, it has to be replaced, and a search for a suitable replacement is required.

The research questions being investigated in this thesis are:

1. If unsupervised neural networks are implemented in hardware, they also require the hardware-based training process. Using contractive autoencoder as an example, can the architecture be efficiently implemented in hardware, including hardware-based training?

2. Hardware implementations of any artificial neural network need efficient processing elements for the network nodes. How must the generic, high-precision, and hardware-efficient processing node be designed, and is the floating point data type required, or can it be replaced by a more hardware-conservative counterpart?

1.3 Contribution

This thesis contributes to finding solutions for FPGA based HW accelerators and addresses accelerating the learning phase as well.

First, chapter 2 presents the HW architectures of the CAE Artificial Neural Network (ANN). Although implementations of FPGA based accelerators for ANNs are available in the literature, the presented architectures in this thesis are novel because they embed the HW based learning in CAE for the first time.

Two conference publications back up the presented proposals: the first paper was published in the International Symposium on Field-Programmable Custom Computing Ma-

chines, FCCM, in 2020, [15]. Further, the second extended publication about the FPGA based CAE architectures with embedded learning was published in Baltic Electronics Conference, BEC, also in 2020, [16].

Secondly, the thesis takes a broader look and proposes Multiply-Accumulate (MAC) architecture, which uses triple-fixed-point datatype. Testing results show that the proposed architecture can replace the floating-point-based calculations without retraining the YOLOv2 CNN [17]. This contribution shows that using triple-fixed-point data enables running DNNs on embedded platforms like FPGAs even while using the same network hyperparameter and weight values as in the case of GPU based counterparts excelling with floating point. Also, this thesis proposes to compare the network's actual output while comparing the accuracy using different data types: the network's output should stay the same. Comparing the inference accuracy can yield wrong conclusions: changing the data type can introduce effects hiding issues related to the network training, like overfitting.

The contribution about using triple-fixed-point based MAC is backed up by a conference paper published in Design, Automation and Test in Europe, DATE in 2021, [18].

Summarizing, the main scientific contributions of this thesis beyond the state-of-the-art are:

- First contractive autoencoder implementation in hardware with hardware-based learning.
- Novel Triple Fixed-Point (TFxP) based MAC unit suitable for various neural network architectures, having high numerical precision (comparable to floating point) and very hardware-efficient implementation. This architecture can be directly used in ANN networks (e.g., CNN), which have been trained in software as hardware implementation, without retraining the network. The results are identical to using float data types but do not require implementing floating point support in HW.

1.4 Thesis Organization

The rest of this thesis is organized as follows.

Chapter 2 is devoted to the CAE. Section 2.1 provides an introducion to the AEs in general, followed by section 2.2 describing specifics of CAE. The literature review describing the state-of-the-art is presented in section 2.3. Section 2.4 and its subsections provide the mathematical background of the CAE network, including forward pass, loss function, gradient descent, and equations for updating the weight and bias values during the training phase. Section 2.5 provides the details of the implementation. First, the theoretical equations are optimized in subsection 2.5.1, followed by theoretical estimations for the execution times in subsection 2.5.2. Subsections 2.5.3 to 2.5.5 descibe the three proposed architectures. The usage of HW resources is provided in table 6, and the performance of the provided implementation is compared in subsection 2.5.7. In subsection 2.5.8, the MNIST database of handwritten digits is used for training and evaluation purposes to prove the functionality of the provided solutions. Section 2.6 concludes the chapter 2.

Further, chapter 3 presents the study performed on MAC unit suitable for DNN implementations in general. Section 3.1 provides an introduction to the MAC unit as a DNN building block, and a literature review of existing solutions is provided in section 3.2. Further, section 3.3 discusses possible candidates for the data type for the MAC unit realization, followed by the description of the MATLAB simulation environment and simulation results using the selected type in section 3.4. Section 3.5 describes the Hardware Description Language (HDL) design of the MAC unit, including details about the HW resources

provided in subsection 3.5.3. The second part of the thesis is concluded in section 3.6.

Overall conclusions of the thesis and guidelines for possible future work are provided in chapter 4.

2 Efficient Hardware Architecture for Contractive Autoencoders

2.1 Introduction

This chapter introduces the AEs before going into the theory and following proposals.

Autoencoders are a kind of ANN that reconstruct the network input to its output. Although this might not sound useful, the real benefit of such a network is related to its middle layer.

Figure 3 present a typical AE network. As can be seen, the network consists of two main portions: the encoder part following the input and the decoder part following the middle layer. As the name suggests, the encoder's role is to *encode* the input signal and represent it in the middle layer. Then, the following decoder portion uses this representation to reconstruct the network input to its output. Also, the number of layers used in such a network may vary, but the principal idea remains the same.

One side comment or explanation about the coloring scheme used in figure 3: the same colors will be used throughout the entire chapter 2. Green will be used for the middle layer features, and blue denotes the external layers.

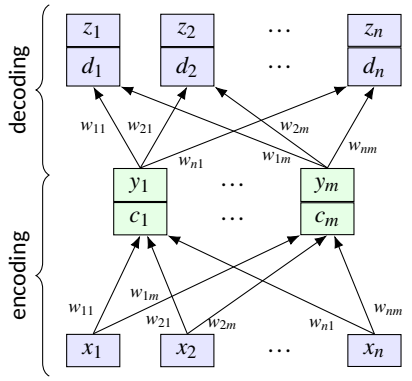


Figure 3 – Architecture of the AE. Middle layer Y is the compressed representation of input X, and Z is the reconstruction of X. The rest of the figures and tables use the same color scheme: blue denotes the external nodes, while green identifies the middle layer.

AEs, as a typical ANN, construct a layer output based on scaled inputs from the previous layer. These scales are referred to as *weight* values: every connection in the network has its weight value w associated with it. The same principle holds for the encoder and decoder portions of the AE. In theory, this corresponds to the MAC operation.

While the operation of the AE is not complicated, there are things to note.

First, AEs are unsupervised networks: there is no need to have labeled data available for the training process. This is caused by the nature of the AE: the network's output must be the same as its input, i.e., everything is inherently available for constructing the loss function, which is the metric used for the network training.

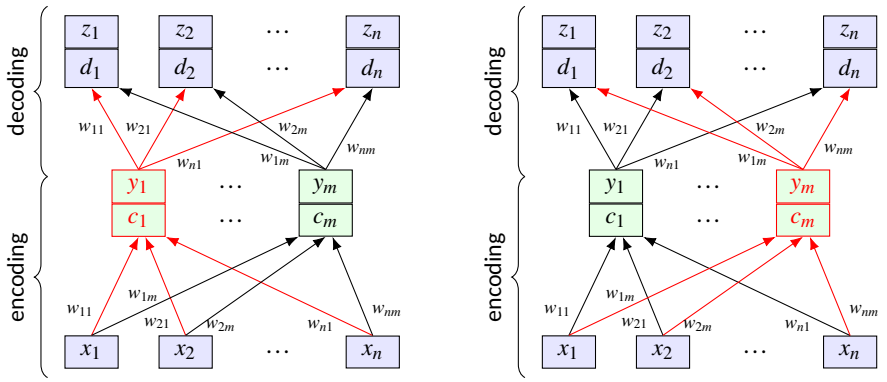
This fact that AEs are unsupervised networks brings them to the focus of this thesis. Providing three state-of-the-art AE architectures for resource-constraint systems with HW enable learning is especially valid for unsupervised networks. On the other hand, embedding the learning process into the final deployment platform is not required for supervised networks: the training process requires labeled data and, therefore, can not be performed while already deployed. Therefore, the training process can be carried out

using any available powerful enough HW.

Further, beyond the fact that AEs are unsupervised networks, the central feature is the representation of the input data in its middle layer. What is important here is that AEs are constructed so that the output can not directly mirror the input. In figure 3, the middle layer is deliberately *narrower*, with less Processing Elements (PEs) on it to illustrate this: the network has to compress its input signal. Further, these compressed features are then used to construct the network output. Therefore, as the output has to match the input, the middle layer of the well-trained AE has to contain intrinsic features of the input signal.

The ability of AEs to extract the input signal features makes it potentially helpful in data pre-processing and paves the road for future work. E.g., feeding the following ANN with features extracted by a AE potentially reduces the layer count of the following network or alternatively speeds up the cascaded ANN training process.

However, feature extraction in AE network requires additional care to succeed. For example, a possible scenario would be remembering the input signal and producing the output based on memorized indexes. This is a typical overfitting problem: the network performs exceptionally well on training data but poorly on a test set. Figure 4 presents this behavior: on figure 4a, the middle layer node c_1 has learned to react on a specific input and is the only node participating in the output code formation. Figure 4b presents the same scenario for the middle layer node c_m.



(a) Middle layer node c_1 is the only one reacting to the specific input, and the output is constructed using the same single node.

(b) Middle layer node c_m is the only one reacting to the specific input, and the output is constructed using the same single node.

Figure 4 – Example of the overfit AE network: a specific middle layer node has learned to represent a single input.

This thesis concentrates on delivering state-of-the-art architectures for the CAE, with the learning process also performed in HW. While CAE is still an AE, section 2.2 introduces its regularization methods.

Further, in addition to the background information provided in this section, the thesis continues with the literature review in section 2.3.

The theoretical background and understanding of the required calculations are definitely necessary before realizing any HW accelerator. Therefore, all the mathematical background of the CAE forward pass and backpropagation is provided in subsections 2.4.1 and 2.4.3. These subsections provide the full calculation scheme, including the descriptions of the Rectified Linear Unit (ReLU) normalization function and all the equations for derivatives used for training.

Additionally, as the basis of the backpropagation, the training process, the loss function deserves a separate paragraph: subsection 2.4.2. The backbone of the loss function is Mean Squared Error (MSE); however, CAE introduces an additional regularization term, which should reduce the sensitivity of the coded values to the small input changes. Mathematically, this term is the Frobenius norm of the Jacobian matrix. The total loss function, MSE plus the additional regularization, is the input for the training process, as in the case of any ANN, although the loss functions vary case by case. Subsection 2.4.4 present the results of the backpropagation calculations: equations for weight and bias updates.

Although the mathematical background is well covered, implementing the calculus on real HW might require additional optimizations. These optimizations should partition the equations to make it possible to reuse the calculated values. Subsection 2.5.1 addresses this and provides updated equations for weight and bias updates.

In total, three different architectures are described in the following chapters. Although they all realize the same network, there are differences in either communication schemes between the PEs or optimizations of those, i.e., optimizations in targeting performance or conserving the HW while keeping the same functionality. Furthermore, as mentioned before, all three approaches can perform the learning process entirely in HW. In a nutshell, all three PEs architectures wrap a HW Digital Signal Processing (DSP) slice complemented by dedicated control Finite State Machine (FSM), block Random Access Memory (RAM) and some registers for storing the intermediate calculation results.

The first described architecture, baseline (BL), described in subsection 2.5.3, follows the structure of the actual CAE network: there are dedicated PEs for different network layers. Also, the data flow follows the layered structure, and the architecture uses a cross-bar switch for communication between the PEs.

In addition to the node architecture's data path, subsection 2.5.3 also provides tables specifying the detailed execution flow during the forward pass and backpropagation. At the same time, the coloring scheme in the presented tables helps to understand the layered execution, and the accompanying description provides a complete understanding. Also, the BL architecture uses optimized equations described in subsection 2.5.1.

The scalability of the BL architecture is limited by the resources available in the target platform: every PE wraps a HW DSP slice. However, the equations provided in the theoretical discussion section must be developed further for vertical scalability to support additional layers.

The second architecture, CAE with efficient Communication (CCom), described in subsection 2.5.4, focuses on optimizing the HW cost of the communication channel. Although the cross-bar switch connects all the PEs, subsection 2.5.6 shows that the area of the actual HW occupied by it is relatively high compared to the resources allocated for PEs.

The CCom architecture skips the cross-bar-switch style communication channel. Instead, it uses a carousel-like transmission scheme: data is rotated to the next node every timestep.

Furthermore, there is an additional consequence: while the BL architecture PEs are synchronized by the availability of data or communication resources provided by the cross-bar, the PEs of the CCom architecture are designed to be synchronous, i.e., the PEs expect the transmission channel to be available at certain moments. Therefore, in addition to the fewer resources allocated for the inter-PE communication, as shown in subsection 2.5.6, subsection 2.5.7 shows that the throughput of the CCom architecture is also higher compared to the BL. Moreover, the throughput of the CCom architecture is almost equal to the theoretical maximum presented in subsection 2.5.2. This effect can be explained by the availability of the resources guaranteed by the synchronous design; there are more

infrequent wait cycles compared to the BL version.

As with BL architecture, the datapath of the PE of the CCom is also presented, and the flow of calculations is the same. However, the CCom architecture does not differentiate between PEs based on the layer they belong to; all the PEs share the same design and wrap a HW DSP slice. This makes the architecture more universal, and together with a lighter use of HW resources spent for the inter-node communication, subsection 2.5.6 shows that it ultimately allows a bigger CAE network to be synthesized on the same target platform.

Subsection 2.5.5 presents the third CAE architecture provided in this thesis: Resource Optimised CAE with efficient Communication (CCom-RO). It is similar to the CCom version but skips some performance-biased additions introduced in the latter. It also uses the carousel-like transmission channel, but unlike CCom, CCom-RO does differentiate nodes based on the network layer.

The configuration of the CCom-RO design flows from the widest CAE network layer towards the narrower ones. First, the most expansive layer defines the features required by all the PEs. Next, if implementing the following layer requires some additional HW, only the amount of nodes matching the narrower layer PE count include the required additions.

However, optimizing away some of the HW from CCom-RO network nodes comes at a price. That means that the CCom architecture keeps all the PEs constantly busy and even calculates multiple sets of some results during the execution to ensure the data is available on every timestep. However, that is not the case for CCom-RO: some results are calculated only by the fully equipped PEs, and the carousel transmission channel introduces an additional delay. It requires additional timesteps to rotate the necessary data to all the depending PEs.

As stated before, all the described architectures perform the same from the functional point of view. The differences come from the inter-PE transmission channel and different emphasis on execution speed or allocated HW resources or performance.

The synthesis results presented in subsection 2.5.6 show that the CCom-RO yields the most extensive CAE network on the same target platform. This result was expected: it improves the communication channel over the BL architecture and, as the acronym suggests, is *resource optimized* compared to the CCom version.

Also, the performance of all three architectures is compared in subsection 2.5.2. It is shown that the CCom architecture results in the fastest possible CAE network. Moreover, CCom architecture almost reaches the theoretical maximum using the presented theoretical basis.

Further, the proof of the correct execution is provided in subsection 2.5.8: the MNIST database of handwritten digits was used for that purpose. Finally, it is shown that the network converges and can produce an output similar to the network input, as required by autoencoders, CAE being one of those.

2.2 Contractive Autoencoder

As stated before, AE is a type of ANN that reconstructs its input signal to the output using the extracted features on the middle layer. Furthermore, there are different types of AEs, distinguished using the regularization methods: additional criteria in the loss function to force the middle layer encoding towards valuable features.

One natural assumption would be that similar inputs should yield similar encoding on the middle layer, i.e., if the input changes very little, it can be reconstructed using almost the same set of features. This is precisely the basis of how the CAE uses regularization.

To achieve this, CAE uses the regularization term based on derivatives of the encoding with respect to the input in addition to the primary qualification that the output has to be the same as the input.

In mathematical terms, this additional regularization is expressed as the Frobenius norm. In a nutshell, a Frobenius norm is a matrix of partial derivatives of middle-layer features with respect to the inputs. This additional regularization forces small changes in the inputs to yield the same features: it makes similar inputs to *contract* in generating the output. In conclusion, for CAE, the regularization controls the behavior of the encoder portion of the network.

2.3 Literature Review

This chapter reviews the state-of-the-art implementations of AEs found in the literature, while the thesis focuses on implementing CAE on a resource constraint system such as a FPGA. More precisely, as the CAE is an unsupervised network, the provided architectures embed the training process in HW to enable autonomous operations, and the literature is reviewed accordingly.

ANNs have gained popularity in various fields nowadays. If carefully searched, examples can probably be found for any possible application, including image recognition, natural language translation, human activity recognition, and anomaly detection [19, 20, 21].

The most appealing feature of such algorithms is the ability to extract latent features from the data automatically. Furthermore, this kind of behavior increases the modeling capabilities [22].

However, regarding handheld and similar resource-constraint devices, the networks are usually pre-trained on other platforms, or data is offloaded to the cloud for computations [23]. While the state-of-the-art addresses FPGA based ANN accelerators ([13, 14]), CAE with embedded learning is missing from the literature.

Further, focusing towards AEs, they are beneficial for squeezing latent features out of the input data or reducing the data dimensionality as stated in [24]. While the extracted features can indeed be used in various applications, anomaly detection is one of the fields where AEs have proved to be a reasonable solution.

The work presented in [25] uses AE for detecting anomalies. At the same time, authors in [26] use AE for detecting anomalies in multi-dimensional time-series data. Finally, to complement the theoretical background, authors in [27] use AE for unsupervised anomaly detection scenario, while they also claim that CAE, the same AE flavor addressed in this thesis, excel very well in this field.

Authors in [28] propose using autoencoders to compress the captured biometric Electrocardiogram (ECG) data in wearable devices. While the compression results and reconstructor error are proved to be feasible, authors execute the algorithm on the Cortex-M4 microcontroller. Moreover, the training process is performed using separate hardware in this work. Another work, [29], also proposes autoencoders for biometric data analysis: authors use Electroencephalogram (EEG) signals to analyze the pilots' fatigue. Although the execution HW platform is not restricted in this work, it supports the idea of using autoencoders for biomedical data. However, people move around, and analyzing their health and overall condition can be beneficial. Therefore, this is the field for wearable and resource-constrained devices, and having suitable implementations around would accelerate the deployment of these methods.

Besides analyzing biological data, autoencoders have shown their usefulness in motion recognition. Authors in [30] use a mobile phone and its sensor data to detect different motions of a human. However, again, the training of the AE is not performed using the

same HW. This prevents the network from learning new trends or changes in a user's movements or requires the data to be offloaded to the cloud.

Another work using sensor data for motion detection is presented in [31], where authors address the problem of detecting unseen falls using the AEs. They state that detecting the fall is not trivial due to the lack of labeled data and train the AE using only regular motions. While this work does not address resource-constrained devices, it supports the idea that the availability of such implementations would broaden the deployment possibilities. Constant unsupervised learning of the regular motions in a wearable device could improve the accuracy of such a system even more. Finally, the work presented in [32] provides another example of using AEs for motion detection and, therefore, possible deployment of proposals presented in this thesis.

Further, another field of study of applications of DNNs addresses the predictive maintenance scenarios. For example, the work presented in [33] runs AE on a FPGA to extract features from the regular operation of a motor. Although the authors train the cascaded classifier on the same target HW, the AE portion of the proposal lacks this feature and is trained separately. There are not many implementation details available, but it is worth noting that authors emphasize the need to run this DNN application on such resource-constraint platforms. Transferring all the data to the cloud for processing throughout the monitored system's entire lifecycle is not always feasible.

The work presented in [34] provides energy- and area-efficient implementation of an AE. Authors use the extracted features, the middle layer of the AE, as an input to the CNN based fault classifier. However, the CNN portion of the solution is running on the controller, not FPGA, but is executed upon request. This idea is presented in figure 5: software-based CNN can complement the binary classifier, using the features extracted by the AE as its input. Compared to the proposals in this thesis, the presented work partially binarises the calculations and misses the HW based training of the network.

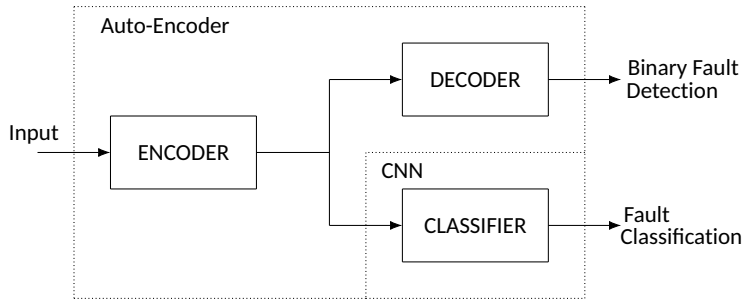


Figure 5 – Cascaded DL network presented in [34]. The features extracted by the AE are used as the input to the software based CNN to complement the binary fault detection output upon request.

The extension to the previous work is provided in [35], where authors propose a partially binarized AE for detecting anomalies in the bearing systems of industrial apparatus. The implementation is very power- and area-efficient but, as stated before, lacks training in HW. However, the missing HW based training should not be taken as criticism towards authors: not every application necessarily needs it. Instead, this reference serves as an example that HW implementations with embedded training are mostly missing from the literature.

An FPGA-based solution for monitoring industrial machinery is proposed in [6]. Moreover, again, authors restrict themselves to semi-supervised learning using another more

powerful HW than the deployment target FPGA. Therefore, the availability of solutions or proposals for the training process on the target HW platform could open new opportunities for these kinds of studies.

Another work addressing unsupervised anomaly detection in resource-constraint devices is presented in [36]. Although the proposed method uses a flavor of a CAE network, the main focus is on the algorithm and not the HW realization. However, the presented study calls for efficient HW based implementations of such algorithms.

AEs have also found their way to boost indoor location services' capabilities. However, solutions based on Global Positioning System (GPS) are not applicable here due to the satellites' signal unavailability. Instead, the parameters of wireless network signals are used. Authors in [37] propose a solution to extract such signal features using an AE. They use FPGA based accelerator for executing the network, but again, do not have the learning implemented in HW: the network has to be trained on a different platform.

Authors in [38] use AE stacked with Long short-term memory (LSTM) for outlier detection. This work nicely demonstrates the benefit of using an ensemble of different networks: LSTM is fed with latent features extracted by the AE. The target platform used in this work is based on Xilinx FPGA, DL accelerators are realized using programmable logic. However, the proposed accelerator involves only the inference phase, and the network's training is performed before deployment. On the other hand, outlier detection could benefit from continuous training to overcome the drift in the normality of monitored processes.

Another work that uses AE for anomaly detection is presented in [12]. Authors use the AE for detecting analog trojans in manufactured integrated circuits and get good results. Also, the implementation is based on FPGA, but again, misses the HW based training. However, as stated in this referenced work, changing the weight values might be required if the noise level changes. This excellent example of an environmental drift demonstrates the need for constant HW based training to overcome the issue.

Further, authors in [39] present the FPGA based implementation of an AE with the focus on the inference phase, no training included. However, the authors claim that using fixed-point data representation gives comparable results to using floating-point data types, supporting the proposals presented in this thesis. Also, the extensive analysis provided in this work clearly shows that FPGA based implementations are more power-conservative compared to the GPU counterparts. However, there is a considerable performance gap between these two platforms in this referenced work, leaving some interpretation room in the presented power figures.

Another general FPGA based ANN accelerator is proposed in [40]. The referenced work is somewhat similar to the proposals presented in this thesis: only the most expansive layer is implemented in HW, and all the following layers reuse these resources. However, the activation function is implemented separately as a single shared unit for all the PEs, reducing the execution speeds. Nevertheless, on the other hand, this approach allows using higher HW complexity for the activation function; a single unit can reserve more HW resources that do not multiply by the size of the implemented layer. However, compared to the proposals in this thesis, the solution lacks HW based training.

Yet another realization of the AE on FPGA can be found in [41]. However, similar to the previously reviewed works, this implementation addresses the inference phase only. But there are also similarities: the referenced work uses fixed-point data representation. Also, the work provides a comparison to GPU platform using floating-point. Authors claim that using fixed-point data degrades the precision of the network, but the loss stays within 10%, depending on the exact format selection.

Finally, authors in [42] provide a solution for HW based inference and training for a stacked AE. The referenced work also compares the FPGA based solution to the GPU based implementation, where FPGA outperforms the GPU in power consumption but lags in terms of performance. Furthermore, the performance of the FPGA based solution is significantly lower for both inference and backpropagation phases. This performance degradation can be related to the fact that authors use OpenCL high-level language for the FPGA design. Naturally, this kind of approach accelerates the development cycle. However, carefully crafted solutions using lower level HDL like Verilog or VHDL surely can increase the network's throughput and make FPGA based solutions more appealing alternatives to GPU counterparts.

To conclude the review, there are quite a few implementations of different flavors of AEs on FPGA platform, but the HW based training still needs to be addressed by the researchers: this is the field where this thesis provides its contribution. Although the focus is on the FPGA based implementation of CAE, the ideas can be used to extend the proposals for other types of networks.

The main contribution of this work is to provide *the first full hardware-based implementation of the CAE [43], comprising hardware-implemented learning*. Additionally, the thesis follows the proposals from the authors of [44] and shares the network weights on the encoder and decoder parts of the CAE, which helps to conserve the memory requirements of the implementation. Additionally, as proposed in [45], the weights and biases and all the calculations use the fixed-point representation of data.

2.4 Background: Theory of Contractive Autoencoder

2.4.1 Forward Pass

The forward pass, inference phase of an ANN generates the output of the network. During this phase, the input signal passes all the network layers and undergoes all the corresponding transforms using the network weight values, layer-specific coefficients, and global network parameters found during the network's training.

The following presents the equations for CAE forward pass calculations for the network presented in figure 3, where n stands for the width of the input and output layers and m denotes the number of nodes in the hidden layer.

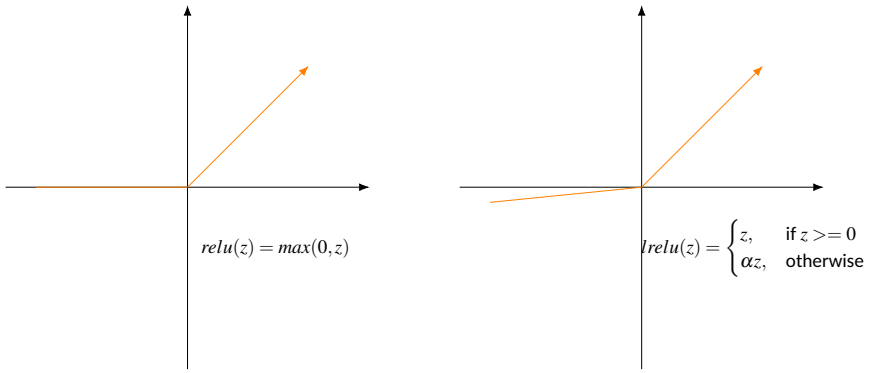
First, the network receives its input via the x nodes, the input layer. Then, all the input layer nodes connect to the next layer using the weight values w as the scaling factors: the weight values define the impact of a node input.

Equations (1) and (2) present the calculations to evaluate the c and y values in the middle layer, where b stands for the node *bias* value. The same equations hold if there were more layers in the network presented in figure 3: every node in every layer accumulates the scaled inputs from the previous layer, adds the bias value, and applies the activation function. The activation function used in the current example is computationally inexpensive ReLU. While the original ReLU function was proposed in [46], authors in [47] propose a modification to allow low negative values to *leak* to the output to avoid dying neurons problem analyzed by the authors. Figure 6 illustrate the differences between these two activation functions. This thesis makes use of the modified, *leaky* ReLU.

$$c_j = \sum_{i=1}^{n} w_{ij} x_i + b_j^{(c)} \tag{1}$$

$$y_j = f(c_j) = ReLU(c_j) \tag{2}$$

The nodes y in the middle layer hold the *coded* input value. A similar procedure then



(a) ReLU activation function, all input values below zero are limited to zero.

(b) LeakyReLU activation function multiplies negative inputs by α, allowing low negative values to leak *to the output.*

Figure 6 – ReLU and LeakyReLU activation functions.

follows in the *decoding* layer: equations (3) and (4) present the calculations for evaluating the output of the network.

$$d_i = \sum_{j=1}^{m} w_{ij} y_j + b_i^{(d)} \tag{3}$$

$$z_i = g(d_i) = ReLU(d_i) \tag{4}$$

2.4.2 Loss Function

Training of the ANN, including CAE, is about minimizing the loss function. The training data is pre-labeled for supervised networks, like CNN, and the network output is assessed based on these labels. On the other hand, training the unsupervised networks like CAE does not require manually prepared data: in the case of CAE, the output has to be the same as the input.

MSE can be used to evaluate the similarity of the network output to its input: equation (5). The network loss L increases if the output does not match the input, and the internal weight and bias values can be changed based on that information.

$$L(X,Z) = \frac{1}{n} \sum_{i=1}^{n} (z_i - x_i)^2 \tag{5}$$

In addition to this, the CAE network adds a regularization term to the loss function. The purpose of an additional regularization is to fulfill some specific criterion: in the case of CAE the additional term should reduce the sensitivity of the input to the coded values in the middle layer and yield to more robust encoding. Mathematically this term translates to the Frobenius norm of the Jacobian matrix (equation (6)) and ensures that a slight change in networks input translates to the same encoding in the middle layer.

$$\|J_f(x)\|_F^2 = \sum_{i=1}^{n} \sum_{j=1}^{m} \left(\frac{\partial y_j}{\partial x_i} \right)^2 \tag{6}$$

The final loss function of the CAE network is the sum of equations (5) and (6): equation (7), where $\theta = W, B^{(c)}, B^{(d)}$ is the collection of all the network parameters, weights

and biases, and λ is the hyper parameter to limit the amount of the contraction term in the total loss.

$$\mathscr{J}_{CAE}(\theta) = L(x, g(f(x))) + \lambda \|J_f(x)\|_F^2 \tag{7}$$

If the network output corresponds to the input, the MSE portion of the loss function is minimal: every network weight and bias value is adjusted accordingly during the training, and the standard method for this is gradient descent, covered in subsection 2.4.3.

2.4.3 Gradient Descent

The gradient descent is the standard method for training the ANNs, including CAE. The purpose of the procedure is to adjust all the network weight and bias values based on the output of the loss function. First, the derivatives of the loss function with respect to all the network parameters are calculated, and the values are updated based on these results.

The total loss of the network, equation (7), is the sum of two terms: the MSE term and the additional contraction term, and the derivatives for these two can be calculated separately.

After substituting the term $(z_i - x_i)^2$ with $l_i = (z_i - x_i)^2$ in in equation (5), the derivative of the MSE with the respect to a weight value w_{uv} can be found using the chain rule as per equation (8).

$$\frac{\partial L}{\partial w_{uv}} = \frac{1}{n} \sum_{i=1}^{n} \left(\frac{\partial l_i}{\partial z_i} \frac{\partial z_i}{\partial d_i} \frac{\partial d_i}{\partial w_{uv}} \right) \tag{8}$$

The first term in the chain rule is the MSE loss derivative with respect to z_i: equation (9).

$$\frac{\partial l_i}{\partial z_i} = 2(z_i - x_i) \tag{9}$$

The second term is the derivative of the ReLU activation function, z_i with respect to d_i: equation (10).

$$\frac{\partial z_i}{\partial d_i} = \begin{cases} 1, & \text{if } d_i >= 0 \\ \alpha, & \text{otherwise} \end{cases} \tag{10}$$

The derivative of the decoding d_i with respect to the weight w_{uv} needs to consider only one term from the equation (3): the one where $j = v$. All the other terms are independent of w_{uv}; therefore, they have zero gradients. The weight value of this selected term is marked as $w_{iv}^{(d)}$ if the following equations and its first index is marked as $i^{(d)}$.

However, y_j in equation (3) is the function of weights itself (equations (1) and (2)). Similar to the equation (3), equation (1) has to consider only the term which has the dependency to w_{uv}: the one where $i = u$. Again, the weight value in questions is marked as $w_{iv}^{(c)}$, and its first index is marked as $i^{(c)} = u$.

The weight value included by the term selected from the equation (1) has both of its indexes fixed to $i = u$ and $j = v$, while $w_{iv}^{(d)}$ can have $i^{(d)} \neq u$. This condition specifies if the weight values included by terms selected from equations (1) and (3) are the same or not, or in other words if the y_j in equation (3) is the function of the same weight value it is multiplied to. Equation (11) specifies these two cases for the derivative calculation.

$$\frac{\partial d_i}{\partial w_{uv}} = \begin{cases} \frac{\partial d_i}{\partial y_v} \frac{\partial y_v}{\partial c_v} \frac{\partial c_v}{\partial w_{uv}}, & \text{if } i^{(d)} \neq u \\ y_v + w_{iv} \frac{\partial y_v}{\partial c_v} \frac{\partial c_v}{\partial w_{uv}}, & \text{if } i^{(d)} = u \end{cases} \tag{11}$$

where:

$$\frac{\partial d_i}{\partial y_v} = w_{iv} \tag{12}$$

$$\frac{\partial y_v}{\partial c_v} = \begin{cases} 1, & \text{if } c_v >= 0 \\ \alpha, & \text{otherwise} \end{cases} \tag{13}$$

$$\frac{\partial c_v}{\partial w_{uv}} = x_u \tag{14}$$

Figure 7 illustrates the term $i^{(d)} \neq u$ in equation (11), and makes an example for calculation path $\partial d_2 / \partial w_{11}$, $u = 1$ and $v = 1$. There is exactly one path from the node d_2, which includes the weight value w_{11}. The dotted line indicates the path in question, which goes through w_{21} in the decoder portion, marked with the blue arrow, and w_{11} in the encoder, marked with the red arrow.

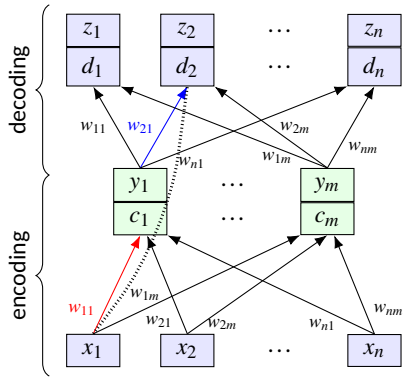


Figure 7 – Illustration for equation (11)*: calculation path for* $\partial d_2 / \partial w_{11}$, $u = 1$ *and* $v = 1$*. The decoder portion has to select the path through the weight value* w_{21}*, where* $i^{(d)} = 2$*. Here,* $i^{(d)} \neq u$*.*

Further, figure 8 illustrates the term $i^{(d)} = u$ in equation (11), and makes an example for calculation path $\partial d_2 / \partial w_{21}$, $u = 2$ and $v = 1$. The value of node y_1 also depends on w_{21}, and the derivative chain rule applies. The dotted line indicates the calculation path, which goes through the same weight w_{21} in the decoder and encoder portions, marked with red arrows.

Calculating the gradients for the bias values has no exceptional case, as the coding and decoding layers have a separate set of those.

Equation (15) specifies the calculation of the MSE loss derivative w.r.t. $b_i^{(d)}$.

$$\frac{\partial L}{\partial b_i^{(d)}} = \frac{1}{n} \frac{\partial l_i}{\partial z_i} \frac{\partial z_i}{\partial d_i} \frac{\partial d_i}{\partial b_i^{(d)}} \tag{15}$$

Equation (16) specifies the derivative of the MSE loss w.r.t. $b_j^{(c)}$.

$$\frac{\partial L}{\partial b_j^{(c)}} = \frac{1}{n} \sum_{i=1}^{n} \left(\frac{\partial l_i}{\partial z_i} \frac{\partial z_i}{\partial d_i} \frac{\partial d_i}{\partial y_j} \frac{\partial y_j}{\partial c_j} \frac{\partial c_j}{\partial b_j^{(c)}} \right) \tag{16}$$

The contraction term is specified by the equation (6). equation (17) specifies the calculation of derivatives of y_i w.r.t. x_i included in the contraction term.

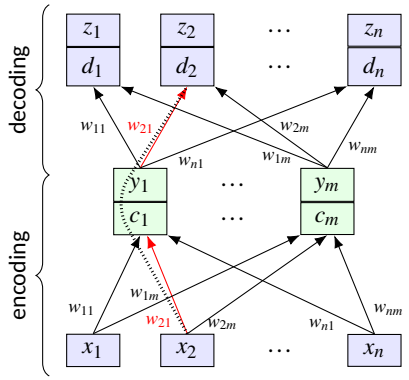


Figure 8 – Illustration for equation (11)*: calculation path for* $\partial d_2/\partial w_{21}$*,* $u = 2$ *and* $v = 1$*. The decoder portion has to select the path through the weight value* w_{21}*, where* $i^{(d)} = 2$*. Here,* $i^{(d)} = u$*.*

$$\frac{\partial y_j}{\partial x_i} = \frac{\partial y_j}{\partial c_j}\frac{\partial c_j}{\partial x_i} \tag{17}$$

where:

$$\frac{\partial c_j}{\partial x_i} = w_{ij} \tag{18}$$

Therefore, every element in the contraction term equals to equation (19).

$$r_{ij} = \left(\frac{\partial y_j}{\partial x_i}\right)^2 = \left(\frac{\partial y_j}{\partial c_j}\right)^2 w_{ij}^2 \tag{19}$$

Derivative term $\partial y_j/\partial c_j$ in equation (19) is constant according to the equation (13). Therefore, derivative of r_{ij} w.r.t. w_{ij} can be calculated according to the equation (20).

$$\frac{\partial r_{ij}}{\partial w_{ij}} = 2w_{ij}\left(\frac{\partial y_j}{\partial c_j}\right)^2 \tag{20}$$

2.4.4 Weight and Bias Update

The negative value of the gradient specifies the direction of change for a parameter to minimize the loss function (equation (7)).

The sum of equations equations (8) and (20) specify the gradient of a weight value. Therefore, the new values for weights have to be calculated according to the equation (21), where α stands for the learning rate and λ limits the effect of the contraction term.

$$w_{ij} = w_{ij} - \alpha\left(\frac{\partial L}{\partial w_{ij}} + \lambda\frac{\partial r_{ij}}{\partial w_{ij}}\right) \tag{21}$$

The bias value updates should follow a similar scheme. Equations (22) and (23) present the update formulas, where β sets the update rate for the biases.

$$b_i^{(d)} = b_i^{(d)} - \beta\frac{\partial L}{\partial b_i^{(d)}} \tag{22}$$

$$b_j^{(c)} = b_j^{(c)} - \beta\frac{\partial L}{\partial b_j^{(c)}} \tag{23}$$

2.5 Novel Architecture for Contractive Autoencoders

This section presents the HW architectures for CAE inference and backpropagation calculations. In total, three architectures are proposed.

The Xilinx Zynq-7020 System On Chip (SoC) is the target HW platform in this work. It has dual-core ARM Cortex-A9 processors and a programmable logic portion for custom HW implementations. The programmable logic section contains 85K logic cells, 53200 LUTs, 106400 flip-flops, 140 36Kbit block RAMs, and 220 DSP slices.

According to the equations (1), (3) and (8), the multiply-accumulate calculations are heavily used; therefore, all the proposed architectures make use of the DSP Intellectual Property (IP) blocks available in the target architecture. Utilizing the DSP IPs frees the rest of the programmable logic for other purposes and allows packing more network nodes to the target HW.

2.5.1 Equations optimization

While the HW implementations of the CAE forward pass exist, the backpropagation and weight update functions presented in subsections 2.4.3 and 2.4.4 are more complex and harder to implement in HW efficiently. Therefore, this section regroups the calculations and defines some reusable values. As shown below, those reusable values can be calculated once and reused multiple times, yielding more efficient HW.

First, equation (24) defines the term k_i:

$$k_i = \frac{1}{n}\frac{\partial l_i}{\partial z_i}\frac{\partial z_i}{\partial d_i} \tag{24}$$

Using this newly defined term k_i, rewriting the equation (8) results in equation (25):

$$\frac{\partial L}{\partial w_{uv}} = \sum_{i=1}^{n}\left(k_i \frac{\partial d_i}{\partial w_{uv}}\right) \tag{25}$$

Next, substituting equations (12) and (14) into equation (11) gives equation (26):

$$\frac{\partial d_i}{\partial w_{uv}} = \begin{cases} w_{iv}\frac{\partial y_v}{\partial c_v}x_u, & \text{if } i^{(d)} \neq u \\ y_v + w_{iv}\frac{\partial y_v}{\partial c_v}x_u, & \text{if } i^{(d)} = u \end{cases} \tag{26}$$

From equation (26), it can be seen that the check for the case where $i^{(d)} = u$ can be ignored while calculating equation (25) and the correction term $k_u y_v$ can be added to the result to compensate for it: equation (27).

$$\frac{\partial L}{\partial w_{uv}} = x_u \frac{\partial y_v}{\partial c_v}\sum_{i=1}^{n}(k_i w_{iv}) + k_u y_v \tag{27}$$

Further, equation (28) defines the reusable summation S_v.

$$S_v = \frac{\partial y_v}{\partial c_v}\sum_{i=1}^{n}(k_i w_{iv}) \tag{28}$$

The weight update value can be rewritten by using this reusable term: equation (29)

$$\frac{\partial L}{\partial w_{uv}} = x_u S_v + k_u y_v \tag{29}$$

Using equation (21) and these newly defined substitutions and reordering the weight value update can be rewritten: equation (30).

$$w_{ij} = w_{ij} - \alpha\left(x_i S_j + k_i y_j\right) - 2\alpha\lambda w_{ij}\left(\frac{\partial y_j}{\partial c_j}\right)^2 \tag{30}$$

Similarly, the output layer bias value calculations, equations (15) and (22), can use the reusable term k_i: equation (31).

$$b_i^{(d)} = b_i^{(d)} - \beta k_i \tag{31}$$

Also, the internal layer bias value calculations, equations (16) and (22), can be rewritten using the reusable term S presented in equation (28): equation (32).

$$b_j^{(c)} = b_j^{(c)} - \beta S_j \tag{32}$$

In conclusion, recognizing and pre-calculating the k (equation (24)) and S (equation (28)) values allows the network weight and bias update process to reuse these results.

2.5.2 Execution time estimation

This subsection provides the theoretical timing estimations for the forward pass and backpropagation execution steps. Then, the described architectures will be compared against the derived values to assess the actual throughput in subsection 2.5.7.

During the forward pass, every network node, PE, must multiply every input from the previous layer by the corresponding weight value and accumulate the results. In addition, every PE must add the bias value and apply the ReLU activation function: equations (1) to (4). As every PE accommodates a HW DSP slice, all these operations execute in a single cycle. This includes the activation function ReLU as it boils down to single multiplication.

Equation (33) presents the generalized formula for the required cycles to complete the forward pass, where l stands for the number of layers and n_i is the i-th layer input size.

$$C_{fwd} = \sum_{i=1}^{l}(n_i + 2) \tag{33}$$

For example, applying equation (33) for the network presented in figure 3 results in $n+m+4$ cycles to complete the forward pass.

To estimate the theoretically required cycle count for backpropagation, equation (21) presenting the formula for updating a single weight value is followed.

The term $\partial L/\partial w_{ij}$ is computationally most demanding as it includes the summation term as per equation (8). To calculate the required cycle count for this term, $\partial l_i/\partial z_i$ requires one cycle for subtraction, $\partial z_i/\partial d_i$ requires one cycle, and $\partial d_i/\partial w_{uv}$ requires three cycles. Multiplying these values takes another two cycles. Therefore, the summation term requires $7n$ cycles in total.

Further, $\partial r_{ij}/\partial w_{ij}$ (equation (19)) requires four cycles: one cycle for $\partial y_j/\partial c_j$, one cycle for calculating power of two, and another two cycles to multiply this value to the weight value and to multiply it by two.

Additionally, multiplying $\partial r_{ij}/\partial w_{ij}$ by λ consumes one cycle, adding the correction y_v to $\partial L/\partial w_{ij}$ present in equation (11) and multiplying this to $2/n$ consumes two cycles. Adding these two terms takes another cycle, and multiplying this sum by α takes one cycle. Finally, subtracting the result from the present weight value also takes one cycle.

Therefore, it takes ten cycles in addition to the $7n$ cycles required by the summation term present in equation (8) to complete the update for a single weight value.

The network presented in figure 3 has m weight values assigned to every PE, and all these values have to go through the same updated procedure: equation (34) presents the formula for the required cycle count C_{bpw}.

$$C_{bpw} = m(7n + 10) \tag{34}$$

However, subsection 2.5.1 provides some optimizations for the backpropagation calculations. Extracting re-usable portions speeds up the calculations and should also be considered for theoretical timing estimations for a true comparison.

Equation (30) has to be followed to estimate the required cycles for updating all the weight values.

The cycle count for the contraction term is the same as before: four cycles. Plus, an additional cycle to multiply by the term $\alpha\lambda$, totaling five cycles.

The remaining operations in equation (30) are single cycle multiplications, additions, and subtractions: the total required cycle count for updating a single weight value totals eleven cycles.

However, equation (30) contains the re-usable k and S values, which require additional cycles to calculate.

Equation (24) has to be followed to estimate the cycle count for k, where $\partial l_i/\partial z_i$ takes two cycles, $\partial z_i/\partial d_i$ takes one cycle and multiplying these values and the constant $1/n$ takes additional two cycles. Therefore, the total cycle count to calculate k is five cycles.

Further, equation (28) defines the required calculations for S. The summation term contains a multiply-accumulate operation carried out by the DSP slice in a single cycle. Therefore, calculating S requires $n + 1$ cycles.

Equation (35) defines the total cycle count required to update all the weights in the network: eleven cycles times m to update all the weights according to the equation (30), plus $5 + n + 1$ cycles to calculate the reusable portions k and S.

$$C_{bpw\_optimized} = 11m + n + 6 \tag{35}$$

2.5.3 Architecture 1: Baseline (BL)

This subsection presents the BL architecture for CAE. As the following architectures, the BL version can also execute training, i.e., backpropagation, in the HW.

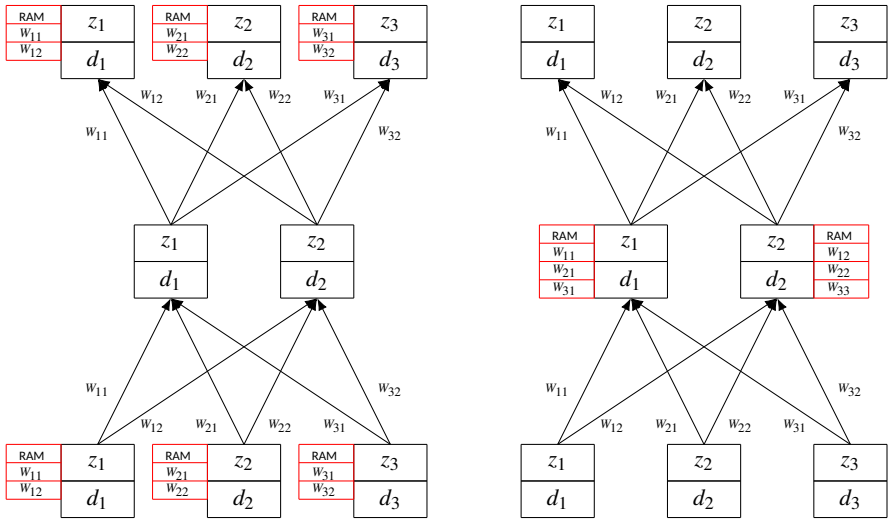
Although the BL architecture makes use of optimizations of calculations presented in subsection 2.5.1, it follows the logical structure of the CAE network: every network node is implemented as a separate PE. Data flow and execution of calculations also follow the actual CAE architecture. Forward pass calculations start by propagating the input values to the middle layer nodes where multiply and accumulate operation (equation (1)) and applying the activation function (equation (2)) takes place, followed by similar operations in the output layer (equations (3) and (4)). The network training follows a similar layer-to-layer flow but in the opposite direction, from the output layer towards the network input.

This scheme means that while the nodes of different network layers are well separated and more straightforward controlling state machines can be used, only one layer executes at a time; the resources associated with the other layers stay idle.

The BL and the following architectures utilize the HW block RAM IPs for storing the network parameters and weight values; Zynq-7020 has 140 units of 36 Kbit RAMs, and

every unit is configurable as two 18 Kbit blocks totaling 280 units. However, the design consideration is which network nodes to assign these resources.

As the internal layer holds the compressed input representation, there are usually fewer middle layer units than input and output units. Therefore, the BL architecture assigns the block RAMs to the middle layer: as $m < n$, this scheme uses fewer blocks while storing more values to a single RAM, as every middle layer PE connect to n external layer nodes (figure 3) and has to store n weight values compared to m values in case of an input or output node. Figure 9 illustrates this situation, assigning weight value block RAMs to external layer nodes (figure 9a), requires more, but smaller size block RAMs compared to when assigned to the internal layer nodes (figure 9b).



(a) Block RAMs assigned to the external layer nodes.

(b) Block RAMs assigned to the internal layer nodes.

Figure 9 – Assigning block RAMs to external layer nodes requires more, but smaller RAMs, compared to when assigned to the internal layer.

Figure 10 depicts the data path of the BL architecture external layer PE. Every unit contains one DSP slice, supporting multiply and accumulate operations and registers T, b, z, and x for result storage. Inputs of the DSP slice have multiplexers to select necessary data for calculations. The Cxx block is the communication channel entry port to exchange data between the PEs.

Further, figure 11 presents the data path of the BL architecture middle layer PE. Again, every unit contains one DSP slice, supporting multiply and accumulate operations and registers y, T, b for result storage. Also, DSP slice inputs are connected to the multiplexers. Compared to the external layer PE, middle layer version includes the block RAMs to hold the weight values. The Cxx block is the communication channel entry port to exchange data between the PEs.

The CAE architecture, figure 3, requires that every network node can communicate to all the nodes in adjacent layers. However, implementing all these connections in Programmable Logic (PL) is not possible due to the HW limitations.

The BL architecture uses a cross-bar switch for communication: figure 12. However, the main focus of this thesis is the design of the CAE network nodes using the DSP blocks and block RAMs available in the hardware. Therefore, this thesis does not provide an

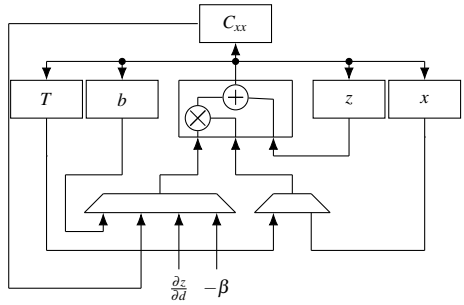


Figure 10 – Data-path of the CAE external layer PE.

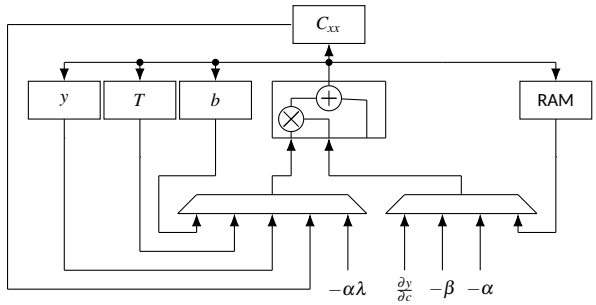


Figure 11 – Data-path of the CAE middle layer PE.

in-depth analysis of different cross-bar flavors but selects butterfly architecture.

As required by the CAE architecture, every input-output PE has to be able to communicate to every internal PE and vice versa. Connecting input-output nodes to one side and internal nodes to the other side of the butterfly cross-bar satisfies this requirement. Additionally, connecting one communication port from both sides of the cross-bar to the AXI bus allows the Zynq processing unit to communicate to every node for controlling and initialization purposes.

The width of the data bus depends on the selected data format: 16 bits in the context of this thesis. However, the implementation adds additional source and destination address fields to indicate the origin and select the target node of data.

The implementation also adds the method to control the network nodes over cross-bar ports marked *CNTRL* in figure 12, and there are two of those to control the PEs on both sides of the cross-bar. The controlling ARM processor uses these ports to write the network's hyperparameters, input the data, and read the network output. During the operation, network nodes, PEs, can distinguish between the control and network execution data by checking the source address: address zero is used for control regardless of the network size.

Table 2 presents the calculation steps for the CAE forward pass, and the coloring scheme indicates the PEs performing the calculations. In step 1, all the input-output nodes broadcast their input value x_i to the middle layer nodes. The following steps, 2 to 4, calculate the sum presented in equation (1), and step 5 adds the bias term $b_j^{(c)}$ to the sum. Step 6 completes the calculation of the middle layer representation y_j by applying the non-linearity function f (equation (2)).

Steps 7 to 8 calculate the terms for summation (equation (3)) and transfer the results

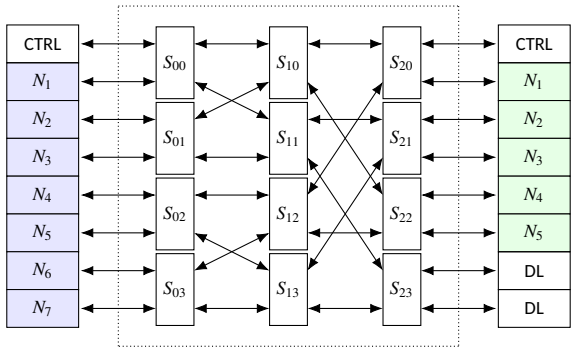


Figure 12 – Layout of the butterfly cross-bar switch. Layers of the CAE connect to the different sides. CTRL ports are used by the ARM processing unit for flow control and data transfer.

to the corresponding input-output nodes. Input-output nodes perform the summation of these terms in steps 9 to 11 and add the bias $b_i^{(d)}$ in step 12 (equation (3)). Step 13 completes calculating the output Z after applying the non-linearity function g (equation (4)).

Table 2 – Calculations of the CAE forward pass

1	$C_{1*} = x_1$	...	$C_{n*} = x_n$
2	$A_1^{(c)} = C_{11} * w_{11}$	...	$A_m^{(c)} = C_{1m} * w_{1m}$
3	$A_1^{(c)} = C_{21} * w_{21} + A_1^{(c)}$	...	$A_m^{(c)} = C_{2m} * w_{2m} + A_m^{(c)}$
...	...	...	...
4	$A_1^{(c)} = C_{n1} * w_{n1} + A_1^{(c)}$	...	$A_m^{(c)} = C_{nm} * w_{nm} + A_m^{(c)}$
5	$T_1^{(c)} = b_1^{(c)} + A_1^{(c)}$	...	$T_m^{(c)} = b_m^{(c)} + A_m^{(c)}$
6	$y_1 = T_1^{(c)} = T_1^{(c)} * f_1$	...	$y_m = T_m^{(c)} = T_m^{(c)} * f_m$
7	$C_{11} = T_1^{(c)} * w_{11}$	...	$C_{1m} = T_m^{(c)} * w_{1m}$
...	...	...	...
8	$C_{n1} = T_1^{(c)} * w_{n1}$	...	$C_{nm} = T_m^{(c)} * w_{nm}$
9	$A_1^{(d)} = C_{11}$	...	$A_n^{(d)} = C_{n1}$
10	$A_1^{(d)} = C_{12} + A_1^{(d)}$	...	$A_n^{(d)} = C_{n2} + A_n^{(d)}$
...	...	...	...
11	$A_1^{(d)} = C_{1m} + A_1^{(d)}$	...	$A_n^{(d)} = C_{nm} + A_n^{(d)}$
12	$T_1^{(d)} = b_1^{(d)} + A_1^{(d)}$	...	$T_n^{(d)} = b_n^{(d)} + A_n^{(d)}$
13	$z_1 = g_1 * T_1^{(d)}$	...	$z_n = g_n * T_n^{(d)}$

Table 3 presents the execution steps for updating the weights and biases, and the coloring scheme of the table corresponds to the table 2. First, every output layer PE calculates the k_i (equation (24)) as the starting point of the gradient descent in table rows 1 to 3 and broadcasts the value to every middle layer PE. Next, table rows 4 and 5 describe the update of the output layer bias values according to the equation (31).

Further, rows 6 to 9 calculate the S_v (equation (28)) and transfer the result back to every connected input-output node. Also, the middle layer PEs transfer their forward pass

value y. Rows 10 and 14 multiply every value received from the middle layer by x, add the $k_i y_j$ term (equation (30)), and transfer the result back to the middle layer. In parallel, every middle layer unit updates its bias value (rows 12-13).

The internal layer finalizes the update process. First, it prepares the weight update value resulting from the contraction term (equation (30)) in rows 17-18. Next, the middle layer unit repeats steps 19-22 for every connected input-output node to finalize the weight update process.

2.5.4 Architecture 2: Efficient Communication (CCom)

This subsection presents the CAE architecture with further optimizations. While the BL architecture nodes are entirely asynchronous and react upon data sent or received from the communication cross-bar switch, the CCom version takes another approach: the PEs are synchronous and expect specific data to be present in its communication channel input and output certain data in a specific clock cycle. This approach simplifies the design of the FSMs inside PEs and can use a communication channel without handshake and address signals.

Therefore, the CCom architecture skips the cross-bar switch style communication channel and replaces it with a carousel-like design: figure 13.

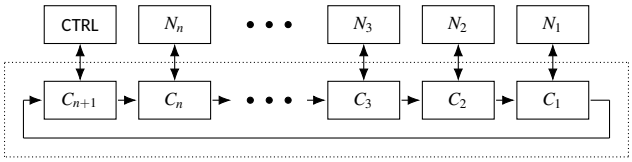


Figure 13 – Carousel like communication channel. Data advances in every clock cycle. The node CTRL is connected to the controlling ARM processing unit.

The carousel implementation advances the data in every clock cycle. However, as the BL architecture, the CCom network needs to be configured as well: the implementation adds two additional data bits to achieve this. One of these bits indicates if a PE sends control data, and the other is used to denote the control from the processor attached to the *CTRL* port. The PEs use the control bit to send the result values of a layer execution, and a separate control bit for data sent by PEs allows the *CTRL* port to distinguish and log those values.

Figure 14 presents the data-path of the CCom network node or PE. The overall approach is the same as in the case of BL architecture: every PE wraps a HW MAC unit. And every PE has a block-RAM, registers for holding various values, and multiplexers for MAC inputs. In the case of CCom architecture, all the PEs share the same architecture.

As stated before, the PEs of the CCom architecture are synchronous: they process the data received from the communication in a sequence defined in the design time.

The first processing step during the execution is the forward-pass calculation. For this, the controlling processor outputs the network input values x_i to the carousel, and every PE stores one of the input values to its x register.

Every PE has layer weight values stored in the block RAM, and the MAC operation follows. During this phase, every MAC performs $A * B + C$ operation, where A and B inputs of the DSP slice are connected to the block RAM and register X outputs, and input C is connected to the communication channel. At the same time, the result of the MAC operation is output to the carousel. This means that every PE adds a $w_{ij}x_i$ term to the value received from the carousel and forwards it, where i is the physical location of the

Table 3 – Calculations of the CAE gradient descent

1	$T_1^{(d)} = z_1 - x_1$	...	$T_n^{(d)} = z_n - x_n$
2	$T_1^{(d)} = 2 * T_1^{(d)}$	...	$T_n^{(d)} = 2 * T_n^{(d)}$
3	$C_{1*} = T_1^{(d)} = \frac{\partial z_1}{\partial d_1} * T_1^{(d)}$	...	$C_{n*} = T_n^{(d)} = \frac{\partial z_n}{\partial d_n} * T_n^{(d)}$
4	$A_1^{(d)} = -\beta * T_1^{(d)}$	...	$A_n^{(d)} = -\beta * T_n^{(d)}$
5	$b_1^{(d)} = b_1^{(d)} + A_1^{(d)}$	...	$b_n^{(d)} = b_n^{(d)} + A_n^{(d)}$
6	$A_1^{(c)} = C_{11} * w_{11}$	...	$A_m^{(c)} = C_{1m} * w_{1m}$
7	$A_1^{(c)} = C_{21} * w_{21} + A_1^{(c)}$	...	$A_m^{(c)} = C_{2m} * w_{2m} + A_m^{(c)}$
...	...	...	...
8	$T_1^{(c)} = C_{n1} * w_{n1} + A_1^{(c)}$	...	$T_m^{(c)} = C_{nm} * w_{nm} + A_m^{(c)}$
9	$C_{*1} = T_1^{(c)} = T_1^{(c)} * \frac{\partial y_1}{\partial c_1}$	...	$C_{*m} = T_m^{(c)} = T_m^{(c)} * \frac{\partial y_m}{\partial c_m}$
10	$A_1^{(d)} = C_{11} * x_1$	...	$A_n^{(d)} = C_{n1} * x_n$
11	$C_{*1} = y_1$	...	$C_{*m} = y_m$
12	$A_1^{(c)} = T_1^{(c)} * -\frac{\beta}{2}$	...	$A_m^{(c)} = T_m^{(c)} * -\frac{\beta}{2}$
13	$b_1^{(c)} = b_1^{(c)} + A_1^{(c)}$	...	$b_m^{(c)} = b_m^{(c)} + A_m^{(c)}$
14	$C_{11} = C_{11} * T_1^{(d)} + A_1^{(d)}$	...	$C_{n1} = C_{n1} * T_n^{(d)} + A_n^{(d)}$
...	...	...	...
15	$A_1^{(d)} = C_{1m} * x_1$	...	$A_n^{(d)} = C_{nm} * x_n$
16	$C_{1m} = C_{1m} * T_1^{(d)} + A_1^{(d)}$	...	$C_{nm} = C_{nm} * T_n^{(d)} + A_n^{(d)}$
17	$T_1^{(c)} = [-\alpha\lambda] * \frac{\partial y_1}{\partial c_1}$	...	$T_m^{(c)} = [-\alpha\lambda] * \frac{\partial y_m}{\partial c_m}$
18	$T_1^{(c)} = T_1^{(c)} * \frac{\partial y_1}{\partial c_1}$	...	$T_m^{(c)} = T_m^{(c)} * \frac{\partial y_m}{\partial c_m}$
19	$A_1^{(c)} = C_{11} * [-\alpha/2]$	...	$A_m^{(c)} = C_{1m} * [-\alpha/2]$
20	$A_1^{(c)} = T_1^{(c)} * w_{11} + A_1^{(c)}$	...	$A_m^{(c)} = T_1^{(c)} * w_{1m} + A_m^{(c)}$
21	$A_1^{(c)} = w_{11} + A_1^{(c)}$	...	$A_m^{(c)} = w_{1m} + A_m^{(c)}$
22	$w_{11} = A_1^{(c)}$	...	$w_{1m} = A_m^{(c)}$
...	...	...	...
23	$A_1^{(c)} = C_{n1} * [-\alpha/2]$	...	$A_m^{(c)} = C_{nm} * [-\alpha/2]$
24	$A_1^{(c)} = T_1^{(c)} * w_{n1} + A_1^{(c)}$	...	$A_m^{(c)} = T_1^{(c)} * w_{nm} + A_1^{(c)}$
25	$A_1^{(c)} = w_{n1} + A_1^{(c)}$	...	$A_m^{(c)} = w_{nm} + A_m^{(c)}$
26	$w_{n1} = A_1^{(c)}$	...	$w_{nm} = A_m^{(c)}$

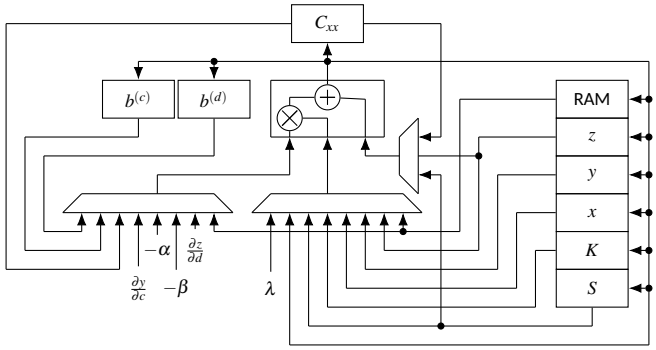


Figure 14 – Data-path of the CCom network node. All the nodes share the same design.

PE in the carousel and j stands for the execution step: every PE adds the value rotating in the carousel to the $w_{ij}x_i$ term and forwards it, resulting in complete $\sum_{i=1}^{n} w_{ij}x_i$ operation.

After this procedure, every PE holds one of the completed MAC results, adds the layer bias value to it, and applies the ReLU activation function, completing the equation (2). Also, it is worth mentioning that the ReLU activation function is computationally light, even in the case of leaky ReLU: it is the multiplication by 1 in the case of a positive value or by a predefined constant otherwise.

The same procedure follows for every layer present in the network. However, one more optimization in CCom architecture targets the execution speed if the execution flow goes from the layer with more PEs towards the narrower one. In that case, all the PEs still perform the calculations, resulting in more than one set of values available for the following layer calculations. For example, if there are n PEs in the currently calculating layer and m PEs in the next layer, and $m < n$, only m PEs need to complete the calculation of the next layer values, resulting in m PEs finally holding the correct data for further execution. This means the values must circulate the entire carousel to be available for all the PEs. In the case of CCom, multiple sets of required values are calculated, and it takes m cycles for all the nodes to receive the data.

However, this means that the count of PEs on the following layers has to be multiple of the PEs in the previous one: the PEs are synchronous and expect the values to arrive in the correct sequence. Therefore, the carousel has to contain *dummy* PEs to satisfy this requirement.

Table 4 provides an example of CCom calculating multiple sets of middle layer values. In this example, there are seven output layer nodes and five middle layer nodes: three dummy nodes $D_1 ... D_3$ must be included to allow two sets of middle layer values to be calculated. However, one of the dummy nodes can be skipped as a communication node in every chain acts the same. Every value has to rotate all the PEs, so it takes a total of ten cycles to complete the calculations, and after this step, every PE receives the input data for the next layer calculations in five cycles in the correct sequence.

Figure 15 illustrates the carousel's state after calculating middle layer values; the figure complements table 4. Every PE can expect five middle layer output values in the successive five clock cycles, although the ordering of received values is not the same for all the PEs. However, different ordering is not an obstacle: controlling FSMs can be parameterized during the synthesis to cope with this.

Table 4 – Performance biased calculation scheme of CCom architecture. Multiple sets of values are calculated to speed up the following execution steps.

D_3	D_2	D_1	N_7	N_6	N_5	N_4	N_3	N_2	N_1
$Y_1^{(1)}$	$Y_5^{(2)}$	$Y_4^{(2)}$	$Y_3^{(2)}$	$Y_2^{(2)}$	$Y_1^{(2)}$	$Y_5^{(1)}$	$Y_4^{(1)}$	$Y_3^{(1)}$	$Y_2^{(1)}$
$Y_2^{(1)}$	$Y_1^{(1)}$	$Y_5^{(2)}$	$Y_4^{(2)}$	$Y_3^{(2)}$	$Y_2^{(2)}$	$Y_1^{(2)}$	$Y_5^{(1)}$	$Y_4^{(1)}$	$Y_3^{(1)}$
$Y_3^{(1)}$	$Y_2^{(1)}$	$Y_1^{(1)}$	$Y_5^{(2)}$	$Y_4^{(2)}$	$Y_3^{(2)}$	$Y_2^{(2)}$	$Y_1^{(2)}$	$Y_5^{(1)}$	$Y_4^{(1)}$
$Y_4^{(1)}$	$Y_3^{(1)}$	$Y_2^{(1)}$	$Y_1^{(1)}$	$Y_5^{(2)}$	$Y_4^{(2)}$	$Y_3^{(2)}$	$Y_2^{(2)}$	$Y_1^{(2)}$	$Y_5^{(1)}$
$Y_5^{(1)}$	$Y_4^{(1)}$	$Y_3^{(1)}$	$Y_2^{(1)}$	$Y_1^{(1)}$	$Y_5^{(2)}$	$Y_4^{(2)}$	$Y_3^{(2)}$	$Y_2^{(2)}$	$Y_1^{(2)}$
$Y_1^{(2)}$	$Y_5^{(1)}$	$Y_4^{(1)}$	$Y_3^{(1)}$	$Y_2^{(1)}$	$Y_1^{(1)}$	$Y_5^{(2)}$	$Y_4^{(2)}$	$Y_3^{(2)}$	$Y_2^{(2)}$
$Y_2^{(2)}$	$Y_1^{(2)}$	$Y_5^{(1)}$	$Y_4^{(1)}$	$Y_3^{(1)}$	$Y_2^{(1)}$	$Y_1^{(1)}$	$Y_5^{(2)}$	$Y_4^{(2)}$	$Y_3^{(2)}$
$Y_3^{(2)}$	$Y_2^{(2)}$	$Y_1^{(2)}$	$Y_5^{(1)}$	$Y_4^{(1)}$	$Y_3^{(1)}$	$Y_2^{(1)}$	$Y_1^{(1)}$	$Y_5^{(2)}$	$Y_4^{(2)}$
$Y_4^{(2)}$	$Y_3^{(2)}$	$Y_2^{(2)}$	$Y_1^{(2)}$	$Y_5^{(1)}$	$Y_4^{(1)}$	$Y_3^{(1)}$	$Y_2^{(1)}$	$Y_1^{(1)}$	$Y_5^{(2)}$
$Y_5^{(2)}$	$Y_4^{(2)}$	$Y_3^{(2)}$	$Y_2^{(2)}$	$Y_1^{(2)}$	$Y_5^{(1)}$	$Y_4^{(1)}$	$Y_3^{(1)}$	$Y_2^{(1)}$	$Y_1^{(1)}$

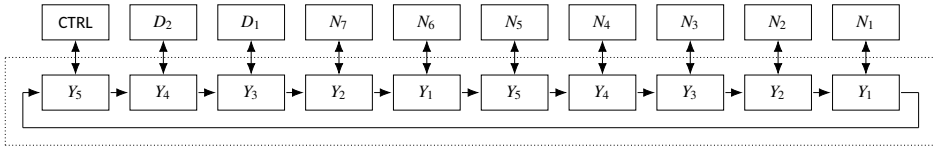


Figure 15 – Data present on carousel nodes after completion of the middle layer values calculations in case of CCom architecture.

2.5.5 Architecture 3: Resource Optimised CAE with efficient Communication (CCom-RO)

The CCom-RO architecture is similar to CCom: they share the carousel-like communication channel architecture (figure 13). Also, the architecture of PEs is combined: there are no dedicated PEs for different layers.

However, as the architecture name suggests, the CCom-RO is a resource-optimized version of CCom, and not all the PEs share the same design. For example, if the CAE network contains m internal and n external layer nodes, and $m < n$, only m PEs include all the registers and, therefore, wider multiplexers to accommodate all the features required to act as a PE belonging to either of those layers.

Figure 16 presents the data-path of the CCom-RO full PE architecture. Every PE is wrapped around the DSP slice, which performs the actual MAC operations. DSP inputs are connected to the multiplexers to select the required signals and registers $b^{(d)}$, $b^{(c)}$, x, y, z, K, and S hold the calculation results. The communication port S_{xx} is required to communicate to the network's other PEs.

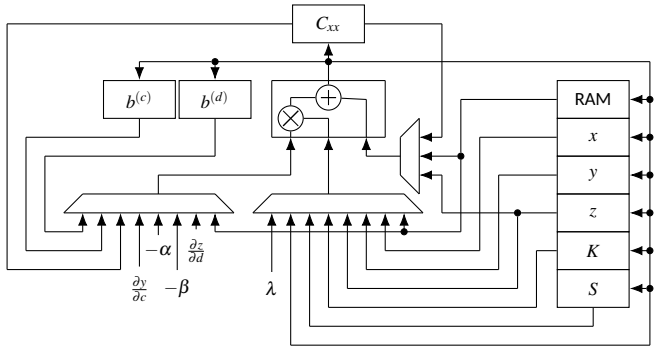


Figure 16 – Data-path of the CCom-RO architecture full PE. This PE can act as it belongs to both internal- or external layers.

As the name of the architecture hints, it tries to optimize the HW resources. Therefore, some of the PEs can act only as part of the external layer. Figure 17 presents this kind of reduced version PE: it misses registers $b^{(c)}$, y, and S. Also, the constant values required only on the middle layer are removed, which yields narrower multiplexers. Otherwise, the PEs carry the same logic compared to the full version: they wrap the DSP slice, add necessary registers for storing the results, and include the communication port C_{xx} to exchange data with the rest of the network.

The optimizations of the PEs abandon the CCom architecture speed-up if the execution flow goes from a wider CAE layer towards the narrower one. Therefore, for a network with m internal and n external layer nodes, and $m < n$, there is a maximum $n - m$ cycles delay for an external layer to start receiving the internal layer output. However, an additional carousel node is present in the real design: a node for communication with the controlling processor. This adds one additional delay step.

Table 5 illustrates the effect of optimizations introduced by the CCom-RO architeture in case of the CAE network with $n = 7$ external- and $m = 5$ middle layer PEs. Only the first m nodes, $N_1 \ldots N_5$, hold the y_j values upon completion of the middle-layer calculation; the remaining $n - m$ nodes have a simplified structure. However, all the PEs must receive all the middle layer values y_j to complete the calculations of the next layer. Therefore, there is an additional delay for PE N_5 before it starts receiving those.

Figure 18 illustrates the carousel's state after calculating middle layer values in the

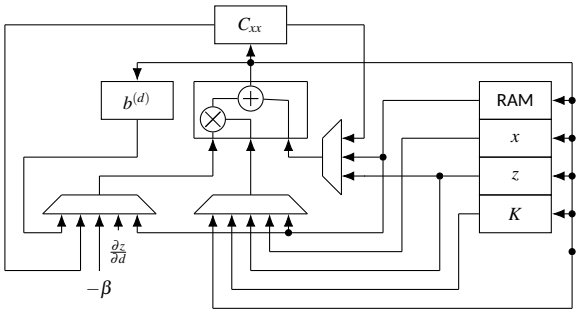


Figure 17 – Data-path of the CCom-RO architecture reduced PE. The reduced version can operate only as an external layer PE.

Table 5 – Resource optimised calculation scheme for CCom-RO architecture. Only one set of internal layer values are calculated. Network nodes $N_6 \ldots N_7$ do not implement all the features required to act as the internal layer node.

$C.$	N_7	N_6	N_5	N_4	N_3	N_2	N_1
Y_1	-	-	-	Y_5	Y_4	Y_3	Y_2
Y_2	Y_1	-	-	-	Y_5	Y_4	Y_3
Y_3	Y_2	Y_1	-	-	-	Y_5	Y_4
Y_4	Y_3	Y_2	Y_1	-	-	-	Y_5
Y_5	Y_4	Y_3	Y_2	Y_1	-	-	-
-	Y_5	Y_4	Y_3	Y_2	Y_1	-	-
-	-	Y_5	Y_4	Y_3	Y_2	Y_1	-
-	-	-	Y_5	Y_4	Y_3	Y_2	Y_1

case of CCom-RO architecture; the figure complements table 5. The node N_5 must idle the longest before it starts receiving the middle layer value for further execution.

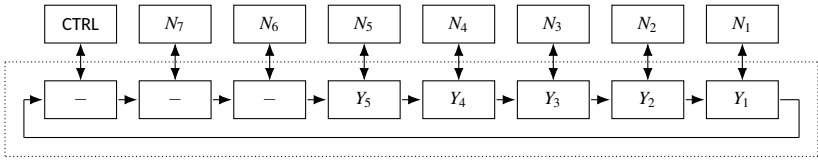


Figure 18 – Data present on carousel nodes after completion of the middle layer values calculations in case of CCom-RO architecture.

2.5.6 Usage of HW Resources

This subsection presents the synthesis results for the proposed CAE architectures, and the work was performed using the Xilinx Vivado Software (SW). All three architectures were designed using the Verilog HDL.

All the PEs wrap a HW DSP slice. Therefore, the total amount of possible PEs is limited to the availability of those. The target platform used for this thesis, Xilinx Zynq 7020, has 220 DSP slices.

Table 6 provides the resources used for synthesizing all three architectures for the target platform. During the synthesis, the count of PEs in the CAE internal layer was fixed to 30, and the number of PEs in the external layer was increased to fill up the target HW. In addition, the clock frequency driving the programmable logic was set to 100MHz.

Table 6 – Maximum network sizes and hardware usage for CAE synthesis targeting Zynq7020 SoC; the size is expressed in FPGA slices.

	ExtNode			MidNode			Chnl	DSP	bRAM
Arch	Count	Size	Total	Count	Size	Total	Size	Count	Count
BL	100	65	6500	30	90	2700	7500	130	15
	ExtNode			Mid+ExtNode			Chnl	DSP	bRAM
CCom	N/A	N/A	N/A	150	105	15750	4377	150	75
CCom-RO	170	75	12750	30	105	3150	1678	200	100

The column *size* expresses the count of consumed HW slices by the design. Each slice on the target platform, Xilinx Zynq 7020, consists of four Look Up Tables (LUTs), eight storage elements, multiplexers, and carry logic. The table cells *Total* under columns *ExtNode* and *MidNode* express the total slices used by either external or middle layer PEs, respectively.

Additionally, the column *chnl* presents the number of slices used by the communication channel: cross-bar switch in case of BL architecture, and carousel type channel in case of CCom and CCom-RO variants. The columns *DSP* and *bRAM* hold the number of DSP slices and block RAMs used by the design.

The data in table 6 suggests that the BL architecture allows the CAE network with the fewest nodes to be synthesized to the target HW, while the number of slices used by a PE is the smallest. However, the communication channel design consumes almost twice the number of slices compared to the CCom. At the same time, the difference is much more radical compared to the CCom-RO version.

The CCom allows the CAE network with more PEs to be synthesized compared to the BL architecture. While the PEs itself consume more HW slices as they combine the functionality from all the network layers, the communication channel is lighter. It has to be noted that the CCom synthesis was configured to include 30 middle layer PEs. However, the design of the external and middle layer PEs is merged. Therefore, the final count of PEs is 30 internal layer PEs and 150 external layer PEs, resulting in 50 more PEs compared to the BL architecture.

Finally, the CCom-RO architecture results in the largest CAE network. Again, an additional explanation is required to interpret the presented numbers correctly: 30 PEs assigned to the column *Mid+ExtNode* also act as the external layer nodes, as the column header suggests. Therefore, the total number of external layer PEs is $170 + 30 = 200$. In addition, the HW slices consumed by the CCom-RO communication channel must also be noted. Middle layer PEs require a pipeline as more data must be simultaneously sent. Therefore, the reduction in required resources is expected because the CCom-RO architecture has fewer of those than the CCom architecture.

2.5.7 Performance Comparison

This subsection provides an execution time comparison of described CAE architectures. The CAE network used for benchmarking consisted of 200 external- and 30 middle-layer nodes, totaling 230 PEs, and the cycle clock was set to 100 MHz. The execution times were acquired using the Xilinx Vivado HDL simulator.

Table 7 provides the execution times for the described architectures: both forward path and training or backpropagation times were measured. In addition, the first table row *Theoretical* provides the timing estimation using the equations provided in subsection 2.5.2.

Table 7 – Execution time of the CAE with 200 external- and 30 middle layer nodes. The clock speed of the designs was set to 100MHz.

Arch	Forward Pass (μs)	Training (μs)
Theoretical	2.3	5.4
BL	13.4	25.7
CCom	2.8	5.9
CCom-RO	6.1	9.5

According to the timing measurements, the fastest architecture is CCom, described in subsection 2.5.4: it is about four times faster compared to the BL architecture (subsection 2.5.3). The communication channel used by different architectures can explain this.

The BL architecture uses the cross-bar switch as the communication channel, and the channel availability synchronizes the PEs. This technique yields less HW resources used per PE (subsection 2.5.6), but at the same time, PEs are forced to idle if data is required or has to be sent, but the channel is not available. In other words, competition for communication resources slows down the overall execution. Further, this kind of resource race is expected: all the PEs start the execution synchronously and require the communication channel simultaneously as they all perform the same amount of calculations.

On the other hand, PEs in CCom architecture are synchronized by design: the state of neighboring PEs is known, and therefore, the availability of the communication resources is expected and guaranteed. This type of PE architecture requires more HW resources compared to the BL version, but the lighter communication channel compensates for it. Furthermore, avoiding the PEs to race for the resources yields fewer idle cycles and faster

execution speeds: data can be sent as soon as it becomes available, and the availability of the communication channel is guaranteed by the synchronous behavior of the PEs.

CCom-RO architecture uses the same kind of communication channel as CCom. However, as the name suggests, CCom-RO is resource optimized; therefore, the reduced execution speed compared to the CCom is expected. Nevertheless, CCom-RO architecture is ~ 1.6 times faster compared to the BL version.

Comparing the presented architectures to the theoretical execution time shows that the CCom architecture almost reaches the target: it can be concluded that PEs in CCom architecture are utilized the best.

2.5.8 Field Test with MNIST database

To test the correct behavior of the described architectures, the MNIST database of handwritten digits was used [48].

For successful operation, the middle layer of the CAE should extract compressed features of the input data, i.e., these features should be sufficient to reproduce the input data in the CAE output.

Furthermore, as the CAE is unsupervised ANN, these compressed features could be used as the input for another unsupervised network. Extracting the features of the input data with a relatively simple CAE network could speed up the training process of the following ANNs. Alternatively, the size of the following network could be reduced after preprocessing.

While the original MNIST database contains 28x28 pixel images, the data was compressed to 14x14 pixels. This compression yields 196 input data bits, suitable for the target platform used in this thesis, Xilinx Zynq-7020 SoC.

The CAE network used in this test was configured to have 196-bit input and output layers to match the size of the compressed MNIST database digits and a 10-bit middle layer. On the target platform, it was possible to synthesize this network using CCom-RO architecture (table 6).

Further, 20 images from the MNIST database were used for the test, while every digit was input 200 times to the CAE, and the network was configured to use 16-bit fixed-point data with 12 fractional bits. In parallel to the HW experiment, MATLAB implementation of the CAE was used to verify the correct behavior.

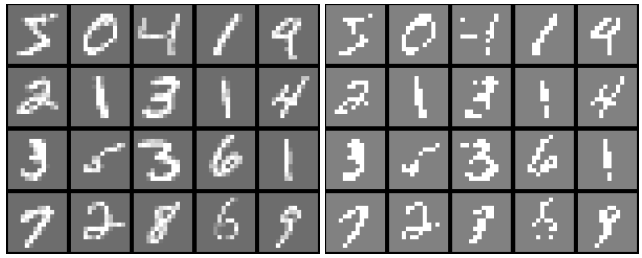
Figure 19 shows the results of the conducted test: the output of the 3-layer 196-10-196 nodes CAE (figure 19b) correlates to the down-scaled 14x14 MNIST database input images (figure 19a), i.e., it can be concluded that the middle layer successfully extracted essential features of the input data.

2.6 Conclusions

This section presents the conclusions of the CAE ANNs presented in previous chapters.

First of all, the motivation of chapter 2 is to provide a CAE implementation which is synthesizable on FPGAs. Further, AE, CAE being one of them, can provide the means of input data filtering for the following ANN: the middle layer of trained CAE eventually contains compressed features of the input signal. This way, the following ANN can either be lighter, containing fewer layers, and PEs or be trained more quickly. Furthermore, CAE is a self-learning, unsupervised network suitable for deploying in fully autonomous environments: data processing can adapt to the changes in the surroundings.

It is important to emphasize that learning in HW is essential for CAE just because it is an unsupervised network. Otherwise, a training process with labeled data would



(a) Input to the CAE, down-scaled 14x14 MNIST images.

(b) Output of the CAE, using 16.12 fixed-point representation and 10 internal layer nodes.

Figure 19 – Operation example of the trained 3-layer 196-10-196 nodes CAE using 16.12 fixed-point representations.

be necessary if it were a supervised network. This kind of separate training could be performed using any HW, not necessarily on resource-constrained devices.

The main contribution of chapter 2 was to present the first implementations of the CAE architectures with full HW based learning. Naturally, this claim has timing limitations to be valid: studies in the field of ANN HW accelerators are very active nowadays.

To be more precise, this thesis's contribution is to provide the first full hardware-based implementation of the CAE [43], comprising hardware-implemented learning. In addition, the provided implementations follow proposals to use shared weights on the input and output layers [44] and fixed point representations for weights and biases [45].

The embedded proposals to share the weight values on different layers and to use the fixed point representation for data representation allows for reducing the HW requirements of the target platform: hyperparameters require less storage space. Therefore, the target HW platform could have fewer resources, yielding more negligible power consumption. This effect should not be overlooked.

3 Multiply-Accumulate Unit for DNN

3.1 Introduction

This chapter takes a step from the CAE HW architecture realization proposals presented in the previous chapter towards versatile execution of the DL algorithms on embedded targets like FPGAs: HW friendly MAC unit suitable to be used as the building block for a DL network is presented and proposed here. Naturally, as DL algorithms and applications based on these algorithms are actively researched nowadays, different proposals addressing various aspects can be found in the literature, including proposals for MAC unit designs and data types in use. However, the novelty of the MAC unit proposed in this thesis is that it makes use of the Triple Fixed-Point (TFxP) data type, which allows direct conversion of the network parameters without retraining the network, and it makes very efficient use of the DSP IP blocks found in FPGAs. On top of that, the method of analyzing the suitability of the replacement data type is also studied. This thesis proposes not to use the network's inference precision for comparison but to analyze and compare the actual output of the network. The problem is that replacing the data type might increase the inference precision of the original network in case the original network was overfitted. The main idea is that the network should behave the same after modifications: it should not perform better or worse.

First, everything is present in the DL architectures for executing the algorithm on conventional computers; why are the research on data types and proposals for HW architectures required at all? Modern computers use the GPUs, and floating point calculations and successful deployments can be found everywhere.

However, even though FPGAs are suitable for designing and deploying parallel architectures and, therefore, similar to the GPUs, can be used as the execution platform for the inherently parallel DL algorithms, there is an essential obstacle: more efficient support for floating point data types is required.

Naturally, floating point calculations are possible on FPGAs, but the realizations would consume much more limited HW resources and energy compared to the fixed point arithmetics. In addition, FPGAs have HW DSP slices available, but again, those slices do not support floating point data. Therefore, the floating-point-based algorithms can not make use of these otherwise available IPs, leave the HW unused, and consume the FPGA logic to rebuild the functionality. Or alternatively, make use of the DSP slices, but spend several to infer a single MAC caused by the floating point requirements.

On the other hand, the total amount of required MAC calculations for a typical DL algorithm must also be analyzed: there would be no real benefit to fine-tuning an algorithm step that is only seldom executed and, therefore, relatively little contributes to the final execution time. Also, additional consumed resources are not critical if only one instance of a specific HW block is used to realize the algorithm. However, neither assumption holds for the MAC unit: MAC operations dominate in executing a DL algorithm, and the algorithms are suitable for parallel execution as well. All the PEs can execute in parallel, and every PE requires a MAC unit. Therefore, performance increases if more MAC units can be inferred in the target HW, provided that the architecture can feed all the units with input data and the MAC units can all execute.

As stated before, the MAC operations dominate while executing a DL network; the total amount can be derived from the equations in use. E.g., equations (1) to (4) define the operations required for the forward pass for the network presented in chapter 2. If we set the number of external layer nodes to 200 and 30 for the middle layer nodes, the total number of MAC operations equals $n*m+m*n = 200*30+30*200 = 12000$. Adding

biases on middle- and external layers requires $200 + 30 = 230$ operations and an equal amount for the ReLU activation function. So we can see that MAC operations form $12000 * 100/12460 \approx 96\%$ of the total. The same holds for the gradient descent; following the equations presented in subsection 2.4.3, it can be seen that the MAC operations indeed dominate.

Also, continuing the analysis of the same previously presented AE network, every PE infers a MAC unit to achieve the node-level parallelism. Therefore, fewer HW resources spent per MAC unit enable more PEs to be inferred, potentially increasing the performance.

So we can conclude that $\approx 96\%$ of total operations are MAC operations for the network presented previously, plus every PE incorporates one of the MAC units. The proposed unit should be moderate in HW utilization numbers and execute fast to avoid becoming the bottleneck.

Further, as the importance of the MAC unit performance and amount of inferred resources per unit is essential and explained above, this chapter continues with the literature review in section 3.2. The literature review brings out various studies on executing DL networks on resource-constraint targets, like FPGAs, while focusing on MAC unit design, the usage of HW resources and retraining requirement.

Section 3.3 analyzes the consequences of using floating point data representation. Also, the section states that fixed point arithmetics suit well for DSP slices available in FPGAs, but the represented values lack the dynamic range. Further, as stated in section 3.3, using the Block Floating-Point (BFP) data representations solves the dynamic range problem, but using a somewhat arbitrary list of possible exponents calls for additional HW resources, like multiplexers to ensure proper alignment after each MAC operation.

Subsection 3.3.1 selects YOLOv2 ([49]), a CNN, as the target network to analyze the range requirements of the data type. In addition to the static analysis of the weight values of the trained network, the chapter also provides a dynamic analysis of the required numerical ranges during the execution: the range of the layer activation values during the inference.

Following the analysis, the exploration of the design space, subsection 3.3.2 proposes TFxP as the data type for the MAC unit. The type reserves two bits for selecting the exponent used for a specific value; therefore, the possible number of required re-alignments of multiplication results is well limited, while the achievable representation range can be adjusted based on the actual DL network in use. Also, the proposed architecture conducts the radix point alignment of the operands in the DSP slice input, ensuring the internal multiplication result inside the DSP slice always has the exact radix point location. This approach allows using the DSP internal accumulator data paths, increasing the performance and conserving required additional resources.

Section 3.4 conducts the simulation of the converted YOLOv2 network, using the COCO dataset for precision assesment ([50]). Considering the simulation environment, MATLAB was selected as a widely used and accepted software package for various calculations. While MATLAB is well-optimized and excels very efficiently in the matrix calculus domain using floating point data, it is not so in the case of using a type not native for the underlying HW like Central Processing Units (CPUs) and GPUs, as described in subsection 3.4.1.

However, MATLAB allows complementing its functionality using C functions. This approach was used to simulate the YOLOv2 DL network converted to use the TFxP data representations in this thesis. Also, the C extension functions use GPU to enhance the simulation time; subsection 3.4.2 describes the C++ and CUDA-based MATLAB convolutional

layer extension with the TFxP data type support.

Subsection 3.4.3 presents the simulation results of the converted YOLOv2 network, using the MATLAB software extended with TFxP support. The overall conclusion is that TFxP can replace the Floating-Point (FP) type without retraining the network. However, the paragraph also states that Mean Average Precision (mAP) is not the best metric to assess the suitability of the TFxP as FP replacement: the network precision might even increase if the original network is not trained perfectly. Instead, Intersection over Union (IOU) is used to measure the similarity of the network outputs: the original network using FP and the one converted to TFxP. This way, the evaluation criterion is not the overall network precision but that the converted network must produce the same output as the original one. For example, if the original network is overfit, the new type should not solve this but retain the original behavior to be considered a suitable replacement.

Further, section 3.5 provides the details about the HW implementation of the TFxP MAC unit, where the proposal makes use of the DSP HW slices available in the target FPGA: Xilinx Zynq SoC. Two flavors of DSPs are considered: DSP48E1, found in the Zynq-7000 series, and newer DSP48E2, present in Zynq Ultrascale devices. The overall idea of those two flavors is the same, regardless of the actual DSP: multiplying two TFxP numbers produces the result with variable radix point location, depending on the ranges of the inputs. This fact disables the internal accumulation path of the DSP slice: it is impossible to directly accumulate values with different radix lengths. Therefore, the paragraph proposes to fix the radix point in the output and alter the input operands accordingly.

Altering the input operands to guarantee the same output radix length raises another issue: multiple possible combinations of connecting the inputs. Subsection 3.5.1 describes the issue and proposes the algorithm that balances the input multiplexers. While the total number of possible combinations is well limited for TFxP data, the method has to be defined to allow automatic HDL code generation. Furthermore, balancing the input multiplexers increases the possible clock frequency for the FPGA by balancing the combinatorial paths and also infers fewer HW resources. The assumption about fewer resources used is that the logic blocks in FPGAs allow building multiplexers with the power of two as the input count; a multiplexer with three inputs consumes the same amount of logic blocks as a multiplexer with four inputs. The fifth input would infer additional HW. Therefore, using two four-input multiplexers infers fewer resources than two multiplexers with five and three inputs.

Further, subsection 3.5.2 describes the MAC unit output formation. First, the acceptable range for the TFxP representation has to be selected; this is achieved by observing the actual magnitude of the output. If possible, the maximum possible fractional part is selected to retain the precision. And vice versa, the fractional part is reduced to fit the output value if required by the output magnitude. In addition, as the output of the DSP slice is 48 bits wide, the under- and overflow check is also possible and conducted.

Finally, chapter 3 is concluded by subsection 3.5.3 where the actual HW usage per single MAC unit is presented.

The main contributions of chapter 3 are:

- Novel Triple Fixed-Point (TFxP) based MAC unit suitable for various neural network architectures, having high numerical precision (comparable to floating point) and very hardware-efficient implementation. This architecture can be directly used in ANN networks (e.g., CNN), which have been trained in software as hardware implementation without retraining the network.

- This chapter also proposes a different evaluation method of the data-type suitability

as the floating-point replacement: instead of analyzing the inference precision, the network outputs are compared directly.

3.2 Literature Review

As there is no doubt in the usefulness of the DL networks, the area is very actively researched. Proposals to use various numeric systems and other strategies to bring the DL algorithms into resource-constraint systems are found in the literature; the goal is to bring down the power consumption of the executing HW. FPGAs are the natural choice for these kinds of tasks: they enable parallelism similar to the GPUs. However, they are more conservative regarding the energy budget and can be more cost-effective depending on the specific make and model. This chapter provides a literature review of such accelerator designs, focusing on data type selection and MAC unit design.

Traditionally, DL algorithms rely on floating-point calculations: this is the natural choice for an application running on CPU or GPU, but not well suited for resource constraint systems. However, it has to be stated that floating-point calculations are indeed possible in embedded targets, such as FPGAs: floating-point calculation units can be synthesized if not available as HW IP blocks. Various proposals are available in the literature to enable and enhance the floating point calculations for such targets: [51, 52, 53, 54, 55, 56, 57, 58, 59, 60].

Also, floating point calculations are proposed for executing DL networks on FPGAs. For example, authors in [61] analyze the power and performance of single-precision floating-point MAC units. However, this reference work does not utilize the DSP blocks available in modern FPGA architectures. Authors in [62] simplify the floating-point format and propose another architecture not based on DSP blocks, leaning more towards Application Specific Integrated Circuit (ASIC) designs. Another work where authors propose a ASIC flavoring design and use a simplified floating-point data type can be found in [63]. Continuing with research proposing modified floating point data types, authors in [64] propose LOCOFloat and target FPGA as the execution platform. However, the proposed modification still yields the inference of HW resources comparable to the floating point-based designs. Further, authors in [65] propose to keep the weight values in floating-point format but use the fixed-point representations for activations. This approach is also more suitable for ASIC designs as it still needs some means of floating-point HW and does not directly fit to the DSP blocks available in FPGAs.

Another research direction is to use integer data types in DL networks; integer-based calculations are less HW hungry and are well suited for FPGAs. For example, authors in [66] experiment with integer representation and conserve approximately half of the required HW resources compared to the floating point-based design. In addition, they also propose the training method: it is only possible to use such severe quantization by retraining the network. Another works using 8-bit integer representations can be found in [67, 68, 69]. Although the architectures are not entirely presented in these works, authors show that the power consumption of the FPGA based solution is reduced by more than an order of magnitude compared to the GPU based counterpart. Also, authors in [70] experiment with the integer data type. Again, the implementation of the HW is not presented, but this work also demonstrates that the floating-point precision is unnecessary and integer types can be used instead. Another similar work can be found in [71], where authors analyze the performance and precision of using integer data types instead of floating point representations. The authors conclude that a 16-bit integer is best suited for that purpose. However, they also retrain the network to maintain accuracy. Moreover, using highly quantized data types, the retraining requirement opens another research direction;

authors in [72, 73, 74, 75, 76] analyze the problems related to low-bit width training and provide proposals for accelerators.

The low bit width of the network parameters and activations provides an interesting optimization possibility: it is possible to pack two multiply operations into a single DSP slice. Authors in [77, 78] provide an example of this kind of design. Also, the same approach is provided in [79, 80], where authors perform two eight-bit multiplications using a single DSP slice.

Further, it is possible to reduce the bit width of the data types even more: another well-studied approach in enabling the DL algorithms to run on FPGAs uses binary- or ternary data representations. These approaches yield low usage of HW resources, but as a drawback, the network requires retraining.

For example, authors in [81] propose a ternary network. The work demonstrates that the inference accuracy of such a DL network can be adequate, but the network has to be retrained, and the authors also implement and propose tools to achieve that. Also, the authors claim that ternary data representation yields severe accuracy loss if the network contains a fully connected layer and propose a fixed-point approach as a subject to study to overcome this.

Another study of a severely quantized DL network can be found in [82], where the data type is reduced to a single bit, binary values. This kind of network also requires retraining. Furthermore, the loss of precision is also encountered. Authors overcome the reduced precision by executing the floating point-based version of the network on CPU if the output of the binary network can not be classified with high enough confidence. This approach boosts the precision, but the network's latency is no longer constant. In addition, authors foresee the FPGA accelerator using more bits for data representation as future work.

A similar deeply quantized network is presented in [83]. Authors achieve a frame rate of 1000 fps while processing the handwritten digits from the MNIST database. This high throughput is expected from a deeply quantized network. However, the presented approach requires special adoption and training of such a network; no direct conversion of a floating-point-based network is possible. Also, the authors discuss time-domain multiplexing of signals to increase the bit-width of the representations.

Also, similar works can be found from [84, 85], all yield efficient hardware but share the same shortcomings: direct conversion is impossible.

Considering the binarization of the network parameters, authors in [86] take an additional step further and apply compression to such a network. They achieve the compression by analyzing the distribution of kernels on DL network layers and select subsets that cover the variation the best. As a result, the authors successfully reduced the network size and improved the performance, with the cost of a slight accuracy drop. However, this research direction requires retraining. In addition, the proposed architecture uses full precision layers as the first and last one, dictating the presence of floating point capable MAC units in the design.

However, these deep quantization techniques are outside this thesis's scope: instead of rebuilding and retraining the network, this thesis searches for a direct replacement for the floating point. Keeping that in mind, it is clear that the dynamic range of binary representations is not sufficient. Sure, the precision can be reclaimed after such substitutions with retraining. Also, binary and ternary representations are undoubtedly conservative with the HW resources. However, retraining the network is a cost that has to be paid. Nevertheless, this research direction should be noted: these works suggest that the high dynamic range of floating point representations is unnecessary, and the format can be

replaced.

Literature also proposes to convert the floating point representations to a different domain while conserving the precision. One work of this kind can be found in [87], where authors investigate the usability of the logarithmic numeric system. The benefit of using logarithmic representations comes from multiplication and division, which convert to addition and subtraction, respectively. However, the drawback of such a conversion is that simple addition and subtraction operations become much more complex. So, this approach can not be considered a good option for MAC unit: simplifying the multiplication at the cost of the complexity of addition operations does not yield a good solution as the resources conserved in one phase are wasted elsewhere. In addition, additional HW is required for conversions. However, this can be avoided if the conversion is performed before the network deployment.

Another work targeting the logarithmic numeric system can be found in [88]. Authors use 8-bit wide BFP data and convert the values to logarithmic scale before multiplications. Further, the values are converted back to the linear scale to avoid the added complexity of the accumulation operation in the logarithmic domain. While the proposed method yields accuracy comparable to the network using the floating point representations, the back-and-forth data conversion from linear to logarithmic scale does not allow the design to use the DSP blocks found in FPGAs.

Another numeric system scientists have studied for DNN is the posit: authors in [89] use the format to implement the MAC unit. They show that the posit numeric system yields better accuracy than the floating point for lower bit widths. However, the usage of HW resources and energy consumption is comparable to the floating point-based solution.

The posit numeric system and related HW architectures are also studied in [90], where authors propose a generator for hardware instantiation. However, the work presented in [91] proposes that the overall cost of the posit HW can even increase compared to the requirements of the floating-point-based systems. Nevertheless, as the authors claim, using posit numeric systems can increase the accuracy of computations.

Another research direction in the literature uses BFP data representations. This approach uses fixed-point values and adjusts the fractional portion upon the actual weight or activation values in the DL network. For example, authors in [92, 93, 94] show that BFP quantization can be used without retraining and still preserve the precision of the network. Further, to decrease quantization error, authors in [95, 96] propose a method of the fractional exponent to effectively use the full range of given mantissa of BFP. However, the detailed HW implementation has not been provided. As another example, authors in [97] experiment with different approximation techniques, including the BFP and also report conserved HW resources compared to the floating-point-based designs while preserving the precision. Also, authors in [98] report the suitability of BFP format for DL networks and propose an accelerator for CNN networks.

Despite floating-point computations being constantly optimized and improved, fixed-point calculations conserve less energy and HW resources. Also, as shown in this review, DL networks do not necessarily need the magnitude of representations provided by the floating-point: proposals to use even single-bit data representations exist. However, the real drawback is that such networks require retraining: deploying an existing network on a resource-constraint system after direct conversion is impossible. Also, as shown, BFP is a promising research direction: authors successfully convert the weight values and perform the inference without sacrificing accuracy. Keeping that in mind, authors in [99] provide Dual Fixed-Point (DFxP) representation which positions between the floating-point and BFP approaches. This format uses a single bit to decide the radix point position. This idea

is developed further in [100], where authors propose dynamically configuring the radix point locations. Further, authors in [101, 102] analyze the resource usage and accuracy of the DFxP calculations: it conserves HW resources compared to the floating-point implementations, and on the other hand, increases the dynamic range of the fixed-point values.

Continuing from here, this thesis proposes TFxP numeric format as a direct replacement for the floating-point data in DL networks: this format adds the third possible radix point located between the highest magnitude and highest precision regions also achievable with the DFxP numbers. Therefore, the precision in the middle range is increased. Also, the proposed architecture uses the DSP blocks found in the FPGAs.

3.3 Data type selection

The data type selection directly impacts the MAC unit and the entire accelerator's overall performance. First of all, FPGAs have DSP slices integrated into the programmable fabric, including those existing hard IP blocks to the MAC architecture gives the best computational performance. However, these blocks are best suited for integer calculations. Floating-Point (FP) arithmetic is possible, but it requires more than one DSP block per MAC unit [103], totaling to less MAC units on a selected FPGA platform and, therefore, reduced possible parallelism.

Although there are FPGAs available with hard FP DSP IP blocks, the integer performance is over 20% better [104]. Also, as only the higher-end devices pack the FP HW, it severely limits selecting a target platform.

Here, we search for a data type that can directly replace the FP parameters without the network retraining. Therefore, binary or ternary quantization techniques do not qualify. On the other hand, going beyond the deep quantization infers much more FPGA HW, and to avoid that, wrapping the MAC unit around the existing DSP slice is a natural choice.

DSP slices integrated into the FPGA are especially suitable for integer calculations. However, integer representations fail to represent fractional numbers, including the ranges $(-1,0)$ and $(0,1)$. Therefore, the multiplication of integer operands can only amplify layer activation values.

Fixed-Point (FxP) representation makes a much better choice for implementing Neural Networks (NNs) on FPGAs. This format fixes the radix point to a specific location and can represent fractional numbers. If x is the value of a memory field interpreted as an integer, and b is the number of FxP fractional bits, equation (36) defines the conversion of x to FP value d.

$$d = x \cdot 2^{-b} \tag{36}$$

Integer math directly applies to summing FxP numbers if operands have the same b amount of fractional bits. For multiplication, the result's fractional part length is the sum of fractional part lengths of operands (equation (37)), which has to be corrected, resulting in shift operation in HW.

$$d_{mult} = (x_1 \cdot 2^{-b}) \cdot (x_2 \cdot 2^{-b}) = x_1 \cdot x_2 \cdot 2^{-2b} \tag{37}$$

The issue with FxP compared to the FP is its much narrower representation range. Authors have addressed this by setting the radix point location b per NN layer or applying another type of partitioning: radix point location is set according to the pre-analysis for different phases. This technique is known as Block Floating-Point (BFP): [94] provides an excellent example of accelerating CNNs using BFP.

BFP format requires determining the common exponent value b_c for a block of data, while the data can be partitioned to match communication patterns or by other means. After partitioning, each data chunk X contains N values with possibly different exponents: equation (38).

$$X = (x_1 \cdot 2^{b_1}, x_2 \cdot 2^{b_2}, \ldots, x_N \cdot 2^{b_N}) \tag{38}$$

The maximum exponent value is used as the common exponent b_p for the entire partition (equation (39)), effectively avoiding overflows.

$$b_p = max_{1 \leq i \leq N}(b_i) \tag{39}$$

For calculations, the values x_i in a partition X are right-shifted by $b_p - b_i$ positions, forming the new data chunk X_p where all the values share the same exponent value b_p: equation (40).

$$X_p = (x_{p1}, x_{p2}, \ldots, x_{pi}) \cdot 2^{b_p} \tag{40}$$

However, BFP conversions do not limit the set of possible exponent values. Each unique exponent b_p requires a different shift operation in MAC output to correct the radix point location, which yields configurable shifters and adds the HW complexity.

The authors in [99] propose DFxP format to extend the range of FxP. This format sacrifices one bit, E, of the representation to select between two possible radix point locations, extending the range of FxP. This technique effectively combines two FP ranges, and the equation (41) defines the value d the representation holds.

$$d = \begin{cases} x \cdot 2^{-b_0} & \text{if } E = 0 \\ x \cdot 2^{-b_1} & \text{if } E = 1 \end{cases} \tag{41}$$

Figure 20 illustrates the DFxP representation, where a_x and b_x are the lengths of the integer and fractional part of ranges. If a value can not fit *RANGE 0*, the precision is reduced, and the value is stored in *RANGE 1*, using range selection bit E to notify that.

However, if DFxP combines ranges with exponent values b_0 and b_1 apart enough, the values just not fitting the *RANGE 0* suffer the most. The precision of values just outside the reach of FxP experience accuracy drop in that case.

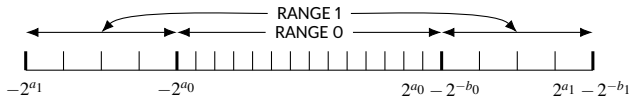


Figure 20 – DFxP representation adds an additional, less precise range to extend the FxP.

To add some flexibility to range selections, authors in [100] take a step towards BFP and use FPGA reconfiguration to adjust the DFxP ranges for different NN layers: they identify NN layers as partitions for BFP which share the same exponent settings. However, reconfiguration decreases the overall throughput, which can be somewhat mitigated using batch execution. Batch execution, on the other hand, increases the latency.

3.3.1 Design Space Exploration

This subsection analyzes a typical DNN to understand a candidate data type's necessary range and precision. The target network is YOLOv2: a CNN comprising 23 convolutional

layers [49]. Although the mentioned network has a newer version available, version two is considered enough for an example in this thesis. Figure 21 presents the YOLOv2 structure.

TYPE	Cnt	SIZE	OUTPUT
Conv	32	3x3	416x416
MPool		2x2/2	208x208
Conv	64	3x3	208x208
MPool		2x2/2	104x104
Conv	128	3x3	104x104
Conv	64	1x1	104x104
Conv	128	3x3	104x104
MPool		2x2/2	52x52
Conv	256	3x3	52x52
Conv	128	1x1	52x52
Conv	256	3x3	52x52
MPool		2x2/2	26x26
Conv	512	3x3	26x26
Conv	256	1x1	26x26
Conv	512	3x3	26x26
Conv	256	1x1	26x26
Conv	512	3x3	26x26
MPool		2x2/2	13x13
Conv	1024	3x3	13x13
Conv	512	1x1	13x13
Conv	1024	3x3	13x13
Conv	512	1x1	13x13
Conv	1024	3x3	13x13
Conv	1024	3x3	13x13
Conv	1024	3x3	13x13
Concat	3072		13x13
Conv	1024	3x3	13x13
Conv	5x(5+C)	1x1	13x13

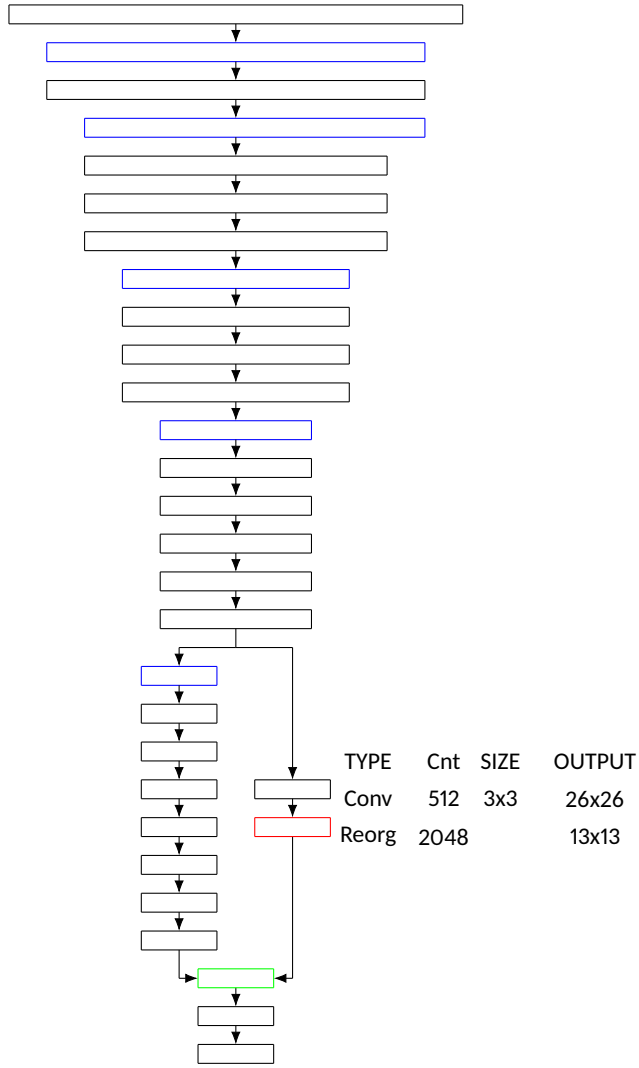


Figure 21 – Structure of the YOLOv2 DL networks: there are total of 23 convolutional layers.

The analysis was performed using MATLAB software. The layer information and pre-trained weight values were loaded using the *Darknet Importer*[1] add-on. For execution, a custom wrapper was developed to collect intermediate activation values.

The first aspect of analyzing was the network weights' range and mean values: the candidate data type should accommodate the values without overflows.

[1] `https://se.mathworks.com/matlabcentral/fileexchange/71277-deep-learning-darknet-importer`

Table 8 presents the results of the analysis: weight values occupy the range $[-18.6, 99.5]$, while the mean value equals 13.7. This hints that 5 integer bits are enough to hold the values without overflows.

Table 8 – Analysis of the weights values of YOLOv2.

	Minimum	Maximum	Median
Weights	-18.6	99.5	13.7

Next, intermediate layer activation values were analyzed using realistic photos and images with all pixels set to black or white. Although the analysis was run in MATLAB, the values were captured right after the convolution operation, just before the activation, batch normalization, or any other operation, which corresponds to the MAC output in HW. Table 9 shows the maximum, minimum, and mean values averaged over all layers. As can be seen, the most comprehensive numerical range is required for the actual photo frame: 7 integer bits are required to avoid overflows. On the other hand, all the activations' mean value equals ≈ 1, requiring only a single-bit integer portion.

Table 9 – Analysis of the layer activation values of YOLOv2.

Input	Minimum	Maximum	Median
Photo	-113.9	106.3	0.7
All white	-57.9	31.6	1.4
All black	-23.1	28.6	1.2

Deviation in range requirements hints that single FxP format can not be used: either precision is lost due to the lack of fractional bits, or overflows are introduced if the length of integer bits field does not suffice. This is especially true for activation values: the mean values show that most calculations benefit from more fractional bits. At the same time, extreme cases require more integer digits to avoid overflows.

3.3.2 Triple Fixed-Point

Data-type suitable for MAC operations in a NN should provide a suitable range to avoid overflows and sufficient precision. Providing suitable precision means that most bits should carry information: exponent value has to be adjustable, just like in FP. However, FP calculations infer a lot of HW resources in FPGAs, and therefore, a better alternative with comparable precision is required.

As the analysis in subsection 3.3.1 shows, the number of integer bits ranges from none to seven in the YOLOv2 network. Most activation values use zero integer bits, while extreme cases require seven bits during the inference using a real photo (table 9).

The results found in the research publications show that BFP can successfully replace the FP format in NNs[94]. And DFxP ([99]) gives promising results too, especially if developing the format towards BFP by adding partition based exponent selection ([100]).

Here, I present the TFxP format. It is related to DFxP and carries a similar idea.

According to the analysis of YOLOv2 CNN, low median values require that all the bits are used for the fractional portion, while the highest values call for seven integer bits.

Most calculations require the lower range, while the higher range is needed to avoid overflows. The values between these two ranges would be converted to the higher range in the DFxP format. TFxP adds a middle range to increase the precision for values in that region: TFxP has three possible exponents to select from for every value.

TFxP representation requires up to four and a minimum of three memory fields: the mode selection field E, the sign bit, integer portion bits, and fractional bits. Depending on the pre-configuration, the integer- or fractional bits may be missing: the integer or fractional portion may occupy all the bits reserved for representing the actual value.

In the following text, the notion $n\_b_0\_b_1\_b_2$ will be used to define the TFxP format, where n stands for the total bit length of the representation, and $b_0 \dots b_2$ define the exponent values for three TFxP ranges.

Table 10 presents the memory allocation for a TFxP format $16\_13\_9\_5$. The first range, identified by the range selection field E value 0, has zero integer bits: all bits are reserved for the fractional portion.

Table 10 – TFxP format 16_13_9_5.

E		**Signed significand**												
15	**14**	**13**	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
0	0	S	fraction												
0	1	S	integer					fraction							
1	0	S	integer								fraction				

Figure 22 gives the visual representation of ranges in TFxP format. Here, a_x and b_x are the bit lengths of integer and fractional portions. The representation has three possible ranges that do not have to be adjacent: exponent values are predefined on design time to suit the system best. The range with the highest exponent value, b_0 in figure 22, has the finest gradation but can represent values with the lowest absolute magnitude. If the range capability does not suffice, the next exponent value b_1 can be selected, which extends the absolute magnitude with the cost of precision. The lowest exponent value, b_2, yields the highest range.

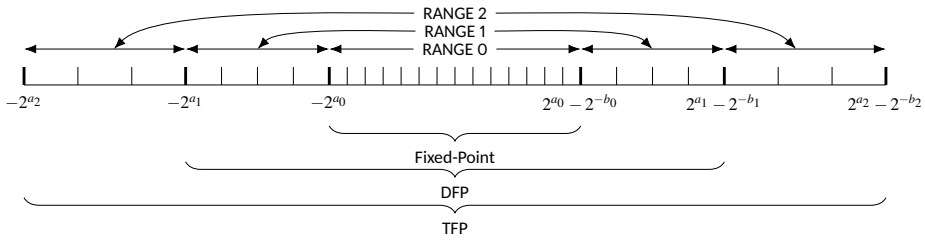


Figure 22 – TFxP representation. Ranges 1 and 2 increase the range while sacrificing the precision.

For backward conversion, from TFxP to FP, equation (42) defines the actual value the TFxP coded memory field holds, where x is the value of combined integer and fractional fields interpreted as an integer, E defines the range, and b is the exponent value of that range.

$$d = \begin{cases} x \cdot 2^{-b_0} & \text{if } E = 0 \\ x \cdot 2^{-b_1} & \text{if } E = 1 \\ x \cdot 2^{-b_2} & \text{if } E = 2 \end{cases} \tag{42}$$

To convert a value to the TFxP format, forward conversion, a suitable target range has to be selected first. Equation (43) presents the range selection criteria: the first range capable of accommodating the value is selected. After determining the target range, the converted value must be truncated to have b_x fractional bits determined by the selected exponent value E and possibly masked to have a maximum of a_x integer bits. Equation (44) presents the width of the integer portion, where n stands for the total bits used by the representation. Overflow can be flagged if the converted range overflows the TFxP highest range.

$$E = \begin{cases} 0 & \text{if } -2^{a_0} < D < 2^{a_0} - 2^{-b_0} \text{ else} \\ 1 & \text{if } -2^{a_1} < D < 2^{a_1} - 2^{-b_1} \text{ else} \\ 2 & \text{if } -2^{a_2} < D < 2^{a_2} - 2^{-b_2} \text{ else} \\ 3 & \text{overflow} \end{cases} \tag{43}$$

$$a_x = n - b_x \tag{44}$$

3.4 Simulation

Apart from the fact that TFxP format extends the range of FxP and improves the precision of the DFxP, the usefulness has to be proved. Studies with BFP [94], DFxP [99] and dynamic DFxP [100] show that the FP format can be replaced, even without retraining of the network. However, these approaches either add HW complexity because of the high amount of possible exponents the system has to support or introduce a time penalty if FPGA reconfiguration scheme is used.

This section provides the simulation results of the YOLOv2 CNN [49] using the new TFxP format for MAC operations.

3.4.1 Environment

The simulation environment used in this work is based on MATLAB software. There was no particular reason to choose MATLAB over other possibilities like Python and its DL libraries. Also, it would have been possible to use the "darknet" software [2] developed especially for the YOLO, but MATLAB provides better visualization and debugging mechanisms than the C console application.

Dataset, the input data to the system under test, is another essential aspect to consider. Although comparing the TFxP versus FP network output is the primary simulation criteria here, a community-improved dataset is better. Firstly, public datasets have a vast amount of data to use. Secondly, results from a common dataset allow a sounder comparison to other works.

In this work, the COCO validation dataset has been used for simulations [50]. This dataset has 40504 pre-annotated images: surely enough to compare the inference accuracy of TFxP based MAC to FP. However, inference accuracy is not the primary evaluation criterion used in this thesis, but the proposed TFxP datatype has to present similar results compared to the FP instead.

[2] `https://github.com/pjreddie/darknet`

Undoubtedly, MATLAB is an accepted software for solving mathematical problems, training, and inferring DNNs among them: it has an additional toolbox available for the DNN analysis. However, FP is the default datatype MATLAB can effectively use. Additionally, FxP can be used, although it slows down the execution. This speed penalty is understandable as the underlying HW does not have native support for FxP operations, and software wrappers must be used.

The support for TFxP is missing in MATLAB, naturally, which can be added using a software layer. However, a possible speed penalty has to be carefully considered here: using a vast amount of testing data, around 40K images from the COCO dataset, would take a long time to execute. E.g., 1 second extra time spent per input frame results in $40504/60/60 \approx 11$ additional hours run time. On the other hand, there are approximately $6.29\text{e}10$ MAC operations involved in the inference of a single input image in YOLOv2 CNN. Therefore, to introduce 1 second additional delay in network output formation, every MAC operation has to execute as little as $1/6.29\text{e}10 = 1.59\text{e}{-11}$ seconds longer. As a result, implementing a straightforward TFxP wrapper yielded an execution time of over half a year for the COCO dataset, and this is only for a single TFxP setting. With run time like this, sweeping the middle range was totally out of the question.

For these kinds of problems, MATLAB supports extending its functionality by mex functions: C/C++ can be used to add support for new data types, like TFxP in this thesis. Also, mex functions can benefit from GPU support: although MATLAB supports GPU directly, direct CUDA [105] programming gives more refined control over the parallel execution.

3.4.2 Triple Fixed-Point Convolutional Layer for MATLAB

The MATLAB TFxP mex extension performs the entire convolution operation: the entire convolutional layer functionality, requiring the TFxP MAC operation, was pushed out from MATLAB and written in C++ and CUDA.

Algorithm 1 presents the high-level structure of the mex extension function. It accepts the layer activation, layer weight values, and TFxP ranges as the inputs, checks the inputs' integrity, and pushes the data to the GPU memory for processing. Input batches are spread over separate CUDA streams to maximize the parallelism.

Algorithm 1: TFxP convolution layer mex function

```
Input: Layer activations[N]
Input: Layer weights
Input: TFxP ranges
Result: Convolution result
SanityCheck()                                          // Check the inputs
toGPU(weights)                                         // Copy to GPU
/* CUDA streams loop                                                  */
for i ← 1 to N do
    toGPU(activations(i))
    padding(activations(i))
    toCols(activations(i))          // flatten 3-D activation in GPU memory
    MAC(activations(i),weights,ranges)
    fromGPU(activations(i))
end
```

The function "toCols()" flatten the input activation data structures to two-dimensional matrixes: 3-D layer activation input is flattened to 2-D in memory to guarantee sequential memory access during the MAC operation. Figure 23 illustrates this operation. The selec-

tion box B, equal to the layer's convolution filter's size, moves across the activation input, and the selected values are transferred to a sequential memory location. The selections outside the input data's bounds are set to zero; this operation is known as "padding."

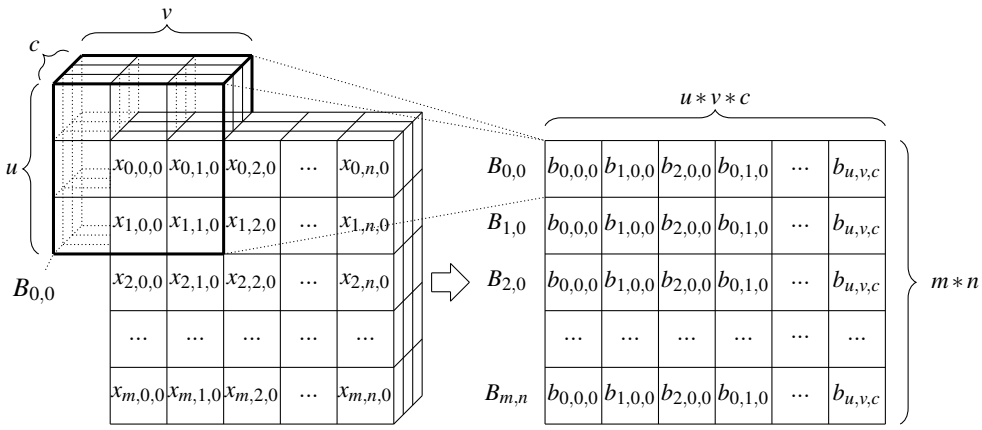


Figure 23 – 3-D layer activation input is flattened to 2-D in memory to guarantee sequential memory access during the MAC operation.

The function "MAC()" performs the actual multiply and accumulate operation: it simulates the execution of the target DSP HW slice. For performance, every output value, the input selected by the box B multiplied by corresponding weight values in the convolutional filters and accumulated, is evaluated by a separate CUDA kernel.

3.4.3 Results

This subsection provides the TFxP based MAC simulation results. In the experiment, YOLOv2 was tested using the COCO evaluation dataset, comprising 40K images.

The network weights were converted from FP to TFxP for testing. No additional training was performed, and the network output was compared to the ground truth: the evaluation dataset. For evaluation, the mAP for output confidence level $0.5 \ldots 0.95$ was used for the assessment. The hypotheses used: if the converted network reaches the same mAP as the original one, the FP can be directly replaced.

The mAP of the original, unconverted network was calculated first. Further, the network weights were converted to DFxP_13_12_5 format, and for comparison, the mAP was found for this configuration too.

Table 11 presents the mAP results of YOLOv2 network inference using the COCO evaluation dataset. As an interesting observation, the mAP is higher if the network was converted to the DFxP_13_12_5 format, i.e., the network under test predicts better if the precision of weights and calculations is decreased.

Table 11 – mAP of YOLOv2 using FP and DFxP_13_12_5 datatypes

	FP	**DFxP**
mAP@[0.5,0.95]	0.277	0.289

A similar simulation was carried out for the YOLOv2 network after converting the weight values to the TFxP format $n\_b_0\_b_1\_b_2$. The notion used here is the same as explained in subsection 3.3.2: n stands for the total bits the format uses, and b_x specifies the length of the fractional part.

Figure 24 presents the inference results of YOLOv2 network with weight values converted to TFxP format. Four different ranges were tested, where the middle range fractional part length, b_1, was swept over all the possible values (b_2, b_0).

Again, the precision of the TFxP converted network outperforms the original, FP based version. Considering the results of DFxP (table 11), this was expected. However, the loss of precision for meaningful middle-range fractional lengths is another phenomenon requiring additional investigation. Here, *meaningful* middle range fractional lengths are considered to be the ones where b_1 is located in the middle region of the range (b_2, b_0): the middle range is considered more meaningful if it adds precision and does not closely follow one of the edge ranges.

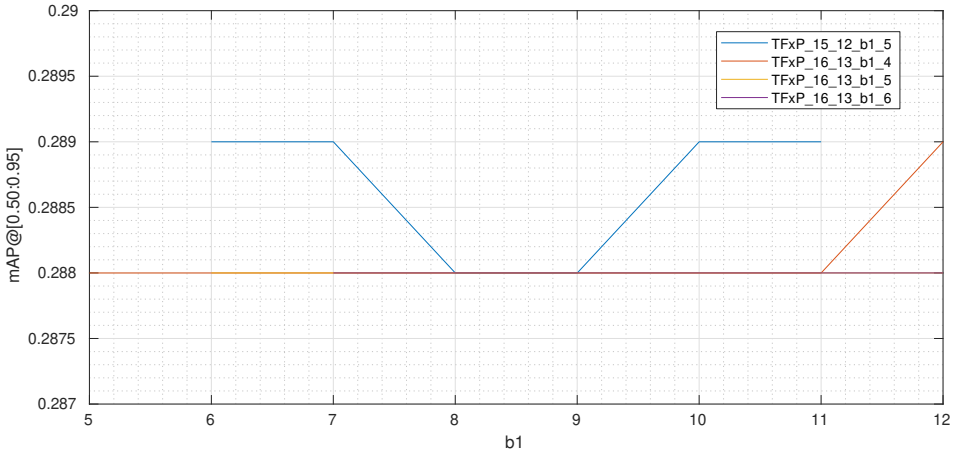


Figure 24 – mAP@[0.5,0.95] of YOLOv2 network with weights and calculations converted to TFxP format $n\_b_0\_b_1\_b_2$, where b_1 is swept for the simulated ranges.

The presented analysis results clearly show that mAP is not suitable to conclude if the TFxP can directly replace the FP. The increase of mAP for reduced precision can not be accurate, but it indicates that the original network could be trained better.

Going further, we can set another criterion to assess the suitability of the replacement type: the proposed type, TFxP, has to produce similar results as the original version. If the original network is overfit, the new type should not solve it, but it has to stay that way. The same holds the other way around: if the network is perfectly trained, the new data type should not alter the situation, and the network should produce the same results.

Therefore, to evaluate the reproducibility of the FP results, the FP results have to be used as the ground truth instead. To achieve this, the converted network's output has to be compared to the original output, not to the ground truth provided by the COCO dataset. For this comparison, the same metric can be used as internally utilized by mAP analysis: IOU.

Figure 25 illustrates the meaning of IOU: IOU of two prediction boxes A and B is the ratio of areas where the boxes intersect, the area surrounded by the green line, and the combination, the union, of those two predictions, the red line surrounded area. The IOU for two perfectly aligned boxes is 1, and the value decreases to zero if the boxes do not

intersect.

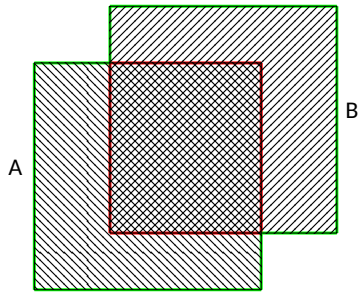


Figure 25 – IOU of two rectangles, A and B, is defined by the ratio of intersection, the double hatched area surrounded by the red line, and union, surrounded by the green line.

Algorithm 2 presents the functionality of the IOU calculation. The algorithm iterates over all the prediction boxes in the ground truth array, the boxes the network predicted using the FP datatype. Then, every ground truth prediction is paired with the best matching counterpart in the test array, the boxes the network predicted using the new replacement datatype. The IOU is calculated and averaged for all these pairs.

Algorithm 2: Average IOU calculation function

```
/* ground truth to be used, FP predictions                          */
Input: truth[N]
/* predictions with the new datatype                                */
Input: test
Result: IOU
IOU = 0
/* Loop over ground truth predictions                               */
for gd ← 1 to N do
    /* Loop over test boxes matching the ground truth box image and
       category                                                      */
    subTest = test.select(img == truth(gd).img && category == truth(gd).category)
    maxIOU = 0
    for bx ← 1 to subTest.elements() do
        tmp = calcIOU(truth(gd), subTest(bx))          // IOU of two boxes
        if tmp > maxIOU then
            maxIOU = tmp
        end
    end
    IOU = IOU + maxIOU
end
IOU = IOU / N
```

Figure 26 presents the analysis results using the proposed metrics. The ordinate shows the IOU of FP versus TFxP over the COCO evaluation dataset, and abscissa sweeps the middle range fractional length b_1 of presented TFxP formats.

The results show that the middle range of TFxP format brings the network predictions closer to the output generated using the FP format. Also, these results support the graph

presented in figure 24: mAP results converge if the new datatype matches the original data better. The effect that the TFxP middle range reduced the mAP results resulted from network training deficiencies and should not be overlooked. Instead, IOU analysis using the FP output as the ground truth describes better the replacement datatype suitability.

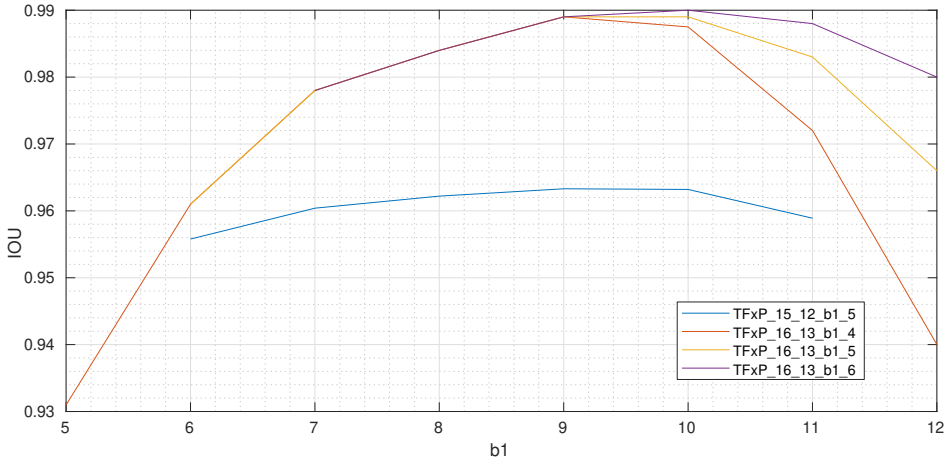


Figure 26 – IOU calculated for FP compared to the TFxP formats $n\_b_0\_b_1\_b_2$, where b_1 is swept.

3.5 HDL design

This section presents the HW design of the TFxP MAC unit. The presented design targets Xilinx FPGAs devices, Xilinx Zynq SoC family, more specifically. Zynq-7000 series Z-7020 and Zynq UltraScale+ ZU9EG were used as the test platforms.

Among other features, Xilinx Zynq devices have DSP slices available in HW. DSP48E1 is used in the Zynq-7000 series, while Zynq UltraScale uses the newer DSP48E2 DSP slice. The core functionality is the same for these two flavors; differences are mentioned in the following text if relevant.

The DSP slices can perform several operations on their inputs, including multiplication and accumulation, and are highly efficient. Therefore, to target power and performance efficiency, the proposed TFxP MAC unit wraps the existing DSP slices.

The Xilinx DSP48E1/2 slices can perform MAC operation using fixed-point numbers or integers. However, the output radix point of a fixed-point result has to be corrected after the multiplication, which is not a problem from the HW point of view: shift operation is enough.

In the case of TFxP, the necessary output correction depends on the ranges of the input operands: multiplying two TFxP numbers, x_t and x_u produces output where the radix point location depends on the modes of input operands (equation (45)).

$$d_{mult} = (x_t \cdot 2^{-b_t}) \cdot (x_u \cdot 2^{-b_u}) = x_t \cdot x_u \cdot 2^{-b_t - b_u} \tag{45}$$

Both of the inputs have three possible fractional lengths. Therefore, the total number of different fractional lengths for output equals *combinations with repetition*. Equation (46) presents the formula to calculate the number of different radix point locations c_o for TFxP MAC output, where n is the number of things to choose from, and r is the number of chosen items. As both of the inputs have three possible radix point locations,

we can choose between three different options, and this selection has to be done for two operands. Therefore, $n = 3$, and $r = 2$.

$$c_o = \frac{(r+n-1)!}{r!(n-1)!} = \frac{(2+3-1)!}{2!(3-1)!} = 6 \tag{46}$$

Figure 27 presents the Xilinx DSP48E1 HW DSP slice data path, where the abstraction level has been chosen to reflect the MAC operation the best. It can multiply inputs A and B and use an internal data path for accumulating multiplication results or add input C.

However, according to equations (45) and (46), the multiplication output of two TFxP numbers, register R_m, can have six different radix point locations after every multiplication cycle, depending on inputs A and B. Further, the register R_m is input to the accumulator, where the second input comes internally from the output register R_o during the MAC operation. However, these registers, R_m and R_o, can have different radix point locations and, therefore, can not be directly accumulated. This means that the DSP internal accumulation path can not be used and must be built externally to match the radix points.

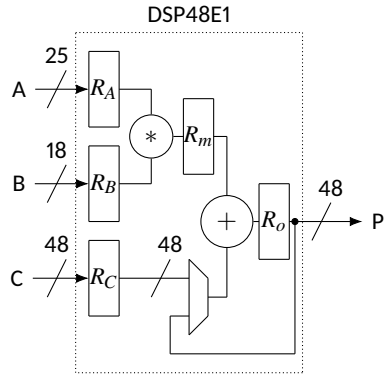


Figure 27 – High level data-path of the Xilinx DSP48E1 HW slice. DSP48E2 has 27-bit A input, compared to 25-bit in the case of DSP48E1.

Building the external DSP accumulation path has an area, speed penalty, or both. Therefore, a different approach was taken to use the internal accumulation possibility: the multiplication output is always guaranteed to have the same radix point even though operands are TFxP numbers.

The output's exponent term $-b_t - b_u$ (equation (45)) has to be fixed to a constant value to achieve that. Naturally, both inputs have their fixed fractional lengths, but we can still add the term b_x to the existing exponents and force the final result to be a constant: equation (47).

$$b_t + b_u + b_x = b_p = const. \tag{47}$$

Adding the term b_x calls an additional shift of one or both of the inputs. This shift can be implemented by a full-featured HW shifter. Alternatively, as the number of possible input combinations is limited (equation (46)), the required set of additional shifts can be implemented by using input multiplexers. Depending on the required fixed shift value of the multiplication output, b_p, an additional shift of b_x bits has to be applied to the input: equation (48).

$$b_x = b_p - b_t - b_u \tag{48}$$

Multiplying two TFxP numbers produces the most extended fractional length in output if both operands belong to the range b_0: the range with the most fractional bits. Therefore, the target output's constant radix point location should be fixed to $2b_0$.

However, there is a maximum for the additional possible shift b_x, and analysis of the DSP inputs and the exact TFxP type defines that. The maximum output shift produced by multiplying two TFxP_16_13_9_5 numbers is $2b_0 = 26$. Similarly, multiplying two range b_2 numbers of the same format produces the shortest fractional length $2b_2 = 10$. All the rest of the input combinations produce the fractional length between these limits. Therefore, the maximum required additional input shift to guarantee the fixed output shift is the difference between these two extremes: $2b_0 - 2b_2 = 16$.

On the other hand, the TFxP_16_13_9_5 format occupies 14 bits in the DSP input: a total of 13 bits plus the sign bit. This means that the DSP A input can be left-shifted by a maximum of $25 - 14 = 11$ bits, and B input can be shifted by $18 - 14 = 4$ bits, and the maximum possible additional shift is 15 bits, 1 bit less than the required maximum. However, this limitation holds for DSP48E1; the E2 version has 27-bit wide A input, allowing $27 - 14 = 13$ bits shift in A input and 17 bits total maximum additional left shift in input.

To implement TFxP_16_13_9_5 MAC using DSP48E1, the maximum shift can be set to 1 bit less than the theoretical limit. Only one input combination out of 6 produces the longest fractional length in output. Furthermore, the chances are that one of the inputs has zero as the least significant bit: losing this last zero does not cause a loss of precision.

Table 12 presents a set of pre-shifts b_x for the TFxP_16_13_9_5 to guarantee the multiplication result with the fixed shift $b_p = 25$. The target output fractional length is $b_p = 25$ bits; therefore, the additional input shift b_x equals the required target minus the existing shifts due to the input modes (equation (48)). The first two columns, M_t and M_u, specify the TFxP ranges for the inputs.

Table 12 – Pre-shift values b_x for TFxP_16_13_9_5 to guarantee the fixed output shift $b_p = 25$ of the multiplication output.

M_t	M_u	b_t	b_u	$b_t + b_u$	b_p	b_x
0	0	13	13	26	25	-1
0	1	13	9	22	25	3
0	2	13	5	18	25	7
1	0	9	13	22	25	3
1	1	9	9	18	25	7
1	2	9	5	14	25	11
2	0	5	13	18	25	7
2	1	5	9	14	25	11
2	2	5	5	10	25	15

Dividing the total required pre-shift b_x between the DSP inputs can be done in various ways. The exact numbers do not matter; the sum of these values is essential. However, the total amount of required different pre-shifts per input specifies the input multiplexers: more unique values require more input channels.

Table 13 presents possible combinations of pre-shifts for MAC inputs to achieve the required b_x: b_{xt} is applied to the MAC input A, and b_{xu} to the input B. The negative pre-shift value -1 results from DSP48E1 input limitations. It is mitigated by right-shifting one of the inputs, depending on the least significant bit of the input x_t.

Figure 28 presents the TFxP MAC unit: Xilinx DSP48E1 slice wrapped by additional input and output logic. The input logic selects additional pre-shift b_x, and the output

Table 13 – Required pre-shift values b_x for TFxP_16_13_9_5 MAC, divided between two inputs.

b_x	b_{xt}	b_{xu}
-1	0/-1 [a]	0/-1 [b]
3	3	0
7	3	4
11	11	0
15	11	4

[a] -1 if $x_t[0] = 0$, 0 otherwise.
[b] -1 if $x_t[0] = 1$, 0 otherwise.

logic converts the result back to TFxP format.

The C input of the MAC should also have the same shift as the multiplication output: $b_p = 25$. Therefore, the value x_v connected to the C input should have an additional shift $b_{xv} = 25 - b_v$, where b_v depends on the selected TFxP range.

Internally, the MAC operation utilizes the full width of the DSP data-path: 48 bits. This feature allows intermediate calculations to overflow; the final MAC result must fit the selected TFxP format. E.g., if b_x is set to 25 bits as for the shift values presented in table 12, the intermediate calculations can use integer portion up to $48 - 25 = 23$ bits.

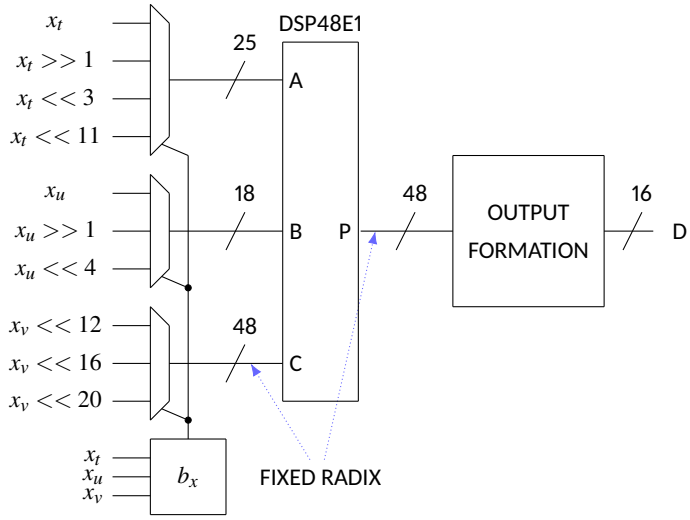


Figure 28 – Xilinx DSP48E1 slice and the wrapper logic to form TFxP_16_13_9_5 MAC unit. Input multiplexers assure the fixed output shift b_p.

3.5.1 Input Multiplexer Selection

The additional required pre-shift b_x can be divided between the DSP inputs: two last columns in table 13 present the values. However, the exact shift applied to an input channel does not matter, but the total amount must match the required value b_x, and multiple combinations can achieve that.

The amount of required different shifts per input defines the input multiplexer: more distinct shifts yield more HW resources. Additionally, the configurable logic blocks in FPGAs are best utilized for specific multiplexer configurations: the possible number of

5:1 multiplexers is the same as 8:1 multiplexers using the same amount of HW, for example. Some of the HW remains unused, wasted if the configuration does not match the HW optimum.

Naturally, the total amount of combinations to consider is limited (equation (46)), but automatic code generation is not possible if manual multiplexer tuning is required, i.e., it would be cumbersome to integrate the proposed MAC unit to a more extensive framework without full automation possibilities.

Therefore, this subsection proposes an algorithm to generate input multiplexers with the fewest channels while addressing the multiplexer balancing between MAC inputs.

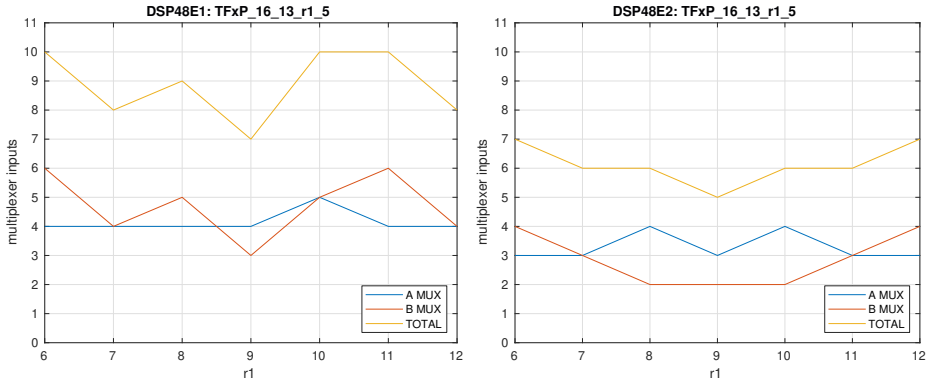
First, the algorithm generates the possible shift value pairs for all input combinations. E.g., if the total additional shift $b_x = 3$, the shift can be assigned to one of the inputs only or split between the inputs. All the possible combinations are registered for the next steps.

Further, the algorithm identifies unique shift values that must be included in the final configuration. In table 13, the last row presents precisely this situation: the total additional shift b_x equals the maximum value possible for the underlying DSP slice. Therefore, both inputs use the maximum shift, and these values must be present in the final HW. Otherwise, the first possible shift values from the set with the least possible combinations available are selected if none of the shift value pairs is the only option for a specific input combination.

The filtering of the possible input shift pairs is performed multiple times. The first two rounds favor either one or another input for unique value selection. Then, the combinations list is traveled in the opposite direction again, setting the priority to one of the inputs.

Finally, the set with the least possible combinations is selected. The final selection is also biased towards more balanced multiplexers.

Figure 29 presents the required number of input multiplexer channels for the TFxP_16_13_r1_5 MAC unit. The DSP48E2 DSP slice has wider A input; therefore, the maximum additional shift b_x can also be larger, allowing more freedom in selecting the possible shift combinations. As a result, the MAC unit wrapping the DSP48E2 slice requires fewer multiplexer channels.



(a) DSP48E1 *(b) DSP48E2*

Figure 29 – Number of input multiplexer channels for TFxP_16_13_r1_5 MAC unit. DSP48E2 has 27 bit wide A input and therefore, requires less channels.

3.5.2 MAC Output Formation

Internally, the DSP slice uses 48-bit registers for multiplication results (figure 27). This means that the intermediate MAC calculations can overflow; only the final results have to fit the target TFxP format. In the case of the convolutional layer, this corresponds to multiplying and accumulating one convolution kernel with the layer input in one position, and table 9 records maximum, minimum, and mean of these values for YOLOv2 CNN.

The common fractional point location b_x is the synthesis parameter for the system and does not depend on MAC inputs, but the selected TFxP format defines it.

Figure 30 presents the 48-bit output of the TFxP_16_13_9_5 MAC unit: the one presented in figure 28. The common shift b_x equals 25, the maximum possible for DSP48E1 in case of TFxP_16_13_9_5 format, i.e., the least 25 bits form the fractional portion of the result. The values of fields R_0, R_1, and R_2 define the final TFxP range, and the field G serves as the over- or underflow guard region.

Also, the R_x fields form the integer portion of the final value. Equation (49) defines the corresponding lengths r_x of these fields, plus the length of the G field, g.

$$\begin{aligned} r_0 &= a_0 \\ r_1 &= a_1 - a_0 \\ r_2 &= a_2 - a_1 \\ g &= 48 - 1 - a_2 - b_x \end{aligned} \tag{49}$$

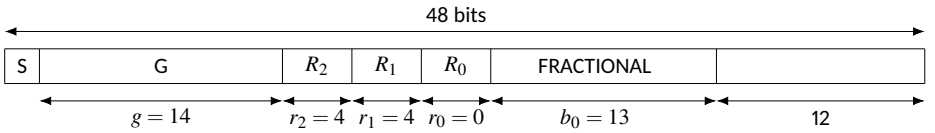


Figure 30 – DSP slice output for TFxP_16_13_9_5 MAC unit. Fields R_0, R_1, and R_2 determine the range. 14-bit G field is used to check the over- and underflows.

The range selection logic uses the R fields, sign bit S, and the guard field G to determine the magnitude of the value: figure 31 presents the high-level schematic diagram. All these fields are tested to be all-ones or all-zeros, and in combination with the sign bit S, the output range is selected: the output multiplexer selects the correct data from the DSP slice output bus P. Also, figure 31 defines the actual signals fed to the output multiplexer using Verilog HDL syntax.

The block *RANGE* in figure 31 is a combinatorial circuit that uses its input signals to drive the output multiplexer, and table 14 presents the truth table of it. The table columns are the following: sign bit S from the DSP output, $G1$ and $G0$, which indicate if the guard field G is all ones or all zeros, and similar fields for R_2 and R_1. The last two columns present the output multiplexer channel number and the TFxP range number for that channel. *OFV* and *UFW* stand for overflow and underflow, respectively.

Figure 32 presents a logic diagram for the output range selection. And due to the automatic adjustment of input operands of the MAC unit, the selected output value can directly be fed to the MAC unit in the following layer. The signal names correspond to the columns in table 14. And the output signals $M_0 \ldots M_2$ correspond to the table column Mux.

3.5.3 Usage of HW Resources

This subsection presents the synthesis results of the TFxP based MAC unit presented in the previous chapters; the Vivado design suite from Xilinx was used to acquire the results.

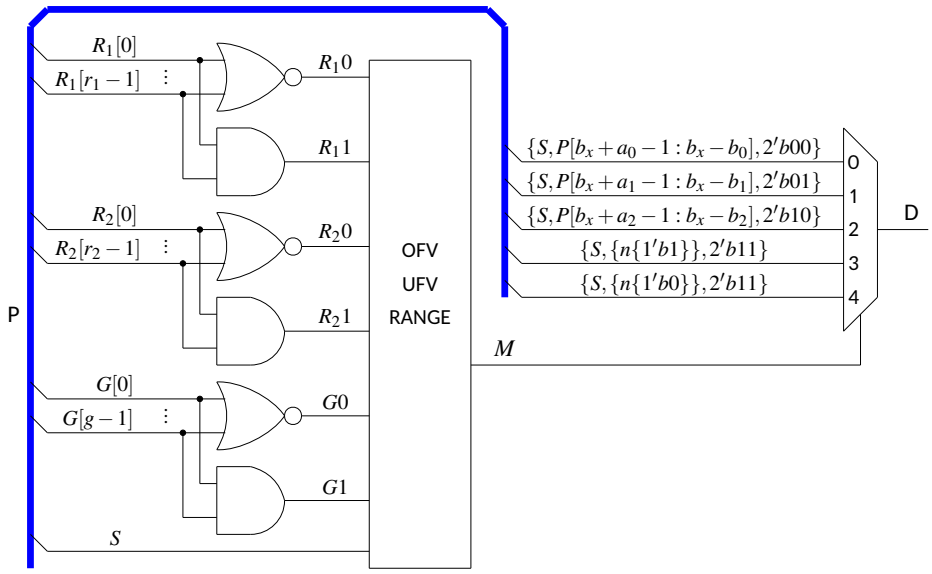


Figure 31 – Formation of the MAC output: fields R_1, R_2, S, and G define the range of the TFxP output.

Table 14 – Truth table of the RANGE selection block presented in figure 31.

S	$G1$	$G0$	$R_2 1$	$R_2 0$	$R_1 1$	$R_1 0$	**Mux**	**Range**
0	x	0	x	x	x	x	3	OFV
0	x	1	x	0	x	x	2	2
0	x	1	x	1	x	0	1	1
0	x	1	x	1	x	1	0	0
1	0	x	x	x	x	x	4	UFV
1	1	x	0	x	x	x	2	2
1	1	x	1	x	0	x	1	1
1	1	x	1	x	1	x	0	0

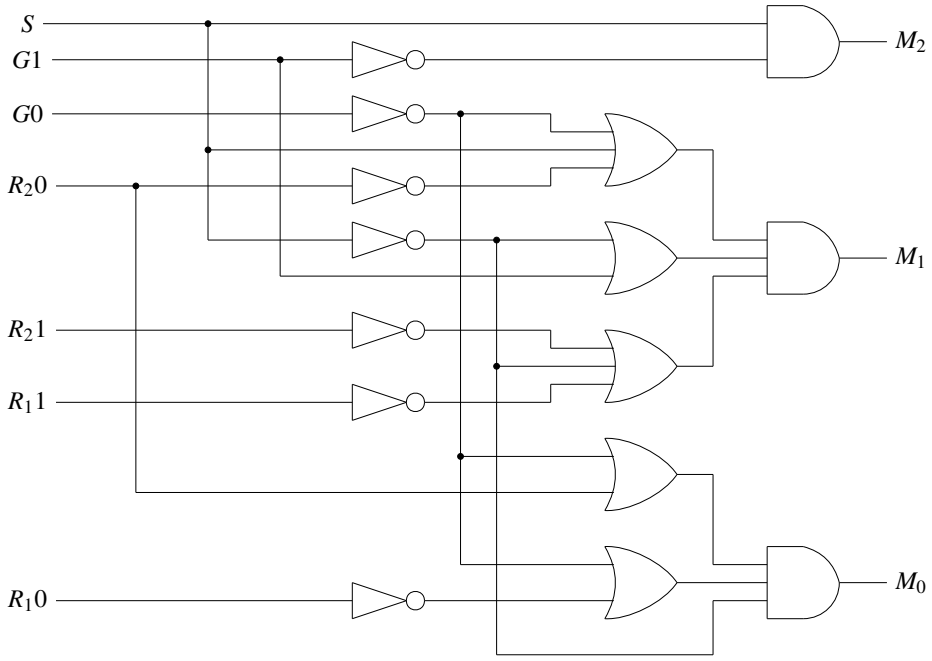


Figure 32 – Logic diagram of the range selections logic block. The output signal M selects the proper range for the output using the multiplexer shown in figure 31. The schematic corresponds to the truth table in table 14.

The synthesis results are summarized in table 15, and the results are presented for the most precise formats found from figure 26. The first three rows present the results for formats where the third TFxP range is changed by a single step. The fourth row presents the results for TFxP format, which is one bit shorter compared to the first three, and the last two rows present the results for the DFxP format for comparison.

The most important comparison is between the DFxP and TFxP formats; it can be concluded that the TFxP does not bring additional resources. The fact that the results show a slight growth of resources for DFxP can be explained by the fact that most of the optimization effort in this thesis addressed TFxP MAC version and the DFxP was designed and synthesized for comparison reasons only. If correctly optimized, the DFxP version would infer the same amount or less HW compared to the TFxP. However, the conclusion holds that the TFxP version is not HW heavy but more precise compared to the DFxP.

Also, the power consumption and maximum clock frequency are the same for all flavors. The maximum clock frequency is restricted by the maximum operating frequency of the DSP slice, i.e., the added wrapper, actual MAC unit design, does not pose any restrictions.

3.6 Conclusions

This section presents the conclusions of the MAC unit proposed in chapter 3.

First, MAC operations form the basis of calculations present in DL networks, reaching over 90% of the entire operations. Therefore, the unit's throughput has to be high to avoid becoming the bottleneck. Also, the HW resources consumed by a single MAC unit have to be carefully considered: more units fit to the target HW enable more concurrent

Table 15 – Inferred HW of DFxP and TFxP formats.

Format	LUTs	Regs	Slices	Power (W)	WNS (ns)	clk (MHz)
TFxP_16_13_9_4	70	20	22	0.142	0.839	393
TFxP_16_13_9_5	69	20	25	0.142	0.837	393
TFxP_16_13_9_6	70	20	26	0.142	0.679	393
TFxP_15_12_9_5	66	19	22	0.140	0.949	393
DFxP_13_12_5	72	18	29	0.133	0.680	393

calculations, i.e., a higher level of parallelism.

Bringing the DL to resource-constraint targets is an active research target nowadays, mainly because there is no doubt about the usefulness of these algorithms. However, the HW execution platforms are mostly restricted to GPUs, and this has been the natural choice for such algorithms requiring massive parallelism. Moreover, due to the extensive usage of GPUs, the floating point is the prevalent data type, but it is unsuitable for resource-constraint targets.

Reducing the precision of the DL network parameters has shown promising results. This has been taken to the extreme: literature has proposals for networks using only single-bit values ([82]), resulting in binarized networks or ternary networks with two-bit data representation ([81]). While these approaches yield effective MAC units, and the recall precision of such networks is undoubtedly acceptable, the entire network has to be retrained: it is impossible to select a well-trained network from the set available for executing on GPUs and seamlessly adapt it for such a deeply quantized target.

The main contribution of chapter 3 is to propose the MAC unit that can be used to directly substitute the floating point-based calculations in DL networks without retraining. The network can be deployed after converting the hyperparameters, like weight values and biases, using the MAC unit and TFxP format proposed in this thesis.

Another important contribution is the proposal to change the evaluation criteria for replacement data type suitability. It has been shown that the network inference precision can increase if the numerical precision is reduced. Therefore, this thesis proposes to use IOU to compare outputs of the converted and original network based on the floating point calculations.

Overall, the analysis and HW design presented in chapter 3 show that the TFxP format can be used as the direct replacement of FP data representation without retraining the network: the novelty presented in this thesis.

4 Conclusions and future work

DL has gained much popularity and can be found in various applications nowadays. Its deployment is so ubiquitous that the application user might not even be aware of the presence of such an algorithm in a system anymore.

It has helped to improve the quality of services in different domains, like analyzing data gathered by medical systems and driver assistance solutions in cars, or has enabled the development of self-driving cars, to mention a few.

The main driver of the growth of popularity of DL algorithms has been the advances in computational power the computers can offer, especially the performance growth of the GPUs. In addition, cloud-based services are also available; there is no need to have a personal computing system available to enjoy the benefits DL has to provide.

As there is no doubt about the usefulness of such algorithms, contemporary research also addresses resource-constrained execution platforms. Also, GPUs are power hungry; executing the algorithm on more conservative targets also has economic benefits.

All this paves the road for FPGAs as the execution platform for the DL. FPGAs consume less power and are suitable for parallel execution just like GPUs but are more suitable for executing fixed-point-based algorithms.

This thesis first introduces and proposes an FPGA based accelerator for CAE network and also includes HW support for the learning phase. CAE is a flavor of an AE and can be used as a self-learning feature extractor or noise filter. Therefore, as CAE is an unsupervised network, continuous learning can be beneficial and help the system cope with environmental drifts during its deployment. To enable this, all three proposed CAE architectures involve HW based learning.

The study results show that the target Xilinx Zynq 7020 SoC can fit 200 PEs in its programmable logic. Also, the performance has been analyzed, and it has been shown that the proposal reaches the theoretical limit derived from analyzing the equations. The field test was performed using the MNIST database of handwritten digits.

Further, the thesis provides a TFxP based MAC unit, aiming not only AEs but various DL networks. There are positive results available in the literature to use the BFP for the DL networks. However, TFxP type carries the information about the radix point location, or the selected range, with it.

First, the thesis analyzed the YOLOv2 network using the proposed TFxP data type. The results suggested that the corresponding TFxP counterparts can directly replace the floating point values. Also, the thesis proposed not to use the converted network's inference precision to assess the conversion's suitability but to compare the output of the converted network to the original one based on floating point calculations. Otherwise, as it has been shown, approximation caused by the data conversion of the network parameters can even improve the inference accuracy. Therefore, comparing the accuracy of the networks is not the best metric to assess the suitability of replacement data type. The network has to behave the same as the original one.

Further, the proposal of the TFxP MAC unit suitable for various DL networks follows. As the range selection setting is embedded into the TFxP type, numbers with different radix point positions, i.e., numbers with different ranges, can simultaneously be loaded into the proposed unit. Properly shifting the input operands still allows using DSP slice internal datapath for accumulation. Also, the thesis implements balancing the input multiplexers, resulting in less inferred HW.

Considering future work, the proposals presented in this thesis can be extended. The proposed MAC unit can be used to build various networks, but the suitable architecture is still to be studied. However, as the MAC unit is conservative with the HW, more of those

can be inferred in the target architecture, enabling more parallel operations. The most intriguing study direction would be incorporating the MAC unit into architecture using dedicated programmable controllers to define the exact behavior. This way, various DL networks can be executed depending on the instructions loaded into the network controllers.

Regarding the proposed CAE, future work should address using the features extracted by the network while allowing it to train itself constantly. For example, these auto-tuning features enable the system to adapt to the installation environment.

List of Figures

List of Tables

References

[1] Q. Rao and J. Frtunikj, "Deep Learning for Self-Driving Cars: Chances and Challenges," in *2018 IEEE/ACM 1st International Workshop on Software Engineering for AI in Autonomous Systems (SEFAIAS)*, pp. 35–38, IEEE Computer Society, 5 2018.

[2] A. N. Aneesh, L. Shine, R. Pradeep, and V. Sajith, "Real-time Traffic Light Detection and Recognition based on Deep RetinaNet for Self Driving Cars," in *2019 2nd International Conference on Intelligent Computing, Instrumentation and Control Technologies (ICICICT)*, pp. 1554–1557, IEEE, 2019.

[3] A. Gogna, A. Majumdar, and R. Ward, "Semi-supervised Stacked Label Consistent Autoencoder for Reconstruction and Analysis of Biomedical Signals," *IEEE Transactions on Biomedical Engineering*, vol. 64, no. 9, pp. 2196–2205, 2017.

[4] M. A. Alsheikh, A. Selim, D. Niyato, L. Doyle, S. Lin, and H.-p. Tan, "Deep Activity Recognition Models with Triaxial Accelerometers," in *AAAI Workshop*, pp. 1–8, 2016.

[5] T. Kautz, B. H. Groh, J. Hannink, U. Jensen, H. Strubberg, and B. M. Eskofier, "Activity recognition in beach volleyball using a Deep Convolutional Neural Network: Leveraging the potential of Deep Learning in sports," *Data Mining and Knowledge Discovery*, vol. 31, no. 6, pp. 1678–1705, 2017.

[6] T. Zabinski, Z. Hajduk, J. Kluska, and L. Gniewek, "FPGA-Embedded Anomaly Detection System for Milling Process," *IEEE Access*, vol. 9, pp. 124059–124069, 2021.

[7] A. G. Ivakhnenko and V. G. Lapa, *Cybernetic Predicting Devices*. CCM Information Corporation, 1965.

[8] K. Fukushima, "Neocognitron: A Self-Organizing Neural Network Model for a Mechanism of Pattern Recognition Unaffected by Shift in Position," *Biological Cybernetics*, vol. 36, pp. 193–202, 1980.

[9] S. Linnainmaa, *The representation of the cumulative rounding error of an algorithm as a Taylor expansion of the local rounding errors*. Master's Thesis (in Finnish), Univ. Helsinki, 1970.

[10] Y. LeCun, B. E. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. E. Hubbard, and L. D. Jackel, "Handwritten Digit Recognition with a Back-Propagation Network," in *Advances in Neural Information Processing Systems 2* (D. S. Touretzky, ed.), pp. 396–404, Morgan-Kaufmann, 1990.

[11] C. Cortes and V. Vapnik, "Support-Vector Networks," *Machine Learning*, vol. 20, pp. 273–297, 9 1995.

[12] J. Kan, Y. Shen, J. Xu, E. Chen, J. Zhu, and V. Chen, "RF Analog Hardware Trojan Detection Through Electromagnetic Side-channel," *IEEE Open Journal of Circuits and Systems*, pp. 1–1, 9 2022.

[13] E. Wang, J. J. Davis, R. Zhao, H. C. Ng, X. Niu, W. Luk, P. Y. Cheung, and G. A. Constantinides, "Deep neural network approximation for custom hardware: Where We've Been, Where We're going," *ACM Computing Surveys*, vol. 52, no. May, pp. 1–39, 2019.

[14] S. Mittal, "A survey of FPGA-based accelerators for convolutional neural networks," *Neural Computing and Applications*, vol. 32, no. 4, pp. 1109–1139, 2020.

[15] M. Kerner, K. Tammemae, J. Raik, and T. Hollstein, "An Efficient FPGA-based Architecture for Contractive Autoencoders," in *2020 IEEE 28th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pp. 230–230, Institute of Electrical and Electronics Engineers Inc., 5 2020.

[16] M. Kerner, K. Tammemae, J. Raik, and T. Hollstein, "Novel Architectures for Contractive Autoencoders with Embedded Learning," in *2020 17th Biennial Baltic Electronics Conference (BEC)*, vol. 2020-October, pp. 1–6, IEEE Computer Society, 10 2020.

[17] J. Redmon and A. Farhadi, "YOLO9000: Better, Faster, Stronger," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2017.

[18] M. Kerner, K. Tammemae, J. Raik, and T. Hollstein, "Triple Fixed-Point MAC Unit for Deep Learning," in *2021 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, vol. 2021-February, pp. 1404–1407, Institute of Electrical and Electronics Engineers Inc., 2 2021.

[19] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *Nature*, vol. 521, pp. 436–444, 5 2015.

[20] J. Schmidhuber, "Deep Learning in neural networks: An overview," *Neural Networks*, vol. 61, pp. 85–117, 2015.

[21] T. Plotz and Y. Guan, "Deep Learning for Human Activity Recognition in Mobile Computing," *Computer*, vol. 51, no. 5, pp. 50–59, 2018.

[22] H. F. Nweke, Y. W. Teh, M. A. Al-garadi, and U. R. Alo, "Deep learning algorithms for human activity recognition using mobile and wearable sensor networks: State of the art and research challenges," *Expert Systems with Applications*, vol. 105, pp. 233–261, 9 2018.

[23] J. Wang, Y. Chen, S. Hao, X. Peng, and L. Hu, "Deep learning for sensor-based activity recognition: A Survey," *Pattern Recognition Letters*, vol. 119, pp. 3–11, 2 2018.

[24] G. E. Hinton and R. R. Salakhutdinov, "Reducing the Dimensionality of Data with Neural Networks," *Science (New York, N.Y.)*, vol. 313, no. July, pp. 504–507, 2006.

[25] C. Zhou and R. C. Paffenroth, "Anomaly Detection with Robust Deep Autoencoders," in *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining - KDD '17*, 2017.

[26] T. Kieu, B. Yang, and C. S. Jensen, "Outlier Detection for Multidimensional Time Series Using Deep Neural Networks," in *2018 19th IEEE International Conference on Mobile Data Management (MDM)*, vol. 2018-June, pp. 125–134, IEEE, 6 2018.

[27] O. K. Oyedotun and D. Aouada, "A Closer Look at Autoencoders for Unsupervised Anomaly Detection," in *ICASSP, IEEE International Conference on Acoustics, Speech and Signal Processing - Proceedings*, vol. 2022-May, pp. 3793–3797, Institute of Electrical and Electronics Engineers Inc., 2022.

[28] D. Del Testa and M. Rossi, "Lightweight Lossy Compression of Biometric Patterns via Denoising Autoencoders," *IEEE Signal Processing Letters*, vol. 22, no. 12, pp. 2304–2308, 2015.

[29] E. Q. Wu, X. Y. Peng, C. Z. Zhang, J. X. Lin, and R. S. Sheng, "Pilots' fatigue status recognition using deep contractive autoencoder network," *IEEE Transactions on Instrumentation and Measurement*, vol. 68, pp. 3907–3919, 10 2019.

[30] X. Zhou, J. Guo, and S. Wang, "Motion Recognition by Using a Stacked Autoencoder-Based Deep Learning Algorithm with Smart Phones," in *Wireless Algorithms, Systems, and Applications* (H. Xu, KuaiZhu, ed.), vol. 9204 of *Lecture Notes in Computer Science*, (Cham), pp. 778–787, Springer International Publishing, 2015.

[31] S. S. Khan and B. Taati, "Detecting unseen falls from wearable devices using channel-wise ensemble of autoencoders," *Expert Systems with Applications*, vol. 87, pp. 280–290, 2017.

[32] L. Wang, "Recognition of human activities using continuous autoencoders with wearable sensors," *Sensors (Switzerland)*, vol. 16, no. 2, 2016.

[33] P. K. Gopalakrishnan, B. Kar, S. K. Bose, M. Roy, and A. Basu, "Live Demonstration: Autoencoder-based Predictive Maintenance for IoT," 2019.

[34] P. Vitolo, G. D. Licciardo, L. Di Benedetto, R. Liguori, A. Rubino, and D. Pau, "Low-Power Anomaly Detection and Classification System based on a Partially Binarized Autoencoder for In-Sensor Computing," in *2021 28th IEEE International Conference on Electronics, Circuits, and Systems, ICECS 2021 - Proceedings*, Institute of Electrical and Electronics Engineers Inc., 2021.

[35] P. Vitolo, A. De Vita, L. D. Benedetto, D. Pau, and G. D. Licciardo, "Low-Power Detection and Classification for In-Sensor Predictive Maintenance Based on Vibration Monitoring," *IEEE Sensors Journal*, vol. 22, pp. 6942–6951, 4 2022.

[36] D. Kim, H. Yang, M. Chung, S. Cho, H. Kim, M. Kim, K. Kim, and E. Kim, "Squeezed Convolutional Variational AutoEncoder for unsupervised anomaly detection in edge device industrial Internet of Things," in *2018 International Conference on Information and Computer Technologies (ICICT)*, pp. 67–71, IEEE, 3 2018.

[37] C. Liu, C. Wang, and J. Luo, "Large-Scale Deep Learning Framework on FPGA for Fingerprint-Based Indoor Localization," *IEEE Access*, vol. 8, pp. 65609–65617, 2020.

[38] N. A. Mohamed and J. R. Cavallaro, "Real-time FPGA-Based Outlier Detection using Autoencoder and LSTM," in *Conference Record - Asilomar Conference on Signals, Systems and Computers*, vol. 2021-October, pp. 1195–1199, IEEE Computer Society, 2021.

[39] M. G. Coutinho, M. F. Torquato, and M. A. Fernandes, "Deep neural network hardware implementation based on stacked sparse autoencoder," *IEEE Access*, vol. 7, pp. 40674–40694, 2019.

[40] L. D. Medus, T. Iakymchuk, J. V. Frances-Villora, M. Bataller-Mompean, and A. Rosado-Munoz, "A Novel Systolic Parallel Hardware Architecture for the FPGA Acceleration of Feedforward Neural Networks," *IEEE Access*, vol. 7, pp. 76084–76103, 2019.

[41] Z. Li, M. Zhu, Y. Zhu, S. Yang, H. Shi, J. Jiang, Q. Wang, and Z. Xing, "FPGA Realization of Stacked Auto-encoder with Three Fully Connected Layers," in *2021 IEEE International Conference on Advances in Electrical Engineering and Computer Applications, AEECA 2021*, pp. 997–1001, Institute of Electrical and Electronics Engineers Inc., 8 2021.

[42] J. Maria, J. Amaro, G. Falcao, and L. A. Alexandre, "Stacked Autoencoders Using Low-Power Accelerated Architectures for Object Recognition in Autonomous Systems," *Neural Processing Letters*, vol. 43, no. 05, pp. 445–458, 2016.

[43] S. Rifai, P. Vincent, X. Muller, X. Glorot, and Y. Bengio, "Contractive auto-encoders: explicit invariance during feature extraction," in *Proceedings of The 28th International Conference on Machine Learning (ICML-11)*, pp. 833–840, 2011.

[44] A. Suzuki, T. Morie, and H. Tamukoh, "FPGA implementation of autoencoders having shared synapse architecture," in *Neural Information Processing*, pp. 231–239, 2016.

[45] J. Jiang, R. Hu, D. Wang, J. Xu, and Y. Dou, "Performance of the fixed-point autoencoder," *Tehnicki vjesnik - Technical Gazette*, vol. 23, no. 02, pp. 77–82, 2016.

[46] V. Nair and G. Hinton, "Rectified Linear Units Improve Restricted Boltzmann Machines," in *Proceedings of the 27th International Conference on Machine Learning*, pp. 807–814, 2010.

[47] K. He, X. Zhang, S. Ren, and J. Sun, "Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification," in *2015 IEEE International Conference on Computer Vision (ICCV)*, pp. 1026–1034, IEEE, 12 2015.

[48] Y. LeCun, C. Cortes, and C. J. Burges, "MNIST handwritten digit database," *ATT Labs*, vol. 2, 2010.

[49] J. Redmon and A. Farhadi, "YOLO9000: Better, faster, stronger," *Proceedings - 30th IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2017*, vol. 2017-Janua, pp. 6517–6525, 2017.

[50] T.-Y. Lin, M. Maire, S. Belongie, J. Hays, P. Perona, D. Ramanan, P. Dollár, and C. L. Zitnick, "Microsoft COCO: Common Objects in Context," in *Computer Vision – ECCV 2014* (D. Fleet, T. Pajdla, B. Schiele, and T. Tuytelaars, eds.), (Cham), pp. 740–755, Springer International Publishing, 2014.

[51] M. Langhammer and B. Pasca, "Design and Implementation of an Embedded FPGA Floating Point DSP Block," tech. rep., Altera, 2014.

[52] A. Ehliar, "Area efficient floating-point adder and multiplier with IEEE-754 compatible semantics," in *Proceedings of the 2014 International Conference on Field-Programmable Technology, FPT 2014*, pp. 131–138, Institute of Electrical and Electronics Engineers Inc., 4 2015.

[53] H. Zhang, D. Chen, and S. B. Ko, "Area- and power-efficient iterative single/double-precision merged floating-point multiplier on FPGA," *IET Computers and Digital Techniques*, vol. 11, pp. 149–158, 7 2017.

[54] K. V. Gowreesrinivas and P. Samundiswary, "Resource efficient single precision floating point multiplier using karatsuba algorithm," *Indonesian Journal of Electrical Engineering and Informatics*, vol. 6, pp. 333–342, 9 2018.

[55] S. Kim and R. A. Rutenbar, "An area-efficient iterative single-precision floating-point multiplier architecture for FPGA," in *Proceedings of the ACM Great Lakes Symposium on VLSI*, *GLSVLSI*, pp. 87–92, Association for Computing Machinery, 5 2019.

[56] M. F. Hassan, K. F. Hussein, and B. Al-Musawi, "Design and implementation of fast floating point units for FPGAs," *Indonesian Journal of Electrical Engineering and Computer Science*, vol. 19, pp. 1480–1489, 9 2020.

[57] S. S. Ganesh, J. J. J. Nesam, and U. Subramaniam, "High Speed Half-Precision Floating-Point Fused Multiply and Add Unit Using DSP Blocks," in *Proceedings - 2020 1st International Conference of Smart Systems and Emerging Technologies*, *SMART-TECH 2020*, pp. 227–230, Institute of Electrical and Electronics Engineers Inc., 11 2020.

[58] A. Panahi, K. Stokke, and D. Andrews, "A Library of FSM-based Floating-Point Arithmetic Functions on FPGAs," *2019 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*, pp. 1–8, 2019.

[59] J. Kralev, "Design of Floating-Point Arithmetic Unit for FPGA with Simulink," in *IEEE EUROCON 2019 -18th International Conference on Smart Technologies*, pp. 1–5, 2019.

[60] V. Krishnan, A. Rajiv, and N. Deborah, "A comparative study on the performance of FPGA implementations of high-speed single-precision binary floating-point multipliers," in *2019 International Conference on Smart Systems and Inventive Technology (ICSSIT)*, pp. 1041–1045, 2019.

[61] G. Jha and E. John, "Performance analysis of single-precision floating-point MAC for deep learning," in *Midwest Symposium on Circuits and Systems*, vol. 2018-August, pp. 885–888, Institute of Electrical and Electronics Engineers Inc., 1 2019.

[62] H. J. Kang, "Short floating-point representation for convolutional neural network inference," *IEICE Electronics Express*, vol. 16, no. 2, pp. 1–11, 2019.

[63] T. Tamble, E.-Y. Yang, Z. Wan, Y. Deng, V. J. Reddi, A. Rush, D. Brooks, and G.-Y. Wei, "Algorithm-Hardware Co-Design of Adaptive Floating-Point Encodings for Resilient Deep Learning Inference," in *2020 57th ACM/IEEE Design Automation Conference (DAC)*, pp. 1–6, 2020.

[64] A. Sanchez, A. de Castro, M. S. Martínez-García, and J. Garrido, "LOCOFloat: A low-cost floating-point format for FPGAs.: Application to HIL simulators," *Electronics*, vol. 9, 1 2020.

[65] L. Lai, N. Suda, and V. Chandra, "Deep Convolutional Neural Network Inference with Floating-point Weights and Fixed-point Activations," 3 2017.

[66] X. Wei, W. Liu, L. Chen, L. Ma, H. Chen, and Y. Zhuang, "FPGA-based hybrid-type implementation of quantized neural networks for remote sensing applications," *Sensors (Switzerland)*, vol. 19, 2 2019.

[67] X. Chen, J. Li, and Y. Zhao, "Hardware Resource and Computational Density Efficient CNN Accelerator Design Based on FPGA," in *2021 IEEE International Conference on Integrated Circuits, Technologies and Applications, ICTA 2021*, pp. 204–205, Institute of Electrical and Electronics Engineers Inc., 2021.

[68] H. S. Lee and J. Wook Jeon, "Accelerating Deep Neural Networks Using FPGAs and ZYNQ," *TENSYMP 2021 - 2021 IEEE Region 10 Symposium*, 8 2021.

[69] V. K. Kodavalla, "Enabling Deep Learning Inferencing in Edge Devices," in *2022 IEEE 3rd Global Conference for Advancement in Technology, GCAT 2022*, Institute of Electrical and Electronics Engineers Inc., 2022.

[70] X. Liu, "Hardware-friendly model compression technique of DNN for edge computing," in *Proceedings - 2021 2nd International Conference on Computing and Data Science, CDS 2021*, pp. 344–355, Institute of Electrical and Electronics Engineers Inc., 2021.

[71] G. Tatar, S. Bayar, and I. Cicek, "Performance Evaluation of Low-Precision Quantized LeNet and ConvNet Neural Networks," *16th International Conference on INnovations in Intelligent SysTems and Applications, INISTA 2022*, 2022.

[72] M. Wang, S. Rasoulinezhad, P. H. Leong, and H. K. So, "NITI: Training Integer Neural Networks Using Integer-Only Arithmetic," *IEEE Transactions on Parallel and Distributed Systems*, vol. 33, pp. 3249–3261, 11 2022.

[73] S. Fox, J. Faraone, D. Boland, K. Vissers, and P. H. Leong, "Training deep neural networks in low-precision with high accuracy using FPGAs," in *Proceedings - 2019 International Conference on Field-Programmable Technology, ICFPT 2019*, vol. 2019-December, pp. 1–9, Institute of Electrical and Electronics Engineers Inc., 12 2019.

[74] C. Su, S. Zhou, L. Feng, and W. Zhang, "Towards high performance low bitwidth training for deep neural networks," *Journal of Semiconductors*, vol. 41, no. 2, 2020.

[75] C. Lammie, W. Xiang, and M. R. Azghadi, "Training Progressively Binarizing Deep Networks using FPGAs," in *2020 IEEE International Symposium on Circuits and Systems (ISCAS)*, pp. 1–5, 2020.

[76] M. Kiyama, M. Amagasaki, and M. Iida, "Deep learning framework with arbitrary numerical precision," *Proceedings - 2019 IEEE 13th International Symposium on Embedded Multicore/Many-Core Systems-on-Chip, MCSoC 2019*, pp. 81–86, 10 2019.

[77] M. Véstias, R. P. Duarte, J. T. De Sousa, and H. Neto, "Parallel Dot-Products for Deep Learning on FPGA," in *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*, pp. 1–4, 2017.

[78] M. P. Véstias, R. P. Duarte, J. T. de Sousa, and H. C. Neto, "A fast and scalable architecture to run convolutional neural networks in low density FPGAs," *Microprocessors and Microsystems*, vol. 77, 2020.

[79] D. Nguyen, D. Kim, and J. Lee, "Double MAC: Doubling the performance of convolutional neural networks on modern FPGAs," in *Proceedings of the 2017 Design, Automation and Test in Europe, DATE 2017*, pp. 890–893, Institute of Electrical and Electronics Engineers Inc., 5 2017.

[80] S. Lee, D. Kim, D. Nguyen, and J. Lee, "Double MAC on a DSP: Boosting the performance of convolutional neural networks on FPGAs," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 38, pp. 888–897, 5 2019.

[81] Y. Chen, K. Zhang, C. Gong, C. Hao, X. Zhang, T. Li, and D. Chen, "T-DLA: An Open-source Deep Learning Accelerator for Ternarized DNN Models on Embedded FPGA," in *Proceedings of IEEE Computer Society Annual Symposium on VLSI*, *ISVLSI*, vol. 2019-July, pp. 13–18, IEEE Computer Society, 7 2019.

[82] S. Amiri, M. Hosseinabady, S. McIntosh-Smith, and J. Nunez-Yanez, "Multi-precision convolutional neural networks on heterogeneous hardware," in *Proceedings of the 2018 Design, Automation and Test in Europe Conference and Exhibition, DATE 2018*, vol. 2018-January, pp. 419–424, Institute of Electrical and Electronics Engineers Inc., 4 2018.

[83] R. Fuchikami and F. Issiki, "Fast and Light-weight Binarized Neural Network Implemented in an FPGA using LUT-based Signal Processing and its Time-domain Extension for Multi-bit Processing," in *2019 IEEE 9th International Conference on Consumer Electronics (ICCE-Berlin)*, pp. 120–121, 2 2019.

[84] A. M. Abdelsalam, A. Elsheikh, S. Chidambaram, J. P. David, and J. M. Langlois, "POLYBiNN: Binary Inference Engine for Neural Networks using Decision Trees," *Journal of Signal Processing Systems*, vol. 92, pp. 95–107, 1 2020.

[85] Q. H. Vo, N. Linh Le, F. Asim, L. W. Kim, and C. S. Hong, "A Deep Learning Accelerator Based on a Streaming Architecture for Binary Neural Networks," *IEEE Access*, vol. 10, pp. 21141–21159, 2022.

[86] Y. Wang, Y. Yang, F. Sun, and A. Yao, "Sub-bit Neural Networks: Learning to Compress and Accelerate Binary Neural Networks," in *Proceedings of the IEEE International Conference on Computer Vision*, pp. 5340–5349, Institute of Electrical and Electronics Engineers Inc., 2021.

[87] M. Haselman, M. Beauchamp, A. Wood, S. Hauck, K. Underwood, and K. Scott Hemmert, "A Comparison of Floating Point and Logarithmic Number Systems for FPGAs," in *13th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'05)*, pp. 181–190, 2005.

[88] C. Ni, J. Lu, J. Lin, and Z. Wang, "LBFP: Logarithmic Block Floating Point Arithmetic for Deep Neural Networks," *Proceedings of 2020 IEEE Asia Pacific Conference on Circuits and Systems, APCCAS 2020*, pp. 201–204, 12 2020.

[89] Z. Carmichael, H. F. Langroudi, C. Khazanov, J. Lillie, J. L. Gustafson, and D. Kudithipudi, "Deep Positron: A Deep Neural Network Using the Posit Number System," in *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pp. 1421–1426, 12 2019.

[90] M. K. Jaiswal and H. K. So, "PACoGen: A Hardware Posit Arithmetic Core Generator," *IEEE Access*, vol. 7, pp. 74586–74601, 2019.

[91] Y. Uguen, L. Forget, and F. De Dinechin, "Evaluating the hardware cost of the posit number system," in *2019 29th International Conference on Field Programmable Logic and Applications (FPL)*, pp. 106–113, 2019.

[92] H. Fan, H. C. Ng, S. Liu, Z. Que, X. Niu, and W. Luk, "Reconfigurable acceleration of 3D-CNNs for human action recognition with block floating-point representation," in *Proceedings - 2018 International Conference on Field-Programmable Logic and Applications, FPL 2018*, pp. 287–294, Institute of Electrical and Electronics Engineers Inc., 11 2018.

[93] H. Fan, G. Wang, M. Ferianc, X. Niu, and W. Luk, "Static Block Floating-Point Quantization for Convolutional Neural Networks on FPGA," in *Proceedings - 2019 International Conference on Field-Programmable Technology, ICFPT 2019*, vol. 2019-December, pp. 28–35, Institute of Electrical and Electronics Engineers Inc., 12 2019.

[94] X. Lian, Z. Liu, Z. Song, J. Dai, W. Zhou, and X. Ji, "High-performance fpga-based cnn accelerator with block-floating-point arithmetic," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 27, no. 8, pp. 1874–1885, 2019.

[95] P.-Y. Tsai, T.-I. Yang, C.-H. Lee, L.-M. Chen, and S.-Y. Lee, "Design of a Tunable Block Floating-Point Quantizer with Fractional Exponent," in *2019 IEEE International Symposium on Circuits and Systems (ISCAS)*, pp. 1–5, 2019.

[96] P. Y. Tsai, T. I. Yang, C. H. Lee, L. M. Chen, and S. Y. Lee, "Tunable Block Floating-Point Quantizer with Fractional Exponent for Compressing Non-Uniformly Distributed Signals," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 67, pp. 1245–1254, 4 2020.

[97] G. Lentaris, G. Chatzitsompanis, V. Leon, K. Pekmestzi, and D. Soudris, "Combining arithmetic approximation techniques for improved CNN circuit design," in *ICECS 2020 - 27th IEEE International Conference on Electronics, Circuits and Systems, Proceedings*, Institute of Electrical and Electronics Engineers Inc., 11 2020.

[98] H. Zhang, Z. Liu, G. Zhang, J. Dai, X. Lian, W. Zhou, and X. Ji, "A Block-Floating-Point Arithmetic Based FPGA Accelerator for Convolutional Neural Networks," in *2019 IEEE Global Conference on Signal and Information Processing (GlobalSIP)*, pp. 1–5, 2019.

[99] C. Te Ewe, P. Y. K. Cheung, and G. A. Constantinides, "LNCS 3203 - Dual Fixed-Point: An Efficient Alternative to Floating-Point Computation," in *Field Programmable Logic and Application*, pp. 200–208, Springer Berlin Heidelberg, 2004.

[100] G. A. Vera, M. Pattichis, and J. Lyke, "A dynamic dual fixed-point arithmetic architecture for FPGAs," *International Journal of Reconfigurable Computing*, 2011.

[101] A. Jacoby and D. Llamocca, "Dual fixed-point CORDIC processor: Architecture and FPGA implementation," in *2016 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*, pp. 1–8, 2016.

[102] A. Jacoby and D. Llamocca, "Dynamic dual fixed-point CORDIC implementation," in *2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pp. 235–240, 2017.

[103] Xilinx, "Performance and Resource Utilization for Floating-point." `https://www.xilinx.com/support/documentation/ip_documentation/ru/floating-point.html`.

[104] D. L. N. Hettiarachchi, V. S. P. Davuluru, and E. J. Balster, “Integer vs. Floating-Point Processing on Modern FPGA Technology,” in *2020 10th Annual Computing and Communication Workshop and Conference, CCWC 2020*, pp. 606–612, Institute of Electrical and Electronics Engineers Inc., 2020.

[105] J. Nickolls, I. a. N. Buck, and M. Garland, “Scalable Parallel Programming,” *Queue*, vol. 6, no. April, pp. 40–53, 2008.

Acknowledgements

Firstly, and most importantly, I want to express my gratitude to my supervisors for all the support I have received during my Ph.D. studies.

Also, I would like to thank my family for allowing me to participate in that journey, indeed, it took me effort and time.

Abstract
Novel Neural Network Accelerator Architectures for FPGAs

Artificial intelligence and Deep Learning (DL) networks have gone through a long journey and proved useful in many application domains: the deployment is so ubiquitous nowadays that the system users are often unaware of the presence of such algorithms.

Although the earliest DL networks date back to 1965, the first proposals did not enjoy direct success: other algorithms, like Support Vector Machines (SVMs), were used instead. Insufficient computational power was the main obstacle for DL, and the problem was solved by the Graphical Processing Units (GPUs).

This thesis focuses on problems and challenges running otherwise proven and accepted DL algorithms on embedded resource-constraint targets, Field Programmable Gate Arrays (FPGAs). There are two main questions stated and answered in this thesis. First, is it feasible to execute the backpropagation directly in hardware in case of unsupervised networks? And second, is it possible to replace the Floating-Point (FP) data in these algorithms without retraining the network and construct a hardware-efficient Processing Element (PE) for DL algorithms based on the new replacement data type?

The thesis proposes three architectures, baseline (BL), CAE with efficient Communication (CCom), and Resource Optimised CAE with efficient Communication (CCom-RO) for Contractive Autoencoder (CAE) DL network to answer the first question. CAE is a flavor of Autoencoder (AE) and uses unsupervised training process. Therefore, as CAE does not require labeled data, the training process can execute throughout the system's entire lifespan to adapt to environmental drifts.

The proposed architectures' novelty is that the backpropagation training process is included in Hardware (HW).

These three proposed architectures differ in communication scheme and optimization. Otherwise, all three architectures include the HW based training and use the node level parallelism: each network node executes as an individual PE. Furthermore, each PE wraps a Digital Signal Processing (DSP) slice available in the target platform: Xilinx Zynq-7020 System On Chip (SoC), resulting in efficient design.

The proper functionality of the architectures is proven using the MNIST digital database of handwritten digits. Also, synthesis results and performance figures of the proposed architectures are presented.

Next, the thesis looks further and searches for data type suitable to replace floating point representations in DL networks. FP data is suitable for GPUs, but realizations on FPGAs infer a lot of resources compared to accelerators for integer-like data arithmetics.

In literature, there are proposals to use even binary representations for weight values in DL networks, i.e., it can be concluded that the numerical range provided by floating point values is optional for successful deployment. However, the drawback of such deep quantization is that the network has to be retrained.

As a result of the analysis of the YOLOv2 Convolutional Neural Network (CNN), the thesis provides a novel Triple Fixed-Point (TFxP) representation. The proposed format uses two bits to select the radix point location, giving an adjustable dynamic range to a conventional fixed point representation. The simulation results show that the precision of the YOLOv2 network did not change after converting floating point weight values to the proposed type. And the precision was preserved without retraining.

Also, the thesis proposes not to analyze the inference precision of the network after the data type conversion but rather to compare the converted and original network outputs. This is because approximation of the calculations and weight values can even improve the precision, as shown in the thesis, and therefore, yield wrong conclusions re-

garding the suitability of the conversion. The network has to perform the same after conversion; the precision should not decrease, and at the same time, it should not increase either.

The simulation results of the converted YOLOv2 network are acquired using MATLAB. However, MATLAB does support calculations using the proposed TFxP values. Therefore, this thesis also implements C and CUDA MATLAB extensions to add the necessary support.

Further, the thesis proposes a Multiply-Accumulate (MAC) unit based on the novel TFxP data type. The proposed architecture accepts TFxP inputs with different ranges and internally adjusts the values to guarantee the exact pre-defined radix point location for multiplication results. This allows the use of the internal accumulation paths of the DSP slice, resulting in fewer additional external HW required to complete the design. Also, synthesis and performance results are presented.

The presented architectures can be extended as future work. For example, the proposed CAE implementation can be cascaded with another type of DL algorithm and use the extracted features as the input to the second processing stage. Also, the TFxP based MAC unit calls for further research: the exact architecture of possible DL execution platform has to be designed, using the proposed MAC unit as the building block.

Kokkuvõte
Uudsed närvivõrkude kiirendite arhitektuurid FPGAdele

Tehisintellekti ja süvaõppe algoritmid on läbi teinud pika arengu ja tõestanud enda kasulikkust erinevates valdkondades: tänapäeval leiab nende rakendusi kõikvõimalikest erinevatest süsteemidest ja süsteemi kasutajad ei ole nende algoritmide kasutamisest enam tihti teadlikud.

Süvaõppe algoritmidel on pikk ajalugu, esmased teadustööd on avaldatud juba 1965 aastal. Sellele vaatamata ei leidnud algoritmid esmalt kuigivõrd kasutust ja konkureerivad algoritmid nagu näiteks tugivektor-masinad leidsid ennemini rakendust. Põhiliseks takistuseks süvaõppe algoritmide laialdasemaks kasutuseks võib pidada ebapiisavat arvutusjõudlust. See probleem sai aga lahenduse seoses graafikaprotsessorite kasutusele võtuga.

Käesolev doktoritöö keskendub süvaõppe algoritmide rakendamisele piiratud jõudlusega seadmetel nagu näiteks väliprogrammeeritav loogika (FPGA). Käesolevas töös püstitatakse ja pakutakse vastus kahele küsimusele: kas oleks võimalik ja ka otstarbekas konstrueerida riistavaline kiirendi süvaõppe algoritmidele, mis tuginevad juhendamata õppimisele (unsupervised learning) ja kas süvaõppe algoritmides oleks üldisemalt võimalik ilma algoritmi uuesti õpetamata asendada ujukoma andmed mõne sobivama andmetüübiga, mis võimaldaks konstrueerida riistvaraliselt säästlikuma kiirendi.

Esiteks pakutakse selles töös välja kolm arhitektuuri "lepingulise autoenkoodri"(contractive autoencoder) süvaõppe algoritmi realisatsiooniks. See algoritm on oma olemuselt autoenkooder ja võimaldab oma olemuselt juhendamata õpetamise (unsupervised leaning) protsessi. Ehk siis teisisõnu see algoritm ei nõua treenimiseks eelnevalt ettevalmistatud ja sildistatud andmeid ja seetõttu võib õppe protsess kesta ka kogu süsteemi kasutusaja jooksul. Järjepidev algoritmi treenimine võimaldab süsteemil näiteks kohanduda keskkonna muutustega.

Väljapakutud arhitektuuride uudsus seisneb just nimelt asjaolus, et ka algoritmi treenimise protsess on teostatud täielikult riistvaraliselt, mis siis omakorda võimaldab algoritmil järjepidevalt õppida nagu see põhimõtteliselt taolise algoritmi puhul võimalik on.

Kolme erineva pakutud realisatsiooni erinevus seisneb kommunikatsiooni kanalis ja optimeerimise meetodites. Muus osas on arhitektuurid samaväärsed. Kõik kolm varianti sisaldavad riistvara põhist treenimist ja samuti on kõigi kolme arhitektuuri paralleel töötluse põhimõtted samad. Samuti, kõigi kolme arhitektuuri puhul sisaldab iga arvutusüksus ühte digitaalset signaaliprotsessorit, mis on töös kasutatud testplatvormil, Xilinx Zynq-7020, realiseeritud riistvaras ja kasutamiseks valmis.

Korrektse funktsionaalsuse hindamiseks kasutatakse antud töös MNIST andmebaasi käsikirjalistest numbritest. Lisaks esitatakse töös kõikide arhitektuuride kohta riistvara sünteesi tulemused ja jõudlusnäitajad.

Edasi võtab käesolev töö laiema vaate ja analüüsib võimalusi süvaõppe algoritmides ujukoma arvutuste asendamiseks. Kirjandusest leiab edukaid rakendusi, kus ujukoma formaat on asendatud isegi binaarsete väärtustega, seega võib järeldada, et see on on kindlasti võimalik ja ujukoma esituste dünaamiline diapasoon ei ole ilmtingimata vajalik. Vaatamata edukatele kasetustele nõuavad taoliselt konverteeritud algoritmid uuesti õpetamist.

Reaalseks analüüsiks kasutatakse selles töös YOLOv2 konvolutsiooni algoritmi, mille tulemusena pakutakse välja uudne formaat: kolme erineva täpsusega püsipunkt esitus, kus täpne koma asukoht määratakse formaadis sisalduvate kahe lisabiti abil. Simulatsiooni tulemused näitavad, et taoliselt ümber konverteeritud YOLOv2 algoritm käitub sarnaselt kasutades kas siis ujukoma arvutusi või töös välja pakutud uudset formaati.

Lisaks pakub töö olulise kriteeriumi andmetüübi sobivuse analüüsiks. Nimelt, nagu on ka töös näidatud, võivad uuest andmetüübist tingitud lähendused kohati isegi originaal

algoritmi täpsust parandada, millest võib teha uue andmetüübi sobivuse kohta valesid järeldusi. Seetõttu antud töös võrreldakse teisendatud algoritmi väljundit originaalrealisatsiooniga: sama sisendi korral peavad väljundid olema võimalikult sarnased ja kogu algoritmi täpsus ei tohiks ei suureneda ega kahaneda.

Algoritmide simulatsioonid on antud töös läbi viidud MATLAB tarkvaraga. Kuivõrd aga MATLAB ei toeta arvutusi kasutades töös välja pakutud uudset formaati, siis on antud doktoritöö raames ka välja töötatud C ja CUDA programmeerimiskeeltel baseeruv täiendusmoodul MATLAB-ile.

Lõpetuseks pakutakse käesolevas doktoritöös välja uudsel andmetüübil baseeruv korrutamis-liitmis seade (MAC). Esitatud realisatsioon aktsepteerib sisenditena väärtusi, mis võivad kasutada erinevaid täpsusi ehk siis väärtusi, milledes koma koht on erineval positsioonil nagu välja pakutud formaat võimaldab. Tagamaks, et korrutamise tulemustes oleks koma koht eelnevalt määratud positsioonil mistahes sisendite kombinatsiooni puhul, nihutatakse sisendväärtuseid vastavalt. Taoline optimeerimine võimaldab kasutada riistvaras juba leiduva konventsionaalse korrutus-liitmis seadme sisemisi struktuure ja vältida lisatava riistvara hulka. Samuti esitatakse uudse korrutus-liitmis seadme riistvara sünteesi- ja jõudlusnäitajad.

Antud doktoritöös välja pakutud lahendusi saab ja tuleks järgnevates töödes edasi arendada. Esiteks, pakutud autoenkooderit saab kasutada sisend andmetest oluliste tunnuste eraldamiseks ja saadud andmeid omakorda kasutada sisendina järgnevale süvaõppe algoritmile. Samuti tuleb edasi arendada pakutud korrutamis-liitmis seadme kasutusvõimalusi, uurides erinevaid võimalikke arhitektuure, mis kasutaks välja pakutud seadet ja andmetüüpi.

Appendix 1

I

M. Kerner, K. Tammemae, J. Raik, and T. Hollstein, “An Efficient FPGA-based Architecture for Contractive Autoencoders,” in *2020 IEEE 28th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pp. 230–230, Institute of Electrical and Electronics Engineers Inc., 5 2020

An Efficient FPGA-based Architecture for Contractive Autoencoders

Madis Kerner*, Kalle Tammemäe*, Jaan Raik*, Thomas Hollstein*†
*Tallinn University of Technology, Tallinn, Estonia
†Frankfurt University of Applied Sciences, Frankfurt, Germany
Email: madis.kerner@taltech.ee, kalle.tammemae@taltech.ee, jaan.raik@taltech.ee, hollstein@fb2.fra-uas.de

***Abstract*—Deep learning neural networks have gained much attention in recent research. Excellent results in various domains have proved the usefulness of such algorithms. However, training a deep learning network requires substantial computational effort; therefore, resource-constrained systems like edge devices in the IoT domain still lack full implementations, and training of the network is offloaded to the cloud. Online or unsupervised training of the network, on the other hand, is often a must if the system has to adjust to possible drift of the environment parameters or there is not enough data available initially. This paper proposes the first Xilinx Zynq FPGA (Field Programmable Gate Array) based implementation of the contractive autoencoder (CAE), including training of the network.**

I. Introduction

Deep learning (DL) algorithms have been proved to be useful in various domains: image recognition, natural language translation, human activity recognition, and anomaly detection [1], [2], [3]. However, the current state-of-the-art solutions rely on graphical processing units and other general-purpose hardware accelerators.

The DL algorithms extract the essential features of the input signal automatically; this enables automatic learning and increases the DL modeling capabilities [4].

Before the deployment, DL algorithms need training, which requires substantial computational power. Therefore, the network is either trained offline, or using the cloud [5].

The broader focus of this work is related to the unsupervised DL algorithms and implementations on resource-constrained systems. One class of this kind of methods are autoencoders, which reproduce the input signal to its output. The middle layer of an autoencoder contains compressed features [6], which can be used for different purposes, like data-compression [7].

[8] describes the framework for FPGA based forward pass execution of various DL networks but does not include the training, which has to be carried out separately.

Considering autoencoders, [9] provides the study of an FPGA based sparse stacked autoencoder, but again, it does lack the training.

Using high-level synthesis is another approach found in the literature; [10] provides the solution to train stacked autoencoders. However, the proposed solution lacks the training speed and the contraction term.

The main contribution of this work is to provide the first hardware-based implementation of the Contractive Autoencoder (CAE) [11]. Also, this paper follows proposals to use shared weights on the input and output layers [12] and fixed-point representations for weights and biases [13].

The proposed architecture uses node-level parallelism. For back-propagation, additional parallelism was achieved by maximally reusing the computational results.

The functionality of the solution was verified using the downscaled MNIST dataset [14]. The $38\mu s$ total execution time for a forward pass and training yields to a maximum of 26KS input rate.

References

[1] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *Nature*, vol. 521, pp. 436–444, 5 2015.

[2] J. Schmidhuber, "Deep Learning in neural networks: An overview," *Neural Networks*, vol. 61, pp. 85–117, 2015.

[3] T. Plotz and Y. Guan, "Deep Learning for Human Activity Recognition in Mobile Computing," *Computer*, vol. 51, no. 5, pp. 50–59, 2018.

[4] H. F. Nweke, Y. W. Teh, M. A. Al-garadi, and U. R. Alo, "Deep learning algorithms for human activity recognition using mobile and wearable sensor networks: State of the art and research challenges," *Expert Systems with Applications*, vol. 105, pp. 233–261, 9 2018.

[5] J. Wang, Y. Chen, S. Hao, X. Peng, and L. Hu, "Deep learning for sensor-based activity recognition: A Survey," *Pattern Recognition Letters*, vol. 119, pp. 3–11, 2 2018.

[6] G. E. Hinton and R. R. Salakhutdinov, "Reducing the Dimensionality of Data with Neural Networks," *Science (New York, N.Y.)*, vol. 313, no. July, pp. 504–507, 2006.

[7] O. Yildirim, R. S. Tan, and U. R. Acharya, "An efficient compression of ECG signals using deep convolutional autoencoders," *Cognitive Systems Research*, vol. 52, pp. 198–211, 2018.

[8] L. D. Medus, T. Iakymchuk, J. V. Frances-Villora, M. Bataller-Mompean, and A. Rosado-Munoz, "A Novel Systolic Parallel Hardware Architecture for the FPGA Acceleration of Feedforward Neural Networks," *IEEE Access*, vol. 7, pp. 76 084–76 103, 2019.

[9] M. G. Coutinho, M. F. Torquato, and M. A. Fernandes, "Deep neural network hardware implementation based on stacked sparse autoencoder," *IEEE Access*, vol. 7, pp. 40 674–40 694, 2019.

[10] J. Maria, J. Amaro, G. Falcao, and L. A. Alexandre, "Stacked Autoencoders Using Low-Power Accelerated Architectures for Object Recognition in Autonomous Systems," *Neural Processing Letters*, vol. 43, no. 05, pp. 445–458, 2016.

[11] S. Rifai, P. Vincent, X. Muller, X. Glorot, and Y. Bengio, "Contractive auto-encoders: explicit invariance during feature extraction," in *Proceedings of The 28th International Conference on Machine Learning (ICML-11)*, no. 1, 2011, pp. 833–840.

[12] A. Suzuki, T. Morie, and H. Tamukoh, "FPGA implementation of autoencoders having shared synapse architecture," in *PLoS One*, vol. 13, no. 03, 2018, pp. 1–22.

[13] J. Jiang, R. Hu, D. Wang, J. Xu, and Y. Dou, "Performance of the fixed-point autoencoder," *Tehnicki vjesnik - Technical Gazette*, vol. 23, no. 02, pp. 77–82, 2016.

[14] Y. LeCun, C. Cortes, and C. J. Burges, "MNIST handwritten digit database," *ATT Labs*, vol. 2, 2010. [Online]. Available: http://yann.lecun.com/exdb/mnist/

Appendix 2

II

M. Kerner, K. Tammemae, J. Raik, and T. Hollstein, “Novel Architectures for Contractive Autoencoders with Embedded Learning,” in *2020 17th Biennial Baltic Electronics Conference (BEC)*, vol. 2020-October, pp. 1–6, IEEE Computer Society, 10 2020

Novel Architectures for Contractive Autoencoders with Embedded Learning

Madis Kerner*, Kalle Tammemäe*, Jaan Raik*, Thomas Hollstein*†
*Tallinn University of Technology, Tallinn, Estonia
†Frankfurt University of Applied Sciences, Frankfurt, Germany
email: madis.kerner@taltech.ee, kalle.tammemae@taltech.ee, jaan.raik@taltech.ee, hollstein@fb2.fra-uas.de

***Abstract*—A Contractive Autoencoder (CAE) is an unsupervised Artificial Neural Network (ANN) with a regularization term controlling the internal representations. During its operation, the autoencoder extracts dominant features of the input, which can be either used for communication bandwidth reduction or as an input to another neural network. While hardware implementations for the forward pass of various ANNs can be found in literature, this paper proposes three novel architectures for Field Programmable Gate Array (FPGA) based implementations of CAE, for the first time with embedded learning: (i) topography-affine inter-layer communication via crossbar; (ii) efficient carousel-type communication scheme; (iii) optimized carousel-type architecture. All architectures use node-level parallel calculation scheme and have been implemented on a Xilinx Zynq 7020 System On Chip (SoC).**

I. Introduction

Successful applications of Deep Learning (DL) algorithms include image recognition, natural language translation, human activity recognition, and anomaly detection [1], [2], [3]. Different implementation will and have moved the boundaries of contemporary life in various ways.

The appealing benefit of DL is its ability to extract the essential features of the input signal automatically; these algorithms do not rely on domain expert knowledge and manual pre-processing of the input. Automatic feature extraction increases the modeling capabilities of DL [4].

Before the deployment, DL algorithms need training. This procedure adjusts the internal weights and biases of the network to ensure the desired behavior. However, training a DL network requires substantial computational power, and therefore, current implementations on resource-constrained systems do not include it. Instead, the model is trained offline, or the task is offloaded to the cloud [5].

This work focuses on unsupervised DL algorithms and implementations on resource-constrained systems; unsupervised behavior enables autonomous operation. One class of this kind of methods are Autoencoders (AEs), which reproduce the input signal to its output, while internal weight values are updated to minimize the difference. The middle layer(s) of an AE contain compressed features [6], which can be used for different purposes, like data-compression [7] or as input to DL network to follow.

978-1-7281-9444-8/20/$31.00 ©2020 IEEE

While the state-of-the-art addresses FPGA based ANN accelerators ([8], [9]), CAE with embedded learning is missing from the literature. [10] provides an FPGA based implementation of the sparse stacked autoencoder. However, the work does not implement hardware-based training of the network, only forward pass.

[11] provides a framework for the forward pass calculations for various architectures, with no training included. However, several applications may require embedded training.

[12] uses high-level synthesis to train stacked autoencoders. While autoencoders are all inherently related, the solution lacks the contraction term and training speed.

The main contribution of this work is to provide *the first full hardware-based implementation of the CAE [13], comprising hardware-implemented learning*. In addition, this paper follows proposals to use shared weights on the input and output layers [14] and fixed point representations for weights and biases [15].

The rest of the paper is organized as follows: Section II provides the necessary background information about CAE and the equations for the forward path and training of the network, Section III presents and analyzes the proposed architectures of the Xilinx Zynq based implementation of CAE, Section IV explains the functioning of the proposed architectures and presents timing and hardware utilization figures, and Section V provides the conclusions.

II. Contractive Autoencoder

An AE is a type of ANN that tries to reproduce the input signal to its output while reducing the data dimensionality [6]. Fig. 1 presents the architecture of an AE network, consisting of encoding and decoding parts.

The internal representation, however, does not necessarily converge to a useful generalization of the input unless some quality measure, known as regularization, is explicitly set. One type of an AE which does this is the Contractive Autoencoder (CAE) [13]; regularization is to ensure that small changes in the input yield to the same internal representation, effectively driving the middle layer towards more general features.

A. Forward Pass

This section presents the equations for CAE forward pass calculations, where n is the number of inputs and outputs, and m is the count of middle layer nodes.

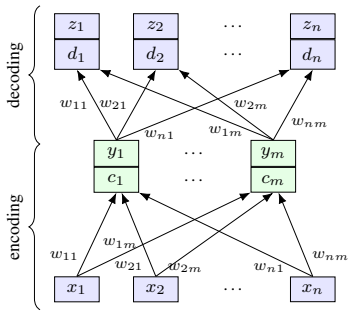


Figure 1: Architecture of the AE. Middle layer Y is the compressed representation of input X, and Z is the reconstruction of X. The rest of the figures and tables use the same coloring scheme: blue color denotes the external nodes, while green color identifies the middle layer.

The forward pass starts with the calculation of hidden representation Y corresponding to the input X (Eq. (1)). The activation function $g(x)$ used in the current proposal is the Rectified Linear Unit (ReLU), first proposed in [16].

$$y_j = g \quad \sum_{i=1}^{n} w_{ij} x_i + b_j^{(c)} \Bigg) \tag{1}$$

Next, using Eq. (2), the internal representation Y is used to calculate Z, the output.

$$z_i = g \left(\sum_{j=1}^{m} w_{ij} y_j + b_i^{(d)} \right) \tag{2}$$

Eqs. (1) and (2) complete the forward pass calculation. However, this paper addresses the training of the CAE as well.

B. Loss Function

Training of a ANN is about minimizing the loss function, which describes the difference between the actual and desired output. We have selected computationally efficient Mean Squared Error (MSE) for that (Eq. (3)).

$$L(X, Z) = \frac{1}{n} \sum_{i=1}^{n} (z_i - x_i)^2 \tag{3}$$

In the case of CAE, the loss function includes a regularization term to force the internal representation to be less sensitive to the input, yielding to more robust features. Mathematically this term translates to the Frobenius norm of the Jacobian matrix (Eq. (4)).

$$\|J_f(x)\|_F^2 = \sum_{i=1}^{n} \sum_{j=1}^{m} \left(\frac{\partial y_j}{\partial x_i} \right)^2 \tag{4}$$

The total loss of the CAE is the sum of Eqs. (3) and (4): Eq. (5), where $\theta = W, B^{(c)}, B^{(d)}$ is the collection of all the parameters, weights and biases, present in the network and λ is the coefficient to limit the amount of the contraction term in the total loss.

$$\mathcal{J}_{CAE}(\theta) = L(x, g(f(x))) + \lambda \|J_f(x)\|_F^2 \tag{5}$$

C. Gradient Descent

Training of the CAE is about minimizing the loss function, ideally to zero. Gradient descent is the standard algorithm for finding such adjustments.

To simplify the notations to follow, we first mark the squared error term in Eq. (3) as $l_i = (z_i - x_i)^2$. After this substitution, we can calculate the derivative of the MSE loss w.r.t. every weight value w_{uv} using the chain rule: Eq. (6).

$$\frac{\partial L}{\partial w_{uv}} = \frac{1}{n} \sum_{i=1}^{n} \left(\frac{\partial l_i}{\partial z_i} \frac{\partial z_i}{\partial d_i} \frac{\partial d_i}{\partial w_{uv}} \right) \tag{6}$$

The terms $\partial l_i / \partial z_i$ and $\partial z_i / \partial d_i$ in the Eq. (6) are computationally light, while the calculation of $\partial d_i / \partial w_{uv}$ follows the chain rule, again. It is important to note the impact of sharing the weights in input and output layers: if $i = u$ in Eq. (6) then $d_u = w_{uv} * y_v$,where $y_v = f(w_{uv})$ and the derivative w.r.t w_{uv} has to follow the product rule. Equation (7) presents the derivative of the decoding value.

$$\frac{\partial d_i}{\partial w_{uv}} = \begin{cases} \frac{\partial d_i}{\partial y_v} \frac{\partial y_v}{\partial c_v} \frac{\partial c_v}{\partial w_{uv}}, & \text{if } i^{(d)} \neq u \\ y_v + w_{uv} \frac{\partial y_v}{\partial c_v} \frac{\partial c_v}{\partial w_{uv}}, & \text{if } i^{(d)} = u \end{cases} \tag{7}$$

Contraction term adds additional member to the final weight update, Eq. (8) presents the formula for calculating the derivative of the contraction term w.r.t. w_{uv}

$$\frac{\partial r_{ij}}{\partial w_{ij}} = 2 w_{ij} \left(\frac{\partial y_j}{\partial c_j} \right)^2 \tag{8}$$

D. Weight and bias update values

The negative value of the gradient specifies the direction of change for a parameter to minimize the loss function (Eq. (5)).

Eqs. (9) and (10) presents the update calculation for the weight and bias values, where α amd β stand for the learning rate.

$$w_{ij} = w_{ij} - \alpha \left(\frac{\partial L}{\partial w_{ij}} + \lambda \frac{\partial r_{ij}}{\partial w_{ij}} \right) \tag{9}$$

$$b_i = b_i - \beta \frac{\partial L}{\partial b_i} \tag{10}$$

III. Proposed Architectures

CAE forward pass, and backpropagation calculations involve many loops: all the nodes in a layer need to multiply the previous layer inputs by the corresponding weights and accumulate the results. Unrolling the loops provides excellent possibilities for hardware accelerators.

All three following architecture proposals make use of the digital signal processing (DSP) slices for Multiply-Accumulate (MAC) operations, and therefore, the maximum number of possible parallel calculations is equal to the amount of available DSP based Processing Elements (PEs). I.e., keeping all

the DSP slices busy at all times gives the best performing accelerator. However, to maximize the use of DSP slices, the hardware utilization per PE has to be small to accommodate the design in the target hardware. Also, the communication channel should use the minimum amount of hardware while providing a reasonable bandwidth to feed the PEs with data.

The following proposals use the node-level parallelism: all the network nodes in CAE make use of one PE. The differences involve communication channel selection and node pruning. All the weight values and intermediate calculations are stored in distributed block RAMs associated with PEs.

The target hardware for this research is Xilinx Zynq-7020 SoC, it incorporates dual-core ARM Cortex-A9 plus Programmable Logic (PL). The PL section contains 85K logic cells, 53200 LUTs, 106400 flip-flops, 140 36Kbit block RAMs, and 220 DSP slices. The ARM cores are responsible for configuring the PL based CAE network and feeding it with the input data.

A. Timing Estimations

This section presents the timing estimations for CAE using the node-level parallelism.

During the forward pass, it takes the number of equal to the previous layer size MAC operations for a node to complete its output. Adding the bias and applying the computationally inexpensive ReLU activation add another two cycles per layer (Eqs. (1) and (2)). Eq. (11) presents the generalized formula for calculating the required cycles to complete the forward pass, where l stands for the number of layers and n_i is the i-th layer input size.

$$C_{fwd} = \sum_{i=1}^{l}(n_i + 2) \tag{11}$$

In total, it takes $n + m + 4$ cycles to complete the forward pass for a network presented in Fig. 1.

Analysis of the Eq. (9) gives the required cycle count to update a single weight value: $7n + 10$. Therefore, the total cycle count to update all the weights equals to $m(7n + 10)$, where m is the number of weights assigned to a node.

To accelerate the backpropagation, we have determined the common parts present in calculations to maximize the data reuse: Eqs. (12) and (13).

$$K_i = \frac{1}{n}\frac{\partial l_i}{\partial z_i}\frac{\partial z_i}{\partial d_i} \tag{12}$$

$$S_j = \frac{\partial y_j}{\partial c_j}\sum_{i=1}^{n} K_i w_{ij} \tag{13}$$

Eq. (14) presents the weight update formula after extracting and substituting the common parts.

$$w_{ij} = w_{ij} - \alpha(x_i S_j + K_i y_j) - \alpha\lambda\frac{\partial r_{ij}}{\partial w_{ij}} \tag{14}$$

After the substitutions, the total cycle count to update a single weight value is reduced to 11 cycles. Calculation of S_j and K_i values takes $n+1$ and 5 cycles, respectively. Eq. (15) defines the cycle count required to update all the weights in the network presented in Fig. 1 in the case of node-level parallelism.

$$C_{bp} = 11m + n + 6 \tag{15}$$

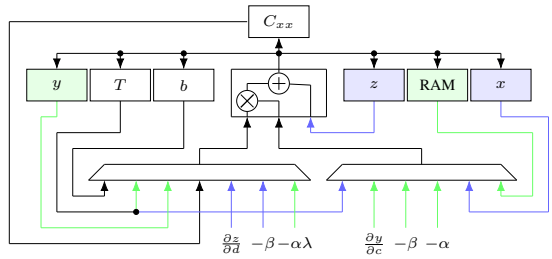


Figure 2: Data-path of the middle and IO layer PE. Blue color designates the items only present in the external layer, while the green color marks middle layer only elements. Items with no background color are present in every PE

B. Architecture 1: Baseline (BL)

The BL architecture follows the logical structure of the CAE (Fig. 1); every node is a separate PE, and the ordering of calculations follows the layered structure. Forward pass calculations start by propagating the input values to the middle layer nodes where MAC operations take place (Eq. (1)), followed by similar operations in the output layer (Eq. (2)). Training of the network follows similar layer-to-layer flow but in the opposite direction, from the output to the input.

This scheme means that while the internal and external nodes are well separated and more straightforward controlling Finite State Machines (FSMs) can be used, only one layer is executing at a time; the resources associated with the other layers are staying idle.

Figure 2 presents the data-path of the PEs, and the coloring scheme follows the rules presented in Fig. 1.

The central part of the PE is the DSP slice, which carries out the calculations. The output of the DSP can be stored to one of the registers, to the block RAM holding the weight values w_{ij}, or transmitted to another node via the cross-bar connection point C_{xx}, and the inputs use multiplexers to connect to the data sources. DSP can execute a set of pre-defined instructions; the instruction selection and switching of the input multiplexers are under control of the FSM. This setup is sufficient for the forward pass and backpropagation calculations.

Network nodes in CAE need to communicate with each other as the output calculation and updating the weight values are performed in collaboration. BL architecture uses the cross-bar switch (Fig. 3) as the communication channel to achieve that. Both sides of the cross-bar have controlling ports *CTRL* for reading and writing the input and output data and for network configuration purposes, and dummy loads *DLd* to

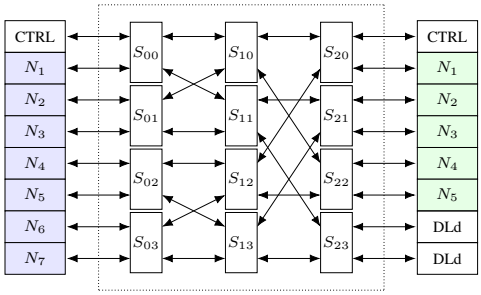


Figure 3: Layout of the butterfly cross-bar switch. Layers of the CAE connect to the different sides. *CTRL* ports are used by the ARM processing unit for flow control and data transfer.

balance the cross-bar and assist synthesizer in pruning the redundant hardware.

C. Architecture 2: Efficient Communication (CCom)

The second proposal combines the network layers to overcome the phenomenon of the idling layers present in BL architecture. Further, it replaces the crossbar switch by the simple carousel-like communication (CCom) channel (Fig. 4).

This kind of architecture reaches the maximum performance while executing the layers with the size equal to the count of available PEs, or if the network layer node count is the multiple of PE count: it takes the number of equal to layers input size cycles for a layer to complete the forward pass. Node N_n in Fig. 4 can proceed with the next layer after receiving the last input value without waiting for the node N_1 to complete.

The same does not hold for smaller layers: $n - m$ PEs are not needed and stay idle. While the CCom architecture can not avoid that problem entirely, it tries to mitigate the consequences by calculating multiple sets of outputs for a smaller layer. In the case of CAE, an output layer node has to receive all the middle layer outputs to complete its forward pass calculation, for example. It takes m cycles if multiple sets of interleaved middle layer outputs are available, compared to n cycles required otherwise. I.e., performing redundant calculations speeds up the start of the next layer calculations. The same holds for the S (Eq. (13)) calculations in case of backpropagation.

Table I presents an example of the described scheme for the middle layer representation in the network with $n = 7$ external layer and $m = 5$ internal layer nodes. Every node adds $w_{ij}x_i$ to the data present in the carousel and forwards it, completing the Eq. (1).

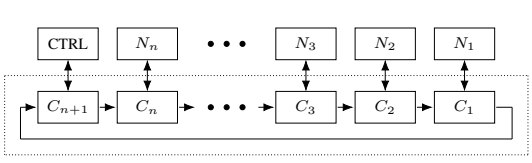


Figure 4: Carousel like communication channel. Data advances in every clock cycle. The node *CTRL* is connected to the controlling ARM processing unit.

However, this scheme requires the total amount of nodes to be equal to multiple of the internal layer units m; the following calculations require the set of internal layer values to arrive in the correct sequence. Therefore, the network must include dummy nodes $D_1 \dots D_3$. These dummy nodes act as a network node with all weight values set to zero and forward the data.

Table I: Performance biased calculation scheme of CCom architecture. Multiple sets of values are calculated to speed up the execution steps to follow.

D_3	D_2	D_1	N_7	N_6	N_5	N_4	N_3	N_2	N_1
$Y_1^{(1)}$	$Y_5^{(2)}$	$Y_4^{(2)}$	$Y_3^{(2)}$	$Y_2^{(2)}$	$Y_1^{(2)}$	$Y_5^{(1)}$	$Y_4^{(1)}$	$Y_3^{(1)}$	$Y_2^{(1)}$
$Y_2^{(1)}$	$Y_1^{(1)}$	$Y_5^{(2)}$	$Y_4^{(2)}$	$Y_3^{(2)}$	$Y_2^{(2)}$	$Y_1^{(2)}$	$Y_5^{(1)}$	$Y_4^{(1)}$	$Y_3^{(1)}$
$Y_3^{(1)}$	$Y_2^{(1)}$	$Y_1^{(1)}$	$Y_5^{(2)}$	$Y_4^{(2)}$	$Y_3^{(2)}$	$Y_2^{(2)}$	$Y_1^{(2)}$	$Y_5^{(1)}$	$Y_4^{(1)}$
$Y_4^{(1)}$	$Y_3^{(1)}$	$Y_2^{(1)}$	$Y_1^{(1)}$	$Y_5^{(2)}$	$Y_4^{(2)}$	$Y_3^{(2)}$	$Y_2^{(2)}$	$Y_1^{(2)}$	$Y_5^{(1)}$
$Y_5^{(1)}$	$Y_4^{(1)}$	$Y_3^{(1)}$	$Y_2^{(1)}$	$Y_1^{(1)}$	$Y_5^{(2)}$	$Y_4^{(2)}$	$Y_3^{(2)}$	$Y_2^{(2)}$	$Y_1^{(2)}$
$Y_1^{(2)}$	$Y_5^{(1)}$	$Y_4^{(1)}$	$Y_3^{(1)}$	$Y_2^{(1)}$	$Y_1^{(1)}$	$Y_5^{(2)}$	$Y_4^{(2)}$	$Y_3^{(2)}$	$Y_2^{(2)}$
$Y_2^{(2)}$	$Y_1^{(2)}$	$Y_5^{(1)}$	$Y_4^{(1)}$	$Y_3^{(1)}$	$Y_2^{(1)}$	$Y_1^{(1)}$	$Y_5^{(2)}$	$Y_4^{(2)}$	$Y_3^{(2)}$
$Y_3^{(2)}$	$Y_2^{(2)}$	$Y_1^{(2)}$	$Y_5^{(1)}$	$Y_4^{(1)}$	$Y_3^{(1)}$	$Y_2^{(1)}$	$Y_1^{(1)}$	$Y_5^{(2)}$	$Y_4^{(2)}$
$Y_4^{(2)}$	$Y_3^{(2)}$	$Y_2^{(2)}$	$Y_1^{(2)}$	$Y_5^{(1)}$	$Y_4^{(1)}$	$Y_3^{(1)}$	$Y_2^{(1)}$	$Y_1^{(1)}$	$Y_5^{(2)}$
$Y_5^{(2)}$	$Y_4^{(2)}$	$Y_3^{(2)}$	$Y_2^{(2)}$	$Y_1^{(2)}$	$Y_5^{(1)}$	$Y_4^{(1)}$	$Y_3^{(1)}$	$Y_2^{(1)}$	$Y_1^{(1)}$

However, this scheme requires the total amount of nodes to be equal to multiple of the internal layer units m; the following calculations require the set of internal layer values to arrive in the correct sequence. Therefore, the network must include dummy nodes $D_1 \dots D_3$. These dummy nodes act as a network node with all weight values set to zero and forward the data.

Figure 5 presents the data-path of the CCom PE, where all the PEs implement all the features. This design choice increases the throughput of the network at the cost of the hardware resources.

D. Architecture 3: Resource-Optimized (CCom-RO)

Architecture CCom-RO shares the carousel-like communication channel (Fig. 4) design with CCom but is resource optimized version of it.

While the PEs in CCom-RO are still combined, only m nodes have the full functionality. The rest $n-m$ nodes include the necessary hardware to support the middle layer related calculations only. Naturally, this design choice reduces the throughput: the network calculates only one set of middle layer features, causing maximum of $n - m$ clock cycles delay for a PE to receive data for further operations.

Table II presents an example of CCom-RO network with $n = 7$ external layer- and $m = 5$ internal layer nodes. Only the first m nodes, $N_1 \dots N_5$, hold the y_j values upon completion of the middle-layer calculation, the remaining $n - m$ nodes have simplified structure.

Figure 5 presents the data-path of the CCom-RO architecture PE. As stated, this architecture has a different design for the PEs: green color indicates the additional paths and registers included in the first m PEs, while the reduced PEs comprise the uncolored part only.

IV. Results and Analysis

First of all, an FPGA based architecture needs to be synthesizable and provide a feasible amount of functionality. As

Table II: Resource optimised calculation scheme for CCom-RO architecture. Only one set of internal layer values are calculated.

N_7	N_6	N_5	N_4	N_3	N_2	N_1
Y_1	-	-	Y_5	Y_4	Y_3	Y_2
Y_2	Y_1	-	-	Y_5	Y_4	Y_3
Y_3	Y_2	Y_1	-	-	Y_5	Y_4
Y_4	Y_3	Y_2	Y_1	-	-	Y_5
Y_5	Y_4	Y_3	Y_2	Y_1	-	-
-	Y_5	Y_4	Y_3	Y_2	Y_1	-
-	-	Y_5	Y_4	Y_3	Y_2	Y_1

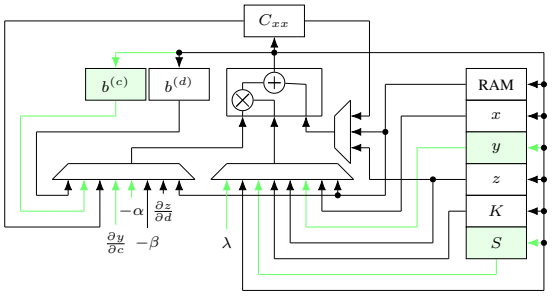


Figure 5: Data-path of the CCom and CCom-RO PEs. In case of CCom, every PE implements all the elements. For CCom-RO, reduced PEs do not incoroprate the green colored features.

all the proposed architectures use one DSP slice per PE, the maximum number of parallel PEs equals to the number of available slices. Selected target platform Xilinx Zynq 7020 has 220 DSP slices.

Table III provides the synthesis results. All the architectures were synthesized using 30 internal layer nodes, 100MHz clock speed, and the external layer size was increased to fill the target hardware. The unit for the size value is hardware slices. The column *Chnl Size* provides the resource usage for communication channel: cross-bar for the BL and carousel-like channel for the CCom and CCom-RO architectures, and the columns *DSP* and *bRAM* hold the total amount of DSP slices and block RAMs used, respectively.

The BL allows the maximum network with fewest nodes to be synthesized to the target hardware. Although the PEs use fewer resources than in the case of the other two architectures, the size of the cross-bar switch is the bottleneck; the carousel-like chain is more straightforward and not as resource hungry.

CCom results in a higher node count compared to the BL. While combining the layers functionality results in a higher amount of resources required by a single PE, the communication channel is lighter, boosting the maximum network size by 50 additional nodes. It has to be noted that CCom synthesis has 150 external- and 30 middle layer nodes, merged into the combined count.

The largest CAE can be synthesized using the CCom-RO architecture; again, it has to be noted that the total size for the external layer is $170 + 30 = 200$, the sum of full and

Table III: Maximum network sizes and hardware usage for CAE synthesis targeting Zynq7020 SoC; the size is expressed in FPGA slices.

	ExtNode		MidNode		Chnl	DSP	bRAM
Arch	Count	Size	Count	Size	Size	Count	Count
BL	100	65	30	90	7500	130	15
	ExtNode		Mid+ExtNode		Chnl	DSP	bRAM
CCom	N/A	N/A	150	105	4377	150	75
CCom-RO	170	75	30	105	1678	200	100

reduced PEs. The largest possible network was expectable as the resources allocated for the performance boost included in the CCom were skipped. The decrease in the communication channel size is expected as well; only the nodes with full middle layer functionality use an additional pipeline to transmit more data at once.

Table IV provides the forward pass and training execution times for the described architectures and the theoretical maximum (Eqs. (11) and (15)). The timing results were acquired using the HDL simulator, and the size of the network was the maximum achieved during the synthesis: 200 external- and 30 middle layer nodes.

The fastest architecture is CCom, and it is about four times faster compared to the BL. The usage of the cross-bar switch can explain the reduced execution time of the BL: a node has to wait until the channel becomes available to be able to transmit the data. Similarly, if data from another node is required for calculations, the node has to wait for it. Further, many calculation results have to be broadcasted to every PE in the following layer; only one broadcast can complete in every clock cycle, the rest of the PEs have to wait for the communication resources become available. In conclusion, synchronization by the communication channel yields to more straightforward design and lower usage of hardware per PE but slows down the execution.

CCom and CCom-RO use synchronous PEs; the state of other PEs is known in every time step, and there is no competition for the communication channel. The availability of resources is guaranteed by design, ensuring the optimal execution flow and faster execution times. Also, the communication channel does not have to implement any handshake signals.

The fact that CCom-RO is slower compared to the CCom is expected. However, CCom-RO is still about 1.6 times faster compared to BL.

Another important observation is that only the CCom architectures almost reached the theoretical maximum performance. CCom-RO does not optimize the calculations for the smaller middle layer, and therefore, the execution speed suffers.

Further, data from the MNIST database of handwritten digits [17] was used to prove the functionality of the design. However, this paper does not aim to outperform state-of-the-art classifiers but uses the MNIST database to prove the operation of the HW based implementation.

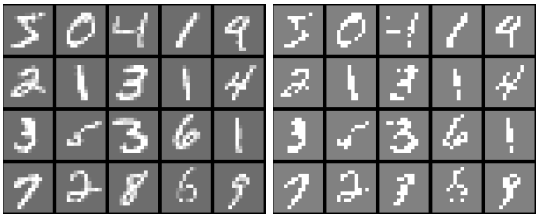
While the database contains 28x28 pixel images, the format was reduced to 14x14 pixels, resulting in 196 node input

Table IV: Execution time of the CAE with 200 external- and 30 middle layer nodes. The clock speed of the designs was set to 100MHz.

Arch	Forward Pass (μs)	Training (μs)
Theoretical	2.3	5.4
BL	13.4	25.7
CCom	2.8	5.9
CCom-RO	6.1	9.5

layer, which was possible to synthesize using CCom-RO architecture. During the experiment, the CAE was trained using the first 20 images from the MNIST database, and every digit was input to the network 200 times. The network converged using 16 bit long fixed-point implementation with 12 fractional bits. In parallel to the hardware execution, Matlab functional model was used to ensure proper execution.

Figure 6 shows the results of the conducted test: the output of the 3-layer 196-10-196 nodes CAE (Figure 6b) correlate to the down-scaled 14x14 MNIST database images (Figure 6a).



(a) Input to the CAE, down-scaled 14x14 MNIST images.

(b) Output of the CAE, using 16.12 fixed-point representation and 10 internal layer nodes.

Figure 6: Operation example of the trained 3-layer 196-10-196 nodes CAE using 16.12 fixed-point representations.

V. Conclusions

The novel contribution of this work is to provide entirely FPGA-based implementations of a CAE, including embedded learning. The three presented approaches follow proposals to use shared weights in the input and output layers [14] and fixed-point representations for weights and biases [15].

The synthesize results and execution speed propose that mimicking the theoretical architecture of the network (BL) in hardware is not feasible; resources used by the cross-bar switch are high compared to the network itself. Also, using the same set of PEs for all layers (CCom and CCom-RO) prune the idling nodes and improve the computation efficiency.

A carousel-like communication scheme resulted in faster execution speed and lower hardware resources used. Whereas the choice between the architectures CCom and CCom-RO is the matter of resources to execution speed tradeoff, while CCom proves that simple communication channel is sufficient to feed the PEs with data in case of node-level parallelism.

Also, Section III-A proves that extracting the common terms from backpropagation algorithms can further improve the node-level parallelism. Rearranging the computational loops resulted in $m(7n+10)/(11m+n+6) = 42300/536 \approx 79$ times fewer cycles to update all the weights in the test network.

The functionality of the network was proved using 20 downscaled images from the MNIST database. Every digit was applied to the network 200 times, and the network converged using 16 bit fixed-point implementation with 12 fractional bits.

CCom-RO architecture resulted in the network with the most nodes in the target hardware platform: Xilinx Zynq 7020 can accommodate CAE with 200 external- and 30 middle layer nodes. While CCom architecture provides the fastest execution time, it takes $2.8 + 5.9 = 8.7(\mu s)$ to execute forward pass and backpropagation in 200-30-200 node CAE, running at 100MHz clock speed.

References

[1] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *Nature*, vol. 521, pp. 436–444, 5 2015.

[2] J. Schmidhuber, "Deep Learning in neural networks: An overview," *Neural Networks*, vol. 61, pp. 85–117, 2015.

[3] T. Plotz and Y. Guan, "Deep Learning for Human Activity Recognition in Mobile Computing," *Computer*, vol. 51, no. 5, pp. 50–59, 2018.

[4] H. F. Nweke, Y. W. Teh, M. A. Al-garadi, and U. R. Alo, "Deep learning algorithms for human activity recognition using mobile and wearable sensor networks: State of the art and research challenges," *Expert Systems with Applications*, vol. 105, pp. 233–261, 9 2018.

[5] J. Wang, Y. Chen, S. Hao, X. Peng, and L. Hu, "Deep learning for sensor-based activity recognition: A Survey," *Pattern Recognition Letters*, vol. 119, pp. 3–11, 2 2018.

[6] G. E. Hinton and R. R. Salakhutdinov, "Reducing the Dimensionality of Data with Neural Networks," *Science (New York, N.Y.)*, vol. 313, no. July, pp. 504–507, 2006.

[7] O. Yildirim, R. S. Tan, and U. R. Acharya, "An efficient compression of ECG signals using deep convolutional autoencoders," *Cognitive Systems Research*, vol. 52, pp. 198–211, 2018.

[8] E. Wang, J. J. Davis, R. Zhao, H. C. Ng, X. Niu, W. Luk, P. Y. Cheung, and G. A. Constantinides, "Deep neural network approximation for custom hardware: Where We've Been, Where We're going," *ACM Computing Surveys*, vol. 52, no. May, pp. 1–39, 2019.

[9] S. Mittal, "A survey of FPGA-based accelerators for convolutional neural networks," pp. 1109–1139, 2020.

[10] M. G. Coutinho, M. F. Torquato, and M. A. Fernandes, "Deep neural network hardware implementation based on stacked sparse autoencoder," *IEEE Access*, vol. 7, pp. 40 674–40 694, 2019.

[11] L. D. Medus, T. Iakymchuk, J. V. Frances-Villora, M. Bataller-Mompean, and A. Rosado-Munoz, "A Novel Systolic Parallel Hardware Architecture for the FPGA Acceleration of Feedforward Neural Networks," *IEEE Access*, vol. 7, pp. 76 084–76 103, 2019.

[12] J. Maria, J. Amaro, G. Falcao, and L. A. Alexandre, "Stacked Autoencoders Using Low-Power Accelerated Architectures for Object Recognition in Autonomous Systems," *Neural Processing Letters*, vol. 43, no. 05, pp. 445–458, 2016.

[13] S. Rifai, P. Vincent, X. Muller, X. Glorot, and Y. Bengio, "Contractive auto-encoders: explicit invariance during feature extraction," in *Proceedings of The 28th International Conference on Machine Learning (ICML-11)*, no. 1, 2011, pp. 833–840.

[14] A. Suzuki, T. Morie, and H. Tamukoh, "FPGA implementation of autoencoders having shared synapse architecture," in *PLoS One*, vol. 13, no. 03, 2018, pp. 1–22.

[15] J. Jiang, R. Hu, D. Wang, J. Xu, and Y. Dou, "Performance of the fixed-point autoencoder," *Tehnicki vjesnik - Technical Gazette*, vol. 23, no. 02, pp. 77–82, 2016.

[16] V. Nair and G. Hinton, "Rectified Linear Units Improve Restricted Boltzmann Machines," in *Proceedings of the 27th International Conference on Machine Learning*, 2010, pp. 807–814.

[17] Y. LeCun, C. Cortes, and C. J. Burges, "MNIST handwritten digit database," *ATT Labs*, vol. 2, 2010. [Online]. Available: http://yann.lecun.com/exdb/mnist/

Appendix 3

III

M. Kerner, K. Tammemae, J. Raik, and T. Hollstein, “Triple Fixed-Point MAC Unit for Deep Learning,” in *2021 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, vol. 2021-February, pp. 1404–1407, Institute of Electrical and Electronics Engineers Inc., 2 2021

Triple Fixed-Point MAC Unit for Deep Learning

Madis Kerner*, Kalle Tammemäe*, Jaan Raik*, Thomas Hollstein*†
*Tallinn University of Technology, Tallinn, Estonia
†Frankfurt University of Applied Sciences, Frankfurt, Germany
email: madis.kerner@taltech.ee, kalle.tammemae@taltech.ee, jaan.raik@taltech.ee, hollstein@fb2.fra-uas.de

***Abstract*—Deep Learning (DL) algorithms have proved to be successful in various domains. Typically, the models use Floating Point (FP) numeric formats and are executed on Graphical Processing Units (GPUs). However, Field Programmable Gate Arrays (FPGAs) are more energy-efficient and, therefore, a better platform for resource-constrained devices. As the FP design infers many FPGA resources, it is replaced with quantized fixed-point implementations in state-of-the-art. The loss of precision is mitigated by dynamically adjusting the radix point on network layers, reconfiguration, and re-training. In this paper, we present the first Triple Fixed-Point (TFxP) architecture, which provides the computational precision of FP while using significantly fewer hardware resources and does not need network re-training. Based on a comparison of FP and existing Fixed-Point (FxP) implementations in combination with a detailed precision analysis of YOLOv2 weights and activation values, the novel TFxP format is introduced.**

I. Introduction

Deep Learning (DL) algorithms have been successfully deployed in various domains, including image recognition, natural language detection, among others [1], [2]. These algorithms automatically extract the input signal's essential features and do not rely on domain expert knowledge and manual pre-processing.

A DL algorithm has a layered structure and comprises various types of layers. Each layer includes neurons that receive and process data from the previous layer and feeds the next layer. This kind of build-up provides excellent possibilities for acceleration: all the neurons can execute in parallel. Therefore, typical platform for running DL is PC based, using Graphical Processing Unit (GPU).

Due to the success of DL, contemporary research explores the possibilities to execute these algorithms in resource constrained devices as well: Field Programmable Gate Arrays (FPGAs) form an excellent platform for this. While FPGAs are power efficient compared to GPUs, the DL algorithms rely on Floating Point (FP) representations of parameters, which infer a lot of Hardware (HW) resources. Despite the search for efficient FP support in FPGAs [3], the available Digital Signal Processing (DSP) slices are still more suitable for fixed-point operations.

Although it is undoubtedly possible to perform FP calculations on FPGAs, the inferred HW resources are high: constructing a half-precision multiply-accumulator, which is a typical computational unit in Artificial Neural Networks (ANNs), requires three DSPs and several hundred LUTs and registers [4].

Contemporary research tries to overcome this obstacle and explores different approximation techniques to get rid of FP representations. Typically, it means quantizing the network parameters to fixed-point numbers of some sort, or binary values in extreme cases [5]. Many works have achieved reasonable inference accuracy using quantized networks, suggesting that the precision of FP is not required. However, there are proposals that better precision than the deep-quantization is necessary [6].

In this paper, we propose a novel Triple Fixed-Point (TFxP) based Multiply-Accumulate (MAC) unit for ANNs. TFxP extends the Dual Fixed-Point (DFxP) format [7] by introducing one additional range. This extra middle range allows extending the usable dynamic range of the format, while the added HW cost is small.

We show that TFxP can be used as the drop-in replacement for FP. To justify the proposal, we first analyzed the required representation range of the YOLOv2 network [8]. This network comprises 23 convolutional layers, which all make heavy use of MAC operations. Further, as our simulations show, the YOLOv2 network achieves the same inference precision with TFxP format as with FP and does not require retraining to accomplish that.

The rest of the paper is organized as follows: Section II performs the design space exploration by analysing the weight and activation values of YOLOv2 network, Section III presents the TFxP format, Section IV analysis and compares the average precision of converted network, Section V presents the XILINX DSP48E1 based TFxP MAC unit and synthesis results, and Section VI provides the conclusions.

II. Design Space Exploration

This section provides an analysis of a Deep Neural Network (DNN) to determine the numeric type requirements. The DNN of our choice was YOLOv2; it is a well-known Convolutional Neural Network (CNN) and comprises 23 convolutional layers, among others.

Convolutional layers make heavy use of MAC operations: kernels move across the input feature map, and input values multiplied by the corresponding weight values are accumulated to form the output. Fig. 1 illustrates this operation.

In a typical CNN, 90% of the execution time goes to the convolutional layers during the inference phase [9]. I.e., the MAC unit has to be efficient; it should execute fast and not infer too much of HW to allow the maximum amount of parallelism.

2021 Design, Automation & Test in Europe Conference & Exhibition (DATE) | 978-3-9819263-5-4/21/$31.00 ©2021110.23919/DATE51398.2021.9474020

Authorized licensed use limited to: Tallinn University of Technology. Downloaded on December 07,2021 at 21:06:20 UTC from IEEE Xplore. Restrictions apply.

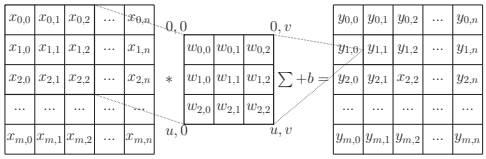


Figure 1. Convolution: m*n input features X are convolved with u*v weight matrixes to form the output Y.

To analyze the required range of the network parameters, we first performed a statical analysis and determined the minimum, maximum, and median of all the network's weight values. After this, we run the inference using a realistic photo, white image, and black image while recording all the layer outputs' extreme and median values. Table I presents the analyses results.

Table I
ANALYSIS OF THE WEIGHTS AND ACTIVATION VALUES OF YOLOV2.

	Minimum	Maximum	Median
Weights	-18.6	99.5	13.7
Photo	-113.9	106.3	0.7
All white	-57.9	31.6	1.4
All black	-23.1	28.6	1.2

As the analysis shows, most activation and weight values are low in magnitude. However, there are larger values present as well. The candidate drop-in replacement for FP should cope with the range and keep the maximum precision for median values. Additionally, the chosen data type has to use a minimum amount of bits to cope with memory bandwidth's limitations.

According to Table I, a numeric type with an 8-bit integer part can fit all the extreme values without over- or underflow. However, median values suggest that this range is not required for most of the calculations. While FP representations inherently solve this problem, fixed-point numbers have to use other means to overcome this.

A typical approach found in literature either makes use of dynamically adjusting the radix point in fixed-point numbers [10], or uses FPGA re-configuration to change the numeric format [11]. Both of these approaches have drawbacks: dynamically adjusting the radix point requires arbitrary shifters in Processing Elements (PEs), while re-configuration slows down the algorithm's execution.

This paper proposes the drop-in replacement for FP representations, which does not require re-configuration or dynamic adjustments: Triple Fixed-Point (TFxP).

III. TRIPLE FIXED-POINT

In search of a numeric format that does not sacrifice small numbers' precision to the range as much as the fixed-point representation does, the authors in [7] propose DFxP. DFxP makes use of a single exponent bit to select the radix point location. Authors in [12] use the format to replace the FP for CORDIC calculations and extend the work to use dynamic DFxP in [13].

Dynamic DFxPs undoubtedly improve the accuracy of computations as it takes a step towards FP representations. While the format is more HW friendly, the dynamic nature calls for arbitrary shifters.

Reserving one extra bit for exponent extends the DFxP to Triple Fixed-Point (TFxP). It bears the same objective as dynamic DFxP: develop the precision for a specific numeric range, but infers less FPGA resources.

Fig. 2 presents the proposed TFxP format, where a_x and b_x are the bit length of integer and fractional parts respectively. Depending on the range, (1) defines the numeric value D the representation is holding, where X is the significand, and E denotes the value of the exponent field.

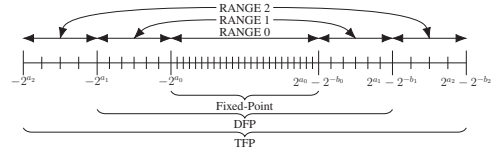


Figure 2. Triple Fixed-Point (TFxP) representation. Ranges 1 and 2 increase the range while sacrificing the precision.

$$D = \begin{cases} X \cdot 2^{-b_0} & \text{if } E = 0 \\ X \cdot 2^{-b_1} & \text{if } E = 1 \\ X \cdot 2^{-b_2} & \text{if } E = 2 \end{cases} \quad (1)$$

In order to convert FP to TFxP, a suitable target range has to be selected. Equation (2) defines the range selection: the first range capable of accomodating the value without the over- or underflow is chosen, and the exponent field E is set accordingly. The value 3 indicates over- or underflow, based on the sign bit S.

$$E = \begin{cases} 0 & \text{if } -2^{a_0} < D < 2^{a_0} - 2^{-b_0} \text{ else} \\ 1 & \text{if } -2^{a_1} < D < 2^{a_1} - 2^{-b_1} \text{ else} \\ 2 & \text{if } -2^{a_2} < D < 2^{a_2} - 2^{-b_2} \text{ else} \\ 3 & \text{overflow} \end{cases} \quad (2)$$

Table II shows the bit fields of the TFxP format 16_13_9_5, where the notion of format is $n\_b_0\_b_1\_b_2$ and n is the total number of bits the representation uses.

Table II
TRIPLE FIXED-POINT (TFXP) FORMAT 16_13_9_5.

Mode		**Signed significand**													
15	**14**	**13**	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
0	0	S	fraction												
0	1	S	integer						fraction						
1	0	S	integer									fraction			

A numeric format's critical property is the dynamic range: the ratio of the absolute values of the largest and smallest numbers the format can accommodate (3).

Authorized licensed use limited to: Tallinn University of Technology. Downloaded on December 07,2021 at 21:06:20 UTC from IEEE Xplore. Restrictions apply.

$$\text{Dynamic Range} = 20\log_{10}(2^{a_2+b_0}) \text{ (dB)} \quad (3)$$

The maximum dynamic range for the TFxP is the same as for DFxP. However, constructing such a range with DFxP loses the precision in the middle range entirely. TFxP mitigates this problem by using the middle range, and therefore, ensures a more comprehensive usable dynamic range.

IV. TFxP YOLOv2 inference precision

This section provides the results of YOLOv2 inference precision using the TFxP and DFxP and FP formats. The analysis was performed using MATLAB, while mex functions were used to add support for DFxP and TFxP.

Before the experiment, all the network's weight values were converted to the data type under test, and the AP@[.5:.95] on COCO 2014 validation dataset [14] was used for comparison. Table III presents the network accuracy for different formats, including the FP. The length of bitfields is marked using the same notation as for TFxP (Table II).

Table III
PRECISION OF THE YOLOV2 NETWORK USING FIXED-POINT (FXP), DFXP, TFXP, AND FP FORMATS.

	FxP 16_13	**DFxP 16_13_9**	**DFxP 16_13_5**	**TFxP 16_13_9_5**	**FP**
AP@[.5:.95]	0.45	0.47	0.49	0.52	0.52

The first format presented in Table III, FxP 16_13, has only three bits for the signed integer part. However, range analysis in Table I suggests that seven bits plus the sign bit are required. Therefore, FxP 16_13 can not reach the precision of FP.

The second format, DFxP 16_13_9, extends the maximum integer part to five bits while preserving the lower range accuracy. The precision increases, but as there are still overflows present, it can not reach the level of FP either.

The DFxP 16_13_5 format sets the upper range to nine bits, ensuring no overflows. The precision increases but does not reach the level of FP. Here, there is a more significant gap between upper and lower ranges than DFxP 16_13_9. TFxP addresses that issue by introducing one additional range.

As the results show, TFxP is the only format that reaches the precision of FP. Compared to DFxP 6_13_5, which can avoid overflows as well, the TFxP extends the precision of middle range values.

V. MAC unit

This section presents the TFxP MAC unit. It has been synthesized to XILINX System On Chip (SoC) device Z-7020 and makes use of HW DSP48E1 slices.

The DSP48E1 has four inputs and can natively perform various operations on them using integers, including MAC. However, using the FxP format requires additional considerations like radix point alignment.

Multiplication of two TFxP numbers produces the output O with shifted radix point, (4).

$$O = D_0 \cdot 2^{b_0} \cdot D_1 \cdot 2^{b_1} = D_0 \cdot D_1 \cdot 2^{b_0+b_1} \quad (4)$$

In the case of FxP multiplications, both operands have the same radix point location. Therefore, the result always has the same shift, and the internal accumulator can directly be used. The same does not hold for TFxP format: the radix point location of the multiplication output is the sum $b_0 + b_1$ (4). The total number of possibilities equals the *combinations with repetitions*: there are six different shifts possible.

Operands with different radix point locations cannot directly be summed; correction logic is required in the accumulator loop. This either reduces the maximum operating frequency of the MAC unit or increases the latency if a pipeline is used.

The proposed MAC unit (Fig. 3) ensures the fixed shift in multiplication output, independent of the input operands. The maximum possible shift in the multiplication output equals $2b^0$: double the shift of the lowest range. In case one of the inputs does not belong to the lowest range, an additional pre-shift has to be applied: the proposed MAC unit uses input multiplexers to achieve that. Compared to FP, the TFxP MAC does not require full-featured shifters as the total amount of possible input combinations is limited.

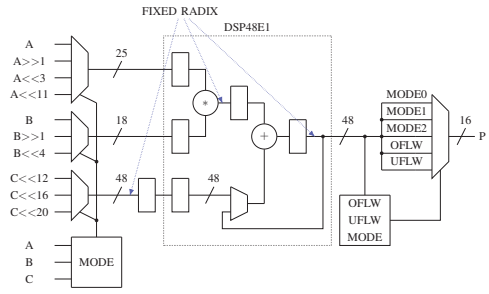


Figure 3. TFxP MAC. The design wrapes XILINX DSP48E1 HW slices. Blue arrows mark the locations where the radix point positions are matched.

The DSP48E1 slice has an additional input C, which can be added to the multiplication result instead of the internal accumulation. This input's radix point is set to the same position as the multiplication output.

The maximum pre-shift is limited to the DSP slice capabilities. Given that the width of the input A is 25 bits, the maximum possible left-shift for that input is $25 - 14 = 11$. For input B, the maximum shift is $18 - 14 = 4$, yielding $11 + 4 = 15$ bits total.

The maximum shift for the format presented in Table II is $2b^0 = 26$ bits. Similarly, the shortest fractional part in multiplication output is $2b^2 = 10$. Therefore, the total required pre-shift is $26 - 10 = 16$, one bit more than the maximum of 15. The proposed MAC unit mitigates this problem by setting the common fixed shift to 25, and performs the 1-bit right shift to one of the operands if both multiplication inputs belong to the lowest range.

Table IV presents the required pre-shifts to fix the multiplication output radix point. The least significant bit of input A selects the behavior if both of the operands are from the

Authorized licensed use limited to: Tallinn University of Technology. Downloaded on December 07,2021 at 21:06:20 UTC from IEEE Xplore. Restrictions apply.

lowest range: if A0 $= 0$, input A is right-shifted; otherwise, the input B is right-shifted, and one or zero bits of data is lost.

Table IV
RANGES OF THE TFXP MULTIPLICATION INPUTS AND REQUIRED PRE-SHIFTS FOR THE FIXED RADIX POINT LOCATION IN THE OUTPUT.

A Range	B Range	$\sum$ Shift	A Pre-Shift	B Pre-Shift
0	0	26	0/-1 [a]	0/-1 [b]
0	1	22	3	0
0	2	18	3	4
1	0	22	3	0
1	1	18	3	4
1	2	14	11	0
2	0	18	3	4
2	1	14	11	0
2	2	10	11	4

[a] -1 if A0 $= 0$, 0 otherwise.
[b] -1 if A0 $= 1$, 0 otherwise.

Figure 4 presents the DSP slice's output: signed fixed-point number with 25 fractional bits. The bits in the *OF GUARD* field have to match the sign bit; over or underflow has occurred otherwise. The range selection logic is similar to overflow check: if all the bits in field R_2 equal to the sign bit, the highest range is not needed. The same holds for the field R_1.

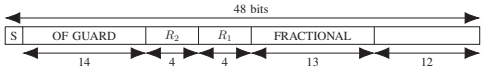


Figure 4. DSP slice output. Fields R_1 and R_2 determine the range. 14-bit *OF GUARD* is used to check the over- and underflows.

Table V presents the synthesis results of the MAC unit. Additionally, all the formats infer a single DSP slice.

The first format, 16_14_9_5, uses a maximum 14-bit fractional part: restricting the exponent field to 1 bit for the range 0 allows it. The maximum output radix point is 28 in that case, which yields to 18 bits pre-shift, 3 bits more than allowed. Additional logic to analyze and loose three least significant bits from the input if both operands belong to the lowest range infers a lot of additional HW compared to 16_13_9_5 format. Therefore, the proposed MAC unit uses the latter.

Table V
INFERRED HW OF DFXP AND TFXP FORMATS.

Format	LUTs	Regs	Slices	Power (W)	WNS (ns)	clk (MHz)
TFxP 16_14_9_5	124	23	35	0.148	0.834	393
TFxP 16_13_9_5	80	22	23	0.147	0.909	393
DFxP 16_13_5	72	18	29	0.133	0.680	393

Comparing the TFxP 16_13_9_5 and DFxP 16_13_5 formats reveals that additional middle range has mild impact on iferred HW. As an interesting observation, the synthesizer managed to combine the TFxP MAC to fewer HW slices compared to the DFxP.

Despite the positive Worst Negative Slack (WNS), the maximum operating frequency is limited by the DSP slice. All the designs can execute at 393 MHz.

VI. CONCLUSIONS

The novel contribution of this work is to provide the Triple Fixed-Point (TFxP) format for Deep Neural Networks (DNNs); it can directly replace the Floating Point (FP) format without the network re-training. The format is proposed based on analyzing the YOLOv2 network parameters and activation values during the inference phase, followed by the converted network's precision analysis. From the application point of view, direct conversion of FP to TFxP allows training the network using a Graphical Processing Unit (GPU), and deployment on Field Programmable Gate Array (FPGA), for example.

In addition to the format proposal, TFxP based Multiply-Accumulate (MAC) unit has been presented. The proposed MAC unit wraps the XILINX DSP48E1 slice to achieve the best performance and power efficiency. Compared to the FP, TFxP MAC infers much less HW resources, while the inference precision of the network is retained without the re-training.

REFERENCES

[1] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *Nature*, vol. 521, pp. 436–444, 5 2015.
[2] J. Schmidhuber, "Deep Learning in neural networks: An overview," *Neural Networks*, vol. 61, pp. 85–117, 2015.
[3] M. Langhammer and B. Pasca, "Design and Implementation of an Embedded FPGA Floating Point DSP Block," Altera, Tech. Rep., 2014. [Online]. Available: https://hal.archives-ouvertes.fr/hal-01089172
[4] Xilinx, "Performance and Resource Utilization for Floating-point." [Online]. Available: https://www.xilinx.com/support/documentation/ip_documentation/ru/floating-point.html
[5] E. Wang, J. J. Davis, R. Zhao, H. C. Ng, X. Niu, W. Luk, P. Y. Cheung, and G. A. Constantinides, "Deep neural network approximation for custom hardware: Where We've Been, Where We're going," *ACM Computing Surveys*, vol. 52, no. May, pp. 1–39, 2019.
[6] S. Mittal, "A survey of FPGA-based accelerators for convolutional neural networks," *Neural Computing and Applications*, vol. 32, no. 4, pp. 1109–1139, 2020.
[7] C. Te Ewe, P. Y. K. Cheung, and G. A. Constantinides, "LNCS 3203 - Dual Fixed-Point: An Efficient Alternative to Floating-Point Computation," in *Field Programmable Logic and Application*. Springer Berlin Heidelberg, 2004, pp. 200–208.
[8] J. Redmon and A. Farhadi, "YOLO9000: Better, Faster, Stronger," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2017.
[9] M. P. Véstias, R. P. Duarte, J. T. de Sousa, and H. C. Neto, "A fast and scalable architecture to run convolutional neural networks in low density FPGAs," *Microprocessors and Microsystems*, vol. 77, 2020.
[10] C. Su, S. Zhou, L. Feng, and W. Zhang, "Towards high performance low bitwidth training for deep neural networks," *Journal of Semiconductors*, vol. 41, no. 2, 2020.
[11] G. A. Vera, M. Pattichis, and J. Lyke, "A dynamic dual fixed-point arithmetic architecture for FPGAs," *International Journal of Reconfigurable Computing*, vol. 2011, 2011.
[12] A. Jacoby and D. Llamocca, "Dual fixed-point CORDIC processor: Architecture and FPGA implementation," in *2016 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*, 2016, pp. 1–8.
[13] ——, "Dynamic dual fixed-point CORDIC implementation," in *2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, 2017, pp. 235–240.
[14] T.-Y. Lin, M. Maire, S. Belongie, J. Hays, P. Perona, D. Ramanan, P. Dollár, and C. L. Zitnick, "Microsoft COCO: Common Objects in Context," in *Computer Vision – ECCV 2014*, D. Fleet, T. Pajdla, B. Schiele, and T. Tuytelaars, Eds. Cham: Springer International Publishing, 2014, pp. 740–755.

Authorized licensed use limited to: Tallinn University of Technology. Downloaded on December 07,2021 at 21:06:20 UTC from IEEE Xplore. Restrictions apply.

Curriculum Vitae

1. Personal data

Name	Madis Kerner
Date and place of birth	18 December 1977 Tallinn, Estonia
Nationality	Estonian

2. Contact information

Address	Tallinn University of Technology, School of Information Technologies, Department of Computer Systems, ICT-509, Ehitajate tee 15A, 12618 Tallinn, Estonia
Phone	+372 620 2267
E-mail	madis.kerner@ttu.ee

3. Education

2017–2023	Tallinn University of Technology, School of Information Technologies, Computer Systems, PhD studies
2015–2017	Tallinn University of Technology, School of Information Technologies, Computer Systems, MSc *cum laude*
1996–2001	Tallinn University of Technology, Institute of Computing, Computer Systems, BSc

4. Language competence

Estonian	native
English	fluent
Finnish	fluent

5. Professional employment

2022– ...	Liewenthal Electronics Ltd., Senior embedded software/FPGA engineer
2010–2022	Teleplan Estonia OÜ, Embedded design Engineer
2008–2010	IPTE Estonia OÜ, Embedded design Engineer
2005–2008	JOT Automation, Embedded design Engineer
2003–2005	Orbis Estonia OÜ, Electronics design Engineer
2001–2003	PMJ-Orbis Hong Kong Ltd., Electronics design Engineer
2000–2001	Orbis Estonia OÜ, Electronics design Engineer

6. Computer skills

- Operating systems: Window, Linux, macOS
- Document preparation: vim, LaTeX
- Programming languages: C, C++, C#, ARM assembler
- Hardware description languages: VHDL, Verilog, System Verilog
- Scientific packages: MATLAB

7. Honours and awards

- 2020, HiPEAC award for:
  M. Kerner, K. Tammemae, J. Raik, and T. Hollstein, "An Efficient FPGA-based Architecture for Contractive Autoencoders," in *2020 IEEE 28th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pp. 230–230, Institute of Electrical and Electronics Engineers Inc., 5 2020

9. Scientific work
Papers

1. M. Kerner, K. Tammemae, J. Raik, and T. Hollstein, "An Efficient FPGA-based Architecture for Contractive Autoencoders," in *2020 IEEE 28th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pp. 230–230, Institute of Electrical and Electronics Engineers Inc., 5 2020

2. M. Kerner, K. Tammemae, J. Raik, and T. Hollstein, "Novel Architectures for Contractive Autoencoders with Embedded Learning," in *2020 17th Biennial Baltic Electronics Conference (BEC)*, vol. 2020-October, pp. 1–6, IEEE Computer Society, 10 2020

3. M. Kerner, K. Tammemae, J. Raik, and T. Hollstein, "Triple Fixed-Point MAC Unit for Deep Learning," in *2021 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, vol. 2021-February, pp. 1404–1407, Institute of Electrical and Electronics Engineers Inc., 2 2021

Elulookirjeldus

1. Isikuandmed

Nimi	Madis Kerner
Sünniaeg ja -koht	18.12.1977, Tallinn, Eesti
Kodakondsus	Eesti

2. Kontaktandmed

Aadress	Tallinna Tehnikaülikool, Usaldusväärsete arvutisüsteemide keskus, Arvutisüsteemide instituut Ehitajate tee 15A, 12618 Tallinn, Estonia
Telefon	+372 620 2267
E-port	madis.kerner@ttu.ee

3. Haridus

2013–…	Tallinna Tehnikaülikool, Informaatika teaduskond, Arvutisüsteemide instituud, doktoriõpe
2011–2013	Tallinna Tehnikaülikool, Informaatika teaduskond, Arvutisüsteemide instituud, MSc *cum laude*
2008–2011	Tallinna Tehnikaülikool, Informaatika teaduskond, Arvutisüsteemide instituud, BSc

4. Keelteoskus

eesti keel	emakeel
inglise keel	kõrgtase
soome keel	kõrgtase

5. Teenistuskäik

2022– …	Liewenthal Electronics Ltd., sardsüsteemide tarkvara ja FPGA insener
2010–2022	Teleplan Estonia OÜ, sardsüsteemide insener
2008–2010	IPTE Estonia OÜ, sardsüsteemide insener
2005–2008	JOT Automation, sardsüsteemide insener
2003–2005	Orbis Estonia OÜ, elektroonika insener
2001–2003	PMJ-Orbis Hong Kong Ltd., elektroonika insener
2000–2001	Orbis Estonia OÜ, elektroonika insener

6. Arvutialased oskused

- Operatsioonisüsteemid: Window, Linux, macOS
- Kontoritarkvara: vim, LaTeX
- Programmeerimiskeeled: C, C++, C#, ARM assembler
- Riistvarakirjelduskeeled: VHDL, Verilog, System Verilog
- Teadustarkvara paketid: MATLAB

7. Tunnustused ja autasud

- 2020, HiPEAK tunnustus artiklile:
  M. Kerner, K. Tammemae, J. Raik, and T. Hollstein, “An Efficient FPGA-based Architecture for Contractive Autoencoders,” in *2020 IEEE 28th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pp. 230–230, Institute of Electrical and Electronics Engineers Inc., 5 2020

8. Teadustegevus

Teadusartiklite, konverentsiteeside ja konverentsiettekannete loetelu on toodud ingliskeelse elulookirjelduse juures.

ISSN 2585-6901 (PDF)
ISBN 978-9916-80-131-4 (PDF)